SIAM J. COMPUT.
Vol. 29, No. 1, pp. 1–28

MULTIPLE NONINTERACTIVE ZERO KNOWLEDGE PROOFS UNDER GENERAL ASSUMPTIONS[*]

URIEL FEIGE[†], DROR LAPIDOT[†], AND ADI SHAMIR[†]

Abstract. In this paper we show how to construct noninteractive zero knowledge proofs for any NP statement under general (rather than number theoretic) assumptions, and how to enable polynomially many provers to give polynomially many such proofs based on a single random string. Our constructions can be used in cryptographic applications in which the prover is restricted to polynomial time.

Key words. Hamiltonian cycle, witness indistinguishability

AMS subject classifications. 94A60, 68Q15

PII. S0097539792230010

1. Introduction.

1.1. Background. Blum, Feldman, and Micali [BFM] suggested the intriguing concept of *noninteractive zero knowledge* (NIZK) *proofs*, aimed at eliminating the interaction between prover and verifier in zero knowledge interactive proof systems [GMR]. The prover P writes down a zero knowledge proof that an input x belongs to a prespecified language L, and any verifier V can check the validity of this written proof against a universal publicly available random string (such as the RAND string of one million random digits), called the *common reference string*. NIZK has become an important primitive for cryptographic protocols, with applications such as signature schemes [BG] and encryption schemes secure against chosen ciphertext attack [NY].

NIZK proof systems for any NP statement were constructed in [BFM] and [DMP87], under specific number theoretic assumptions (namely, that it is difficult to distinguish products of two primes from products of three primes, or that it is difficult to decide quadratic residuosity modulo products of two primes). The main disadvantage of these *bounded* NIZK proofs is that the prover can prove only one statement of size bounded by the length of the common reference string: if polynomially many proofs are given using the same reference string, the zero knowledge property breaks down.[1] In [BDMP] it was finally shown how a single prover can give polynomially many proofs using the same reference string, but the scheme is still based on a specific number theoretic assumption: deciding quadratic residuosity (modulo composite integers whose factorization is not known) is computationally hard. Moreover, their scheme cannot support polynomially many provers.

A variation of the NIZK model was suggested by De Santis, Micali, and Persiano [DMP88]. In their *noninteractive with preprocessing* model, the verifier and prover create a common reference string (which need not look like a random string) during an interactive preliminary stage. Based on this common reference string (CRS), the prover can then prove any single NP statement (of bounded length). Unlike

[*]Received by the editors April 13, 1992; accepted for publication (in revised form) June 18, 1997; published electronically September 14, 1999. A preliminary version of this paper appeared in *Proc. 31st Symposium on Foundations of Computer Science*, 1990, pp. 308–317.
http://www.siam.org/journals/sicomp/29-1/23001.html

[†]Dept. of Applied Math. and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel (feige@wisdom.weizmann.ac.il).

[1]The method suggested in [BFM] for overcoming this difficulty was found to be flawed.

the original NIZK model, in the noninteractive with preprocessing model, the proof should look convincing only to the verifier who takes part in the initial preprocessing stage, which makes this model unsuitable for applications such as signature schemes. [DMP88] showed an implementation of this idea based on the general assumption that one-way functions exist. Under the stronger cryptographic assumption that *oblivious transfer* protocols exist, [KMO] shows how after an initial preprocessing stage, the prover can noninteractively prove polynomially many NP statements, but again the proof is verifiable only by its original recipient. [BeMi] show how to do oblivious transfer without interaction (and hence NIZK proofs, by [KMO]) in a model where the verifier is first given a special public key.

1.2. Our results. In this paper we answer the two major open questions associated with the concept of NIZK, as presented by Blum, De Santis, Micali, and Persiano [BDMP]: how to construct NIZK proof systems for any NP statement under general (rather than number theoretic) assumptions and how to enable polynomially many provers to share the same random reference string in giving such proofs.

As a preliminary result leading to our solution of the first open question, we construct (under the assumption that one-way functions exist) a very simple zero knowledge *noninteractive with preprocessing* proof for Hamiltonicity, whose efficiency is comparable with the efficiency of the interactive proofs presented by Blum [Blum] and Goldreich, Micali, and Wigderson [GMW]. In contrast, all the previously known constructions of *NIZK with preprocessing* proofs [DMP88] are more complex and less efficient than their interactive counterparts. Then, under the assumption that one-way permutations exist, we show that if the prover and verifier initially share a common random string (which we call a common reference string), then the initial preprocessing stage of our protocol can be discarded, yielding a NIZK proof for any NP statement in the original noninteractive model of Blum, Feldman, and Micali. This noninteractive protocol is the only known implementation which relies on general computational assumptions and is conceptually simpler than the number-theoretic protocols[2] presented by Blum, De Santis, Feldman, Micali, and Persiano. Under the stronger assumption that certified trapdoor permutations exist (i.e., that the prover can demonstrate that his chosen function is indeed a permutation without revealing its trapdoor), our NIZK protocol can be carried out by probabilistic polynomial time provers and thus can be used in cryptographic applications which require NIZK protocols.

As a solution to the second open problem, we show how to transform any *bounded* NIZK proof system for an *NP complete* language into a *general* NIZK proof system in which polynomially many independent provers can share the same reference string and use it to prove polynomially many statements of polynomial length. The transformation is based on the general assumption that one-way functions exist.

Independent of our work, De Santis and Yung [DY] also show how to transform bounded NIZK proof systems into general ones, although their transformation produces noninteractive proofs which are longer than ours.

In order to use NIZK proof systems in cryptographic applications it is often necessary to extend the security conditions imposed on NIZKs to withstand adaptive attacks (see [BG], [NY]). The original definitions of NIZK proof systems assume that the statements to be proved are chosen independently of the CRS, whereas the

[2]This is based on the assumption that deciding quadratic residuosity (modulo composite integers whose factorization is not known) is computationally hard.

adaptive setting allows for the possibility that statements to be proven are chosen after the CRS is given, and may depend upon the CRS. In the last section of this paper we show that our constructions also satisfy the more stringent conditions imposed by the adaptive setting.

1.3. Definitions. $A(x)$ denotes the random variable describing the output of a probabilistic algorithm A on input x. Informally, $\nu(n)$ denotes functions vanishing faster than the inverse of any polynomial; i.e., $f(n) \leq \nu(n)$ is shorthand notation for

$$\forall d\ \exists N \text{ s.t. } \forall n > N\ \ 0 \leq f(n) < \frac{1}{n^d}$$

and $f(n) \geq 1 - \nu(n)$ is shorthand notation for

$$\forall d\ \exists N \text{ s.t. } \forall n > N\ \ 1 \geq f(n) > 1 - \frac{1}{n^d}.$$

DEFINITION 1.1. *A binary relation R is* polynomially bounded *if it is decidable in polynomial time and also there is a polynomial p such that for all $(x, w) \in R$ it is the case that $|w| \leq p(|x|)$. For any such relation and any x we let $w(x) = \{w : (x, w) \in R\}$ denote the witness set of x. We let $L_R = \{x \mid \exists w \text{ s.t. } (x, w) \in R\}$.*

R will denote a polynomially bounded relation in what follows. Note that if R is polynomially bounded, then L_R is in NP.

A NIZK proof system for NP allows a prover P to use a publicly available random string (the *CRS*) in order to prove in writing (without interaction) any NP theorem, without revealing any knowledge besides the validity of the theorem. Any polynomial time verifier V with access to the CRS can verify the validity of the proof.

The input of P is a triple (x, ω, σ) where $(x, \omega) \in R$, R is a polynomially testable relation, and σ is the CRS. Its output $P(x, \omega, \sigma)$ is a noninteractive proof (based on the witness ω, with respect to the CRS σ) that $x \in L_R$. The initial input of V (before receiving P's proof) is the pair (x, σ). Let $|x| = n$ denote the size of the common input x. Let $V(x, \sigma, P(x, \omega, \sigma))$ denote the output of the verifier V, after receiving the noninteractive proof $P(x, \omega, \sigma)$. This output may be either "accept" or "reject." For brevity of notation, we sometimes do not explicitly specify x and σ as inputs to V, when x and σ are clear from the context.

As in the case of interactive proofs, noninteractive proofs satisfy the *completeness* and the *soundness* conditions: if $x \in L_R$ then P's proof causes V to accept, and if $x \notin L_R$ the probability that V accepts P's output is negligible. The following is the formal definition of a noninteractive proof.

DEFINITION 1.2. *A noninteractive proof system for an NP language L_R is a pair of probabilistic algorithms (P, V) (where V is polynomial time) satisfying the following conditions.*

There exist two integers $b, c \geq 1$ such that the following hold:

(1) Completeness. $\forall (x, \omega) \in R$, $\forall \sigma$ *of length* $n^b k^c$ $V(x, \sigma, P(x, \omega, \sigma)) = \textit{accept}$.

(2) Soundness. *If σ is a random string, then the probability of succeeding in proving a false statement is negligible, even if the theorem is chosen by P after seeing σ.*[3] *Formally, $\forall n \geq 1$ at least $(1 - \nu(k))$ of the strings σ of length $n^b k^c$ satisfy*

$$\forall x \in (\{0,1\}^n - L_R) \qquad \forall y \quad V(x, \sigma, y) = \textit{reject}.$$

[3]In nonadaptive definitions of soundness, the prover could produce a false statement based on the choice of the random string, whose proof (associated with this particular string) will be accepted. Our definition of soundness disallows this possibility.

Remark. In the definition above (and in Definition 1.3 below), k is a security parameter (known to all parties) that quantifies how "sound" a noninteractive proof must be (or quantifies the "zero knowledge" property in Definition 1.3). More formally, we assume that the value of k (represented in unary notation as 1^k) is an additional input provided to all the algorithms, but we do not make this dependence explicit in order to simplify our notation. It is often convenient (and is the practice of most other papers on zero knowledge) to choose $k = n$ and require the desired properties of noninteractive proof systems to hold for "large enough n."

As in the case of interactive proofs (see [GMR]), the formal definition of NIZK proofs involves the notion of a probabilistic expected polynomial time simulator M, whose input is just the common input x of P and V (without an appropriate witness ω), and its output $M(x)$ consists of two strings: one of them simulates the common (random) reference string, and the other one simulates the real noninteractive proof (sent by P). Informally, a noninteractive proof is zero knowledge if such a pair of strings is computationally indistinguishable from what V sees in the actual noninteractive proof which is $(\sigma, P(x, \omega, \sigma))$. The following is the formal definition of NIZK.

DEFINITION 1.3. *A noninteractive proof system for an NP-language L_R is zero knowledge if there exists a probabilistic machine M (called a simulator) such that for any $x \in L_R$, $M(x)$ terminates in expected polynomial time and the two ensembles $\{(\sigma, P(x, \omega, \sigma))\}$[4] and $\{M(x)\}$[5] are computationally indistinguishable on L_R by any nonuniform polynomial time distinguisher $D = \{D_l\}_{l \geq 1}$:*

$$\exists b, c \geq 1 \quad \exists M \, \forall D = \{D_l\}_{l \geq 1} \, \forall (x, \omega) \in R \, \forall d \geq 1 \quad \exists K \geq 1 \, \forall k > \ \mathrm{Max}(K, |x|),$$

$$|Pr(D_k(M(x)) = 1) - Pr(D_k(\sigma, P(x, \omega, \sigma)) = 1)| < \frac{1}{k^d},$$

where the probability space is taken over the random choices of $\sigma \in_R \{0,1\}^{|x|^b k^c}$ and over the random tapes of P and M.

Remark. A nonuniform algorithm $D = \{D_l\}_{l \geq 1}$ is an algorithm that for every input length l gets an auxiliary input ("advice") of length polynomial in l. These algorithms are equivalent to polynomial size circuit families.

In cryptographic applications we would like to use efficient protocols for both P and V. The term *NIZK proof systems with efficient provers* denotes NIZK proof systems in which the truthful prover (in the *completeness* condition) is probabilistic polynomial time (in n, the length of the input x, and in k, the security parameter). "Cheating" provers (in the *soundness* condition) are never required to be computationally bounded.

2. NIZKs under general cryptographic assumptions.

2.1. A NIZK proof with preprocessing. In this subsection we present a protocol for a different model which is called a NIZK proof with preprocessing. This is not the final protocol: it is presented here just as an intermediate step in order to facilitate the understanding of the final NIZK proof system.

Consider a prover who wants to prove the Hamiltonicity of an arbitrary graph G with n nodes. We assume that the prover and the verifier are allowed to execute a

[4]Every element in this ensemble is uniquely determined by the choice of $\sigma \in_R \{0,1\}^{|x|^b k^c}$ and the prover's random tape.

[5]Here, every element is uniquely determined by the value of the random tape of the probabilistic machine M.

preliminary interactive stage which is independent of G (i.e., at this stage they know that in the noninteractive stage the prover will prove the Hamiltonicity of an n node graph, but they don't know which graph it will be). Only after the termination of this interactive stage they get G and execute the noninteractive move in which the prover sends a written message to the verifier in order to convince him in zero knowledge that G is Hamiltonian. The verifier is not allowed to ask the prover any questions and should be convinced just by reading this message.

The basic step. Let H be a randomly chosen directed Hamiltonian cycle on n nodes. We call such a graph a *good graph.* Note that this is a cyclic list without any starting point. The adjacency matrix of H is a matrix in which each row and each column contains exactly one entry that is set to 1, and the locations of the entries that are 1 define a permutation with a single cycle. Let S be the adjacency matrix of a good graph in which each entry is replaced by a string that hides it (for example, by the hard bit construction of [GL] or by a probabilistic encryption), so that a polynomially bounded observer cannot determine the locations of the ones.

Assume now that S is given to both P and V, and that P wants to prove to V that some n-nodes graph G is Hamiltonian. Since P is infinitely powerful, he can recover (to himself) the hidden 0/1 values of S which define the Hamiltonian cycle H in the good graph and determine a permutation π on the nodes of G such that H is a Hamiltonian cycle in $\pi(G)$ (i.e., $H \subseteq \pi(G)$).

In order to convince V that G is Hamiltonian, P just sends it a message which consists of the permutation π and the decryptions of all entries $S_{i,j}$ in the good matrix S for which $(i,j) \notin E(\pi(G))$. V accepts the proof iff all the revealed entries are 0. The proof system is *complete* since P can carry it out and V will accept it when G is indeed Hamiltonian. The proof system is *sound* because V's acceptance implies that all the n ones that remain unrevealed in S (and define a Hamiltonian cycle) correspond to edges of $\pi(G)$, which means that $\pi(G)$ contains a Hamiltonian cycle and thus the common input graph G is Hamiltonian.

We informally argue that the proof is *zero knowledge.* All that the verifier receives is a random permutation π and a collection of random encryptions (the entries of S) along with the decryption of those $S_{i,j}$ for which $(i,j) \notin E(\pi(G))$. All these decrypted values are 0. Since the original good matrix H (which defines the 0/1 values of S) is randomly chosen with uniform distribution (among the $(n-1)!$ possibilities), then so is $\pi \in_R Sym(n)$ (the permutation group on $[1 \ldots n]$). This follows from the fact that any two different Hamiltonian cycles H and H' determine two disjoint sets A_H and $A_{H'}$ of n permutations, where each permutation in A_H ($A_{H'}$) maps the Hamiltonian cycle of G onto H (H').[6] Therefore, for any permutation in $Sym(n)$, the probability that V receives it is $\frac{1}{n!}$. Therefore, this protocol can be easily simulated: the simulator (whose input is just the graph G) chooses a random permutation $\pi \in_R Sym(n)$ with uniform distribution, chooses random 0/1 values for all entries $S'_{i,j}$ for which $(i,j) \in E(\pi(G))$, fixes all the others to be 0, and produces random encryptions for all entries of S'. Then the simulator outputs π and the decryptions of all entries $S'_{i,j}$ for which $(i,j) \notin E(\pi(G))$. Since π is uniformally chosen in $Sym(n)$ and all the above encryptions are randomized, this simulation is computationally indistinguishable from the real proof.

Based on the above, we construct a NIZK proof system with preprocessing (regardless of whether P is efficient or not): In the preliminary interactive stage P sequentially sends k (= security parameter) good random matrices $S_1, S_2, \ldots, S_k$ to

[6]There may be several Hamiltonian cycles in G, but we concentrate on any one of them.

V and receives k random bits $b_1, b_2, \ldots, b_k$ from V. In the noninteractive move it reveals all entries of those S_i's for which $b_i = 0$ and executes the above basic step for those S_i for which $b_i = 1$. If all the S_i with $b_i = 0$ are good (i.e., hidden adjacency matrices of Hamiltonian cycles), then V can conclude with high probability that at least one of the other S_i is also good, in which case G is guaranteed to be Hamiltonian.

In order to compare this protocol with Blum's protocol for Hamiltonicity [Blum], let us recall that in the first move of Blum's scheme P randomly permutes G and sends V the encrypted adjacency matrix of this isomorphic copy. V then sends a random bit to P and according to that bit P either reveals all the entries in the matrix and the permutation or reveals only the entries which correspond to the edges of the Hamiltonian cycle. Our protocol resembles Blum's protocol, with one major difference: in Blum's protocol all the moves depend on G, while in our protocol only the last move depends on G. As a result, Blum's protocol cannot be split into a preprocessing stage and a noninteractive proof as we did in our protocol.

Remark. This particular NIZK proof with preprocessing can be extended for *directly* proving (without reduction to the Hamiltonian problem) a variety of graph theoretic problems which are satisfied by a single minimal (or maximal) graph (under isomorphism). This family includes clique, graph coloring, graph partition into k-cliques, three-dimensional matching, etc. We don't know how to extend our proof technique directly to other NP-complete problems.

2.2. A NIZK proof with a common reference string. In this subsection we show that under the assumption that (strong) one-way permutations exist, if the prover and the verifier initially share a random string (or CRS) σ, then the initial preprocessing stage of the protocol described in section 2.1 can be discarded, yielding a NIZK proof for any NP statement in the noninteractive model of Blum, Feldman, and Micali.

DEFINITION 2.1. *A permutation f is a (strong)* one-way function *if for any x ($|x| = n$), $f(x)$ (which is also of length n) can be computed in polynomial time, but for any nonuniform polynomial time algorithm A,*

$$Prob(A(y) = f^{-1}(y)) \leq \nu(n),$$

where the probability is computed over the random choices of $y \in_R \{0,1\}^n$.

We remark that strong one-way permutations exist if "weak" one-way permutations (i.e., the probability of not inverting y is nonnegligible) exist [Yao], [GILVZ].

In our NIZK construction we want to guarantee the hardness of inverting the one-way permutation f at the single bit level. This idea is captured in the notion of *a hard-core predicate* of a one-way function. A hard-core predicate of a function f is a predicate $B : \{0,1\}^* \longrightarrow \{0,1\}$, which is efficiently computable but such that given only $f(x)$ it is hard to guess $B(x)$ with a probability significantly better than 1/2.

DEFINITION 2.2. *We call the predicate $B : \{0,1\}^* \longrightarrow \{0,1\}$ a* hard-core predicate *of the function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ if the following conditions are satisfied:*

1. *B is computable in deterministic polynomial time.*
2. *For every nonuniform polynomial time probabilistic algorithm A, for every integer $c > 0$, and for every large enough n,*

$$Prob\{A(f(x)) = B(x)\} < \frac{1}{2} + \frac{1}{n^c},$$

where the probability is taken over the coin tosses of A and for x uniformly chosen in $\{0,1\}^n$.

The idea of hard-core predicates was introduced and first implemented (based on a specific one-way permutation) by [BM]. [Yao] presented a general transformation of any (strong) one-way function into one which has a hard-core predicate, but the transformation was impractical. Goldreich and Levin [GL] provided an alternative transformation which was much more efficient. Our NIZK construction uses such a hard-core predicate in order to extract hard-to-guess bits from any given one-way function.

2.2.1. Informal description. Assume that P and V possess a CRS σ, and P wants to send V a NIZK proof based on σ (rather than on an interactive preprocessing stage) that an arbitrary n node graph G is Hamiltonian. Basically, the proof technique consists of two stages.

1. Interpretation of the CRS as an encoding of a string of "secret" bits—a hidden random string (HRS). Only the unbounded prover can initially read the hidden bits, but he can later selectively reveal the value of some of the hidden bits to the polynomially bounded verifier without revealing any information on the other hidden bits.
2. Interpretation of the HRS as a sequence of $n \times n$ matrices in such a way that at least one of them represents a good graph with overwhelming probability. For each matrix the prover P is allowed to do one of two things: to prove to V that the matrix is not good by revealing all its entries, or to use it as if it is a good matrix in the Hamiltonicity testing protocol of section 2.1. Note that the prover can claim that a bad matrix is good, but he cannot successfully claim that a good matrix is bad, and thus he will be forced to use all the good matrices in the protocol. If the input G is non-Hamiltonian, P will fail to convince V except in the extremely unlikely case in which all the matrices defined by the HRS are bad.

The first stage is implemented by considering the CRS as a concatenation of polynomially many blocks $u_1, u_2, \ldots$, where each block contains k (= security parameter) random bits. We define a corresponding intermediate random string (IRS) by concatenating the values of $w_1, w_2, \ldots$, where each w_i is equal to $f^{-1}(u_i)$. The ith bit s_i in the HRS is a hard-core bit defined by the ith block u_i in the CRS. By revealing s_i we mean that P sends to V the bit s_i along with w_i from the IRS. By checking s_i we mean that V checks that $f(w_i) = u_i$ and $B(w_i) = s_i$. Note that $f^{-1}(u_i)$ exists and is uniquely defined by our assumption that f is a one-way permutation, and thus even the unbounded prover cannot "flip" the value of this bit without being caught by the verifier.

The second stage is implemented by interpreting the HRS as a sequence of $n \times n$ 0/1 matrices which with overwhelming probability contain at least one good matrix. Notice that if we naively interpret each block of n^2 consecutive bits from the HRS as an $n \times n$ 0/1 matrix, then the probability that even one of these polynomially many matrices is good is exponentially low. Therefore, we have to encode our matrices in a more complicated way.

Assume that the HRS can be partitioned into equal size segments $z_1, z_2, z_3, \ldots$, each of which defines (in some way) an $n^2 \times n^2$ matrix B_i of zeros and ones, such that for each i and each (j, l) the probability that the (j, l)th entry in B_i is 1 equals $\frac{1}{n^3}$ (i.e., $Pr\{B_i(j, l) = 1\} = \frac{1}{n^3}$). Therefore, for any segment z_i the expected number of 1-entries in the corresponding matrix B_i is n, and we will later prove that every B_i contains, with nonnegligible probability, exactly n rows and n columns, each of which contains a single 1-entry and the $n \times n$ permutation matrix induced by these rows and columns is Hamiltonian. Therefore, if the length of the HRS is large enough (polynomial in n and k), then, with high probability, at least one of its segments defines a good matrix.

All we have to show is how to transform a given random string into a sequence of matrices, each of which has the property of B_i. Consider the given random string as a concatenation of polynomially many consecutive blocks of m bits where $m = \log(n^3)$ (w.l.o.g. we can assume that it is an integer). We interpret a block as 1 if all its m bits are 1, and 0 otherwise. Therefore, for every m-bit block, the probability of being interpreted as 1 is $\frac{1}{n^3}$ and thus we can pack each consecutive segment of n^4m random bits into the desired $n^2 \times n^2$ 0/1 matrix B_i discussed above.

Informally, the proof technique is the following: for each matrix B_i, the prover must either prove that it contains no good $n \times n$ submatrix or execute the basic step (described in section 2.1) on a good matrix derived from B_i. In order to construct a good matrix from a given matrix $B = B_i$ and to prove that the input n-node graph G is Hamiltonian, P executes the following.

1. If the number of ones in B is different from n or there exists a row or a column which contains at least two ones, then P proves this fact by revealing all entries in B. Otherwise (i.e., B contains an $n \times n$ permutation submatrix), P reveals all entries in the n^2-n rows and the n^2-n columns which contain only zeros, and removes them from B. If the resulting $n \times n$ Boolean matrix does not represent a permutation with a single cycle (i.e., it is not an adjacency matrix of some Hamiltonian cycle), then P proves this fact by revealing all entries of the remaining $n \times n$ matrix.
2. Otherwise (i.e., the remaining $n \times n$ matrix forms an adjacency matrix of some Hamiltonian cycle), P must execute the basic protocol (described in the previous section) on the resulting $n \times n$ good matrix.

In order to formally describe the scheme and prove its correctness, we introduce some notation and definitions.

2.2.2. Notation and definitions. Let

$$\sigma = r_1 \dots r_{poly(k,n)} \qquad r_i \in_R \{0,1\}$$

$$= u_1 \dots u_{poly(k,n)} \qquad u_i \in_R \{0,1\}^k$$

be the CRS, shared by P and V. Let f be a (strong) one-way permutation, and let s_i be the hard bit that corresponds to the k-bit string u_i (with respect to f).

We associate with the CRS an IRS defined by $(f^{-1}(u_1), \dots, f^{-1}(u_{poly(k,n)}))$. According to the definition of a hard bit (see [GL]) each s_i (of the HRS) is polynomially computable from the corresponding $f^{-1}(u_i)$ (of the IRS) which acts as its "witness."

For each $i \geq 1$ let

$$b_i = \bigwedge_{j=1}^{m} s_{(i-1)m+j},$$

where $m = \log(n^3)$.

Let B_i be an $n^2 \times n^2$ matrix which is defined as follows: $B_i(j,l) = b_{(i-1)n^4+(j-1)n^2+l}$ for every $1 \leq i, j, l$.

DEFINITION 2.3. *We say that B_i is a proper matrix if it contains exactly n ones and each column and row contains at most a single one.*

If B_i is a proper matrix let N_i be the $n \times n$ matrix obtained by removing all the $n^2 - n$ columns and $n^2 - n$ rows which contain only zeros. Otherwise N_i is undefined.

DEFINITION 2.4. *A Boolean $n \times n$ matrix is called good if it is the adjacency matrix of a graph which consists of a single directed cycle passing through all the n vertices.*

With some abuse of notation, we also call the large $n^2 \times n^2$ matrix B_i good if it is proper, and if the (unique) $n \times n$ submatrix N_i it defines is good.

2.2.3. The scheme. Assume that P and V have a CRS with $2n^7k^2m$ bits and fix some one-way permutation f.

P's protocol. For each $1 \leq i \leq kn^3$ do the following:

1. If the matrix B_i is not good then reveal all its entries.
2. Otherwise (B_i contains a good $n \times n$ submatrix N_i), reveal and remove (the entries of) all the $n^2 - n$ columns and all the $n^2 - n$ rows which contain only zeros, and execute the noninteractive stage (of the protocol described in section 2.1) on the remaining good matrix N_i.

V's protocol. For each $1 \leq i \leq kn^3$ do the following:

1. If P reveals all entries of B_i then check that the revealed bits are correct and that the matrix they define is not a good matrix.
2. Otherwise, P should reveal $n^2 - n$ columns and $n^2 - n$ rows: check that the revealed bits are correct and that all entries they define in these rows and columns are zeros; in addition, P should execute in this case the basic protocol (described in section 2.1) on the remaining hidden $n \times n$ matrix: check that this proof is carried out correctly.

Accept the proof iff each of these kn^3 checks is successful.

2.3. Correctness.

2.3.1. Completeness. If the input graph is indeed Hamiltonian, then the prover can execute correctly the proof with respect to each one of the good matrices (if they exist). In each $n^2 \times n^2$ matrix that does not yield a good submatrix, P is just required to reveal the entire matrix and V will accept its proof as valid. As a result, an honest prover never fails.

2.3.2. Soundness.

CLAIM 2.5. *For every i, the probability that exactly n entries of B_i are 1 is at least $\frac{1}{4\sqrt{n}}$.*

Proof. The bits of the HRS are unbiased and independent, and for each j the probability that $b_j = 1$ is $1/n^3$. Therefore, the probability that B_i has exactly n ones is

$$\binom{n^4}{n} \left(\frac{1}{n^3}\right)^n \left(1 - \frac{1}{n^3}\right)^{n^4 - n} > \frac{1}{4\sqrt{n}}$$

for all sufficiently large n. □

The size of B_i is $n^2 \times n^2$ and the 0/1 value of each entry is determined independently of the others since these bits are determined by nonoverlapping blocks from the random CRS. Assume now that B_i has exactly n ones. Its entries are independent of each other, and thus the locations of the n ones in the n^2 rows of B_i can be viewed as the result of placing n balls in n^2 buckets. The same argument holds also for the columns. By the birthday paradox, with constant probability, the n ones are located in n distinct rows and n distinct columns, and thus with constant probability a matrix B_i with n ones is proper.

The number of permutations in $Sym(n)$ which consist of a single cycle (of length n) is $(n-1)!$. Therefore, the probability that N_i is a good matrix, given that it is a permutation matrix, is n^{-1}.

We conclude that, for every i, the probability that B_i is good is at least $\geq dn^{-3/2}$, where d is a constant. Thus if the length of the CRS is $\Omega(n^{13/2}k^2m)$ bits, then with probability $(1-e^{-nk})$ at least one of the B_i's yields a good matrix. Any such matrix will expose a cheating P, since it cannot prove that B_i is bad and cannot use it in the basic step, and these are its only two options.

Remark. If $\log(n^3)$ is not an integer, set $m = \lceil \log(n^3) \rceil$ and choose B_i as a $\lceil bn^2 \rceil \times n^2$ matrix where $b = \frac{2^m}{n^3}$ $(1 < b < 2)$.

2.3.3. Zero knowledge. We construct a random polynomial time simulator M which generates a "random string" and a "proof" of Hamiltonicity which are polynomially indistinguishable (by nonuniform distinguishers) from those generated by a real execution of the protocol.

Consider the task of M. Compared with the real prover P, it is handicapped in two respects: It cannot invert one-way functions (and thus cannot expose the HRS defined by the given CRS), and it may not know a Hamiltonian cycle in G. We solve one problem at a time and use the transitivity of indistinguishability to prove that the two solutions can be combined into one. First we construct a random polynomial time algorithm P' that cannot invert one-way functions, but does have access to the Hamiltonian cycle of G. P' uniformally generates a CRS in such a way that it can recover its associated HRS. Since the original CRS (appended to a proof of the real prover) is also uniformally chosen, these two strings are identically distributed. Next we construct a probabilistic expected polynomial time simulator M whose input is the Hamiltonian graph G (without its Hamiltonian cycle) and whose output is polynomially indistinguishable from that of P'. Therefore, these constructions imply that our scheme is zero knowledge.

Let P' be the random polynomial time algorithm whose input consists of the graph G along with its Hamiltonian cycle. The instructions of the original P and the definition of the one-way permutation f (which is fixed in the original proof) are parts of P'. P' executes the real protocol with the following exception: instead of using the given random CRS to compute its associated IRS, it chooses a truly random IRS and computes its associated CRS by applying the one-way permutation f (in the forward direction) to each consecutive block of k bits in the IRS. Namely, for each segment v_i ($v_i \in_R \{0,1\}^k$) in the IRS, P' evaluates $f(v_i)$ and set $f(v_i)$ to be the ith k-segment of the new CRS. The output of P' consists of his CRS accompanied by a noninteractive proof for the Hamiltonicity of G (which is performed exactly according to the instructions of the real P). Since both the original CRS (which is used by P) and that produced by P' are uniformally distributed random strings, and P' and P behave identically once the CRS is determined, we conclude that the output of P' is indistinguishable from the original CRS followed by the real prover's proof.

We now change P' further to get the simulator M. This simulator accepts a Hamiltonian graph G and the security parameter k as inputs but is not given any Hamiltonian cycle in G. Its output is a string σ_k of length n^7k^2m bits and a "proof" of Hamiltonicity based on this CRS. The basic idea behind the simulator is that it tries to execute the protocol and changes the CRS in an indistinguishable way whenever it encounters difficulties. To do this, it leaves unchanged all the bits of the HRS which were part of bad matrices but changes to zero all the bits of the HRS which were part of good matrices. This would allow the simulator to successfully carry out both stages of the proof (namely, demonstrating the unsuitability of the bad matrices by revealing all their entries and revealing the required zero entries in all the presumably good remaining matrices). To change an HRS bit to zero, the simulator repeatedly tries new random IRS values until it finds one which makes the corresponding HRS bit zero and then replaces the corresponding CRS block by f applied to the new IRS value (which remains random and indistinguishable from the original value). More precisely, the simulator M performs the following steps:

1. M randomly chooses a sequence of n^7k^2m truly random bits and uses them as the IRS. Every segment in this IRS that yields a good matrix M changes all entries whose value is 1 to zeros. This is done in the following way: for each i for which N_i is a good matrix and for each j, l such that $N_i(j,l) = 1$, M executes the following trial: it replaces all bits in the IRS which give rise to this $N_i(j,l)$ by new random km bits. M repeatedly executes this trial until $N_i(j,l) = 0$ (the probability of success is $1 - \frac{1}{n^3}$).
2. M computes the CRS σ_k from the modified IRS by applying f in the forward direction and then computes the HRS from the IRS in exactly the same way as it is done by P.
3. For each i such that B_i has not been changed in the first step, M reveals all entries of B_i. For each of the other B_i's, M reveals $n^2 - n$ random rows and $n^2 - n$ random columns. Since the resulting $n \times n$ matrix contains only zeros, M can easily simulate the basic step by choosing a random permutation $\psi \in_R Sym(n)$ and revealing every $B_i(j,l)$ such that there is no edge between j and l in $\psi(G)$.

The output of M is denoted by $(\sigma_k, proof'(\sigma_k, G))$, where the first component plays the role of the CRS and the second one includes all revealed bits and permutations. Let τ_k be a string of length n^7k^2m bit which is produced by P' as a CRS and denote by $proof(\tau_k, G)$ a proof of P' based on G and τ_k.

For any nonuniform distinguisher D, let $D(x)$ denote the 0/1 output of D on input x. Let

$$\rho^D_{P',k,G} = Pr\{D((\tau_k, proof(\tau_k, G)), G) = 1\},$$

$$\rho^D_{M,k,G} = Pr\{D((\sigma_k, proof'(\sigma_k, G)), G) = 1\},$$

where the probabilities are taken over the random tapes of P' and M and thus also over the random choices of τ_k and σ_k (which are chosen by P' and M, respectively).

LEMMA 2.6. *For any polynomial Q and any positive integer t, there exists a positive integer K, such that for any Hamiltonian graph G of size n, for any nonuniform distinguisher D, and for any $k \geq \max(K, n)$,*

$$|\rho^D_{P',k,G} - \rho^D_{M,k,G}| < \frac{1}{Q(k)},$$

where the running time of D is bounded by k^t.

Proof. The proof is based on the well-known hybrid argument of [GM]. We assume the existence of some efficient distinguisher D, a polynomial Q, and an infinite subset of security parameters $\mathcal{I} \subset \mathcal{N}$ such that for every $k \in \mathcal{I}$,

$$(*) \quad |\rho^D_{P',k,G} - \rho^D_{M,k,G}| \geq \frac{1}{Q(k)}.$$

Our goal is to show how to use the existence of these entities in order to successfully predict some hard-core bit, in violation of the assumption that f is a one-way permutation. More precisely, we would like to construct a probabilistic polynomial time nonuniform algorithm C which, given as input $f(x)$ for a randomly chosen x, predicts its hard-core predicate $B(x)$ with nonnegligible probability of success. This C chooses a Hamiltonian graph G and constructs a proof of Hamiltonicity whose CRS is a mixture of an initial segment corresponding to a real proof by P' and a final segment corresponding to a simulated proof by M. (Note that both P' and M are polynomial time, and thus their behavior can be replicated by C.) It is not difficult to show that for some location of this boundary, our assumption implies that the distinguisher's probability of outputting 1 jumps nonnegligibly when the CRS boundary moves by a single block. If C embeds its input $f(x)$ at this crucial location in the CRS, it can use the success of the distinguisher to predict the value of the hard-core bit of its input. This will lead to a contradiction, and thus the probability distributions generated by P' and by M are polynomially indistinguishable.

From here on we omit the superscript D and the subscript G. Let k be an element in $\mathcal{I}$. Let $\alpha = (i_1, \ldots, i_t, \psi_1, \ldots, \psi_u)$ $(1 \leq i_1 < \cdots < i_t \leq n^7 km$ and for each $1 \leq i \leq u$ $\psi_i \in Sym(n))$ and let $\rho_{\alpha,k}$ $(\rho'_{\alpha,k})$ denote the probability that the hidden bits which are revealed by P' (respectively, M) are those which are indexed (in the HRS) by $i_1, \ldots, i_t$ and $\psi_1, \ldots, \psi_u$ are the permutations associated with good matrices chosen by P' (respectively, M). Since M and P' follow exactly the same procedure for choosing the locations of the revealed bits (M may change the *value* of some revealed bits from 1 to 0 by replacing their corresponding CRS blocks but does not try to move their *location*), and both of them choose truly random permutations to apply to G, we conclude that for any α,

$$\rho_{\alpha,k} = \rho'_{\alpha,k}.$$

Let $proof(\tau_k, G, \alpha)$ and $proof'(\sigma_k, G, \alpha)$ denote proofs of P' and M based on τ_k and σ_k, respectively, in which the locations of the revealed bits and the random permutations (in $Sym(n)$) are identical and are defined by $\alpha = (i_1, \ldots, i_t, \psi_1, \ldots, \psi_u)$. Denote by $\rho_{P',\alpha,k}$ the probability that D outputs 1 on $(\tau_k, proof(\tau_k, G, \alpha))$ (produced by P') and by $\rho_{M,\alpha,k}$ the probability that D outputs 1 on $(\sigma_k, proof'(\sigma_k, G, \alpha))$ (produced by M).

It is obvious that

$$(**) \quad \rho_{P',k} = \sum_{\alpha} \rho_{\alpha,k} \rho_{P',\alpha,k}$$

and

$$(***) \quad \rho_{M,k} = \sum_{\alpha} \rho_{\alpha,k} \rho_{M,\alpha,k}.$$

We now want to show that for any fixed choice of α, D is unable to distinguish between $(\tau_k, proof(\tau_k, G, \alpha))$ and $(\sigma_k, proof'(\sigma_k, G, \alpha))$.

CLAIM 2.7. *For every α,*

$$|\rho_{P',\alpha,k} - \rho_{M,\alpha,k}| < \frac{1}{Q(k)}.$$

Proof. Assume that this is not true; namely, there is α for which w.l.o.g.

$$\rho_{P',\alpha,k} - \rho_{M,\alpha,k} \geq \frac{1}{Q(k)}.$$

We now use the hybrid argument by scanning across the list of locations of revealed bits and defining a sequence of associated probabilities. For every $1 \leq j \leq n^7km$, $\rho^j_{\alpha,k}$ denotes the probability that D outputs 1 on the following (string, proof): the first $k(j-1)$ bits in the string are a prefix of a CRS generated by P' from a randomly chosen IRS, while the remaining bits in the string are generated by M from a random IRS which was modified to force zeros to appear in certain HRS revealed positions. The proof following the string is the implied combination of a prefix produced by P' and a suffix produced by M, where both parts are based on the vector α. By the hybrid argument we conclude that there is $1 \leq i \leq n^7km$ for which

$$\rho^{i+1}_{\alpha,k} - \rho^i_{\alpha,k} \geq \frac{1}{Q(k)n^7km}.$$

We'll now describe the formal construction of the probabilistic polynomial time nonuniform algorithm C_k whose auxiliary input is the graph G, including the definition of a Hamiltonian cycle, α, i, and the auxiliary input needed for D. This algorithm uses the polynomial time P', M, and D as subroutines and on input $f(x)$ (where x is randomly chosen) outputs a bit b which is the hard bit of $f(x)$ with probability $\geq \frac{1}{2} + \frac{1}{poly(k)}$. This is a contradiction to the assumption that f is oneway.

The algorithm C_k executes the following steps.

1. Run P' so that the indices of the hidden bits which are revealed and the permutations associated with the Hamiltonian matrices are according to α.
2. Run M according to the same rule.
3. Erase from the output of P' all the bits coming after the $(i-1)$th block of the CRS (call this prefix S_P).
4. Erase from the output of M the first i blocks; namely, keep the last n^7k^2m-ik bits of the CRS and append the revealed bits and the permutations associated with the Hamiltonian matrices (which are based on α and thus are identical for P' and M). Call this suffix S_M.
5. Feed D with $S_P \circ f(x) \circ S_M$.
6. If $D(S_P \circ f(x) \circ S_M) = 1$, then $b = 1$ else $b = 0$.

Without loss of generality, assume that the bit defined by the ith block of P' is 1, and M changes it to 0 (these are the only changes made by M, and unchanged locations cannot possibly trigger a reaction by the distinguisher). It is easy to verify that with probability $\geq \frac{1}{2} + \frac{1}{poly(k)}$, b is the hard bit of $f(x)$ and this is a contradiction to the assumption that f is oneway. $\square$

This claim together with $(**)$ and $(***)$ contradicts $(*)$ which completes the proof of the Lemma. $\square$

2.4. Efficient provers.

2.4.1. The scheme. If the truthful prover is restricted to be a random polynomial time machine (namely, it has the same computational power as V) then it can not invert the one-way permutation in the protocol described in the previous section. In order to overcome this difficulty we use the notion of *families of certified trapdoor permutations.* Therefore, our assumption in this section is that such families exist. Informally, a permutation is trapdoor if its values can be computed in polynomial time and it is hard to compute its inverse, but there exists an auxiliary information (which is called the trapdoor) such that there is an algorithm which gets this trapdoor information as an auxiliary input and computes the inverse of this function in polynomial time. Such a family is certified if it is easy to verify that a given function does belong to this family. The following is the formal definition of a family of certified trapdoor permutations.

DEFINITION 2.8. *Let I be an infinite set of indices. A set of functions $F = \bigcup_{k\geq 1} F_k$ where*

$$F_k = \{f_i : D_i \longrightarrow D_i \quad : \quad i \in I \bigcap \{0,1\}^k\}$$

is a family of certified trapdoor permutations *if for every $i \in I$, f_i is a permutation over the finite domain $D_i \subseteq \{0,1\}^k$ and the following conditions are satisfied:*

1. *There exists a random polynomial time generating algorithm G that on input k (in unary representation) generates a random pair $(i, t(i))$, where $i \in I \bigcap \{0,1\}^k$ (defines the function f_i) and $t(i)$ is a trapdoor information for f_i.*
2. *There exists a probabilistic polynomial time algorithm that on input i determines whether $f_i \in F$ (in particular whether f_i is a permutation).*
3. *There exists a random polynomial time algorithm that on input $i \in I$ chooses a random element $x \in D_i$ with uniform distribution over D_i. There exists a polynomial time algorithm that for any $i \in I$ and any x checks whether $x \in D_i$.*
4. *There exists a polynomial time algorithm A such that*
$$\forall i \in I \quad \forall x \in D_i \qquad A(i,x) = f_i(x).$$
5. *For any random polynomial time algorithm B, for every constant $c > 0$, and for every large enough integer k,*
$$Prob\{B(i, f_i(x)) = x\} < \frac{1}{k^c}$$
where the probability space is taken over the random tape of B combined with the distribution of i as generated by $G(1^k)$ and a random uniform choice of $x \in D_i$.
6. *There exists a polynomial time algorithm C such that*
$$C(i, t(i), f_i(x)) = x \qquad \forall x \in D_i, \quad \forall i \in I.$$

For simplicity, we assume that there exists a family of certified trapdoor permutations in which each of the domains $D_i = \{0,1\}^{n_i}$, where n_i is some integer that depends on i. (However, we emphasize that this assumption can be relaxed in several ways, such as allowing the domains D_i to cover a nonnegligible fraction of $\{0,1\}^{n_i}$, or using one-to-one functions rather than permutations. In particular, the number theoretic assumptions used in [BDMP]'s construction of NIZKs are a special case of our relaxed assumptions.) Under this assumption, the scheme described in section 2.2 can be modified to accommodate probabilistic polynomial time provers. Efficient P randomly chooses a trapdoor permutation $f \in F_k$, sends its index i (of length k) to V, and keeps the trapdoor information $t(i)$ secret. The ability of P to invert f efficiently is due to its knowledge of the trapdoor information.

The proof of *completeness* remains unchanged. The proof of *soundness* has to be modified for the following reason. In contrast to the scheme described in section 2.2 in which the (unbounded) prover does not choose the one-way permutation, in the efficient scheme a cheating prover may choose a particularly useful trapdoor permutation after seeing the CRS. Namely, he can choose a trapdoor permutation for which the corresponding hidden string HRS (determined by the given CRS and this particular trapdoor permutation) does not yield any Hamiltonian matrix. Such a string enables a cheating prover to "prove" the Hamiltonicity of *any* graph, in particular non-Hamiltonian graphs, without getting caught by the verifier.

To overcome this difficulty, we only have to extend the length of the CRS. Recall (see section 2.3.2) that for any fixed (either trapdoor or one-way) permutation, if the length of the CRS is $\Omega(n^{13/2}k^2m)$ bits, then the fraction of bad CRSs (those which do not yield any Hamiltonian matrix) is e^{-nk}. Since all random bits of the CRS are independent, we conclude that if the length of the CRS is twice as long, then the fraction of bad CRSs with respect to any fixed trapdoor permutation is less than e^{-2nk}. But recall that in the new scheme the prover can choose any trapdoor permutation he wishes, out of a family of at most 2^k trapdoor permutations. Therefore, $\frac{2^k}{e^{2nk}} < 2^{-k}$ is an upper bound on the fraction of random strings which can be bad relative to some trapdoor permutation.

The proof of the zero knowledge property of our new scheme resembles its counterpart for the original scheme (with an unbounded prover), except that we have to consider a family of certified trapdoor permutations rather than a fixed one-way permutation. Namely, the probability distribution over all pairs (CRS , proof) includes now also the uniform distribution over all trapdoor permutations in this family, rather than just the uniform distribution over all random strings. Now, the intermediate simulator P' (which has an access to a Hamiltonian cycle in G) should uniformly choose a random trapdoor permutation f (without its trapdoor information) in order to have an output's distribution identical to that of the real prover. The main simulator M (which does not know any Hamiltonian cycle in G) uses the same random f, and both P' and M should apply f in the forward direction. The rest of the proof (regarding the indistinguishability between the outputs of M and P') goes exactly as introduced in section 2.3.3.

We remark that the certification assumption (part 2 of Definition 2.8) can be relaxed. It states that *there exists a polynomial time algorithm that on input i determines whether* $f_i \in F$. This item is included to guarantee that a cheating prover cannot choose a function which does not belong to the prespecified family F of trapdoor permutations. In particular, if P sends to V a definition of a trapdoor function which is not a permutation at all, *soundness* does not necessarily hold, as the opening of the hard-core bits is not unique. Recently, Bellare and Yung [BY] constructed a NIZK proof system for proving that a given function (whose description is of polynomial size) is "almost permutation" (permutation on all but a small fraction of the domain). They showed that this implies that part 2 of the definition is not necessary for the construction of efficient NIZK proof systems for NP.

2.4.2. Public-key cryptosystems secure against chosen ciphertext attacks. The existence of public-key cryptosystems which are secure against passive eavesdropping under the assumption that certified trapdoor permutations exist is well known [GM, Yao, GL]. Naor and Yung [NY] show how to construct a public-key cryptosystem which is provably secure against chosen ciphertext attacks (CCS-PKC), given a public-key cryptosystem which is secure against passive eavesdropping and a NIZK proof system in the shared string model. Using their result together with our construction (for polynomial time provers) we have the following corollary.

COROLLARY 2.9. *CCS-PKC exist under the general assumption that certified trapdoor permutations exist.*

This is the first known CCS-PKC which does not rely on the hardness of specific computational number-theoretic problems.

3. Multiple NIZK proofs based on a single random string.

3.1. Introduction. In the previous section we constructed a *bounded* NIZK proof system, in which the prover can prove a single statement. In this section we construct a *general* NIZK proof system, in which polynomially many statements can be proven by polynomially many provers independently. Our main concern will be to control the length of the common random string σ. Recall that in the case of bounded NIZKs, the length of σ is some explicit polynomial in n (the length of the statement to be proved) and k (a security parameter). One may attempt to transform a bounded NIZK proof system into a general one by reusing the bits of σ over and over again each time a new statement is proved. Unfortunately, it is not known whether the zero knowledge property is preserved if the number of statements proven exceeds $O(\log n)$. If there is an a-priori bound m on the number of statements to be proved, then an alternative approach may be to extend σ by a factor of m. However, this solution is extremely wasteful in the length of σ (and even more if m turns out to be an overestimate). Thus throughout this paper we make the requirement that *the length of σ depends on n and k alone, but not on the number of statements to be proven, which can be an arbitrary polynomial in n.*

Notation and conventions. Throughout the following sections, n denotes the size of a single input statement, whereas m denotes the number of input statements to be proved, each of length n. The value of m is a function of n (typically, some polynomial in n), though we do not introduce special notation to denote this fact. To avoid excessive use of parameters, we identify the security parameter k with the length n of a single input statement (see the remark following Definition 1.2). We use the $\nu(n)$ notation of section 1.3 whenever it is not a source for confusion. We assume that outputs of provers (denoted by $P(x, w, \sigma)$) include explicitly the input x and the CRS σ. Recall our nonstandard use of *ensembles* (see section 1.3) in which the ensemble for the simulator is indexed by inputs $x \in L_R$, whereas the ensemble for the truthful prover is indexed by inputs $x \in L_R$, together with a valid witness w for each input. The two ensembles were said to be indistinguishable if, for any (large enough) input $x \in L_R$, the corresponding distributions (the prover's proofs and the simulator's simulated proofs) were indistinguishable, regardless of the witness w used by the truthful prover. We extend this concept of ensembles to accommodate multiple noninteractive proofs. The ensemble for the simulator is indexed by sequences of equal length inputs in L_R, where, for each sequence, its length m is bounded by some polynomial in the length n of single input statements in the sequence. The ensemble for the truthful prover is indexed by a sequence of equal length inputs in L_R, together with a sequence of corresponding valid witnesses. The two ensembles are indistinguishable if for any sequence of equal length inputs in L_R the corresponding distributions (the prover's sequence of proofs and the simulator's simulated proofs) are indistinguishable, regardless of the sequence of witnesses used by the truthful prover. For computational indistinguishability, the probability of distinguishing between the two ensembles must decrease as fast as $\nu(n)$, which is equivalent to $\nu(mn)$, by our requirement that m is polynomial in n. The distinguishing algorithm is denoted by D and runs in nonuniform polynomial time. Hence $\forall D$ quantifies over all nonuniform polynomial time algorithms. D will typically receive the output of the prover as input, which by our convention regarding $P(x, w, \sigma)$ implies that D also sees x and σ.

DEFINITION 3.1. *A noninteractive proof systems for the language L_R is* general zero knowledge *if there exists a random polynomial time simulator M such that for* *any positive constant c, for any $m \leq n^c$, the two ensembles $\{(\sigma, P(x_1, w_1, \sigma), \ldots, P(x_m, w_m, \sigma))\}$ and $\{M(x_1, x_2, \ldots, x_m)\}$ are computationally indistinguishable. In any sequence of instances that indexes the ensembles, all x_i are of the same length (denoted by n), and for all $1 \leq i \leq m$, $(x_i, w_i) \in R$.*

Remark. An important feature of our definition of general zero knowledge noninteractive proof systems is that each of the statements x_j is proven independently. Consequently, polynomially many provers can share the same random reference string σ and prove polynomially many statements independently. A somewhat weaker definition, in which the proof of statement x_j may depend on the proof of previous statements, is given in [BDMP]. Their definition applies only to the case that a single prover uses σ to prove polynomially many statements. The construction that they propose does not support polynomially many independent provers (i.e., our stronger definition).

In this section we show how to transform any bounded NIZK proof system for an NP complete language L_R into a general NIZK proof system for the same language L_R. Our transformation uses the NP-completeness of L_R in an essential way, and does not handle cases in which L_R is not NP-complete. Our transformation applies only to NIZK proof systems with efficient provers (and does not apply to NIZK proof systems such as that of section 2.2 in which P inverts one-way permutations).

We now give a quick overview of our construction. It is based on the concept of *witness indistinguishability* [FS], which informally means that it is intractable to distinguish which of two possible witnesses P is using in his proof of an NP statement. We prove that any NIZK proof system with efficient provers is also witness indistinguishable. Furthermore, the witness indistinguishability property is preserved even if polynomially many noninteractive witness indistinguishable proofs are given using the same reference string (again, provided that the prover in each individual proof is efficient).

If one could argue that any sequence of noninteractive witness indistinguishable proofs is also zero knowledge then we would be done. It is not true that in general witness indistinguishability implies zero knowledge, but there are special cases where this implication holds. We show, under the assumption that one-way functions exist, that any NIZK proof system for any NP-complete language can be modified to a new noninteractive proof system for which witness indistinguishability always implies zero knowledge.

3.2. Noninteractive witness indistinguishability. In this subsection we define the concept of noninteractive witness indistinguishability and prove some of its important properties.

DEFINITION 3.2. *A noninteractive proof system (P, V) is* bounded witness indistinguishable *over R if for any large enough input x, any $w_1, w_2 \in w(x)$, and for a randomly chosen reference string σ, the ensembles which differ only in the witness that P is using, but not in x or σ, are computationally indistinguishable. In more detail,*

$\forall D \; \exists N \; \forall n > N \; \forall x \in L_R \bigcap \{0,1\}^n \; \forall w_1, w_2 \in w(x)$,

$$\sum_{\sigma} 2^{-|\sigma|} \cdot |\ \text{Prob}(D(P(x, w_1, \sigma)) = 1), -\ \text{Prob}(D(P(x, w_2, \sigma)) = 1)| \ < \ \nu(n).$$

The probability space is that of P's random coin tosses.

We remark that there are two plausible ways of defining noninteractive witness indistinguishability. In the first alternative, the proofs that use different witnesses need to look similar when both use the same CRS σ. In the second alternative, each proof may use a different σ (that is, for each witness we first average the output of the distinguisher over all choices of σ, and only then compare between the use of different witnesses). The first alternative is stronger, and we adopted it in Definition 3.2. The second alternative is also useful, as it relates more naturally to noninteractive zero knowledge, where σ produced by the simulator M is not required to be identical to σ used by the prover. For the case of efficient provers, the following lemma shows the equivalence of the two alternatives.

LEMMA 3.3. *Noninteractive proof system* (P, V) *with efficient provers is* bounded witness indistinguishable *over* R *if and only if*

$\forall D\ \exists N\ \forall n > N\ \forall x \in L_R \bigcap \{0,1\}^n\ \forall w_1, w_2 \in w(x)$,

$$\left|\sum_{\sigma} 2^{-|\sigma|}(Prob(D(P(x,w_1,\sigma)) = 1) - Prob(D(P(x,w_2,\sigma)) = 1))\right| < \nu(n).$$

Proof. The "only if" direction is obvious. We prove only the "if" direction.

Assume that for some infinite sequence $\mathcal{I}$ of triplets (x, w_1, w_2) of inputs together with their respective witnesses, some nonuniform polynomial time algorithm D can distinguish between P using witness w_1 and P using witness w_2. Formally, for some $k > 0$,

$$\sum_{\sigma} 2^{-|\sigma|} \cdot |Prob(D(P(x,w_1,\sigma)) = 1) - Prob(D(P(x,w_2,\sigma)) = 1)| > \frac{1}{(|x|)^k},$$

where $(x, w_1, w_2) \in \mathcal{I}$, and the probabilities are taken over the random choices of P. We construct a new nonuniform *random* polynomial time distinguisher D', which uses "knowledge" of both w_1 and w_2 and contradicts the condition of the lemma. (D' can be transformed into a *deterministic* nonuniform distinguisher by standard averaging techniques.) The distinguisher D' essentially simulates the behavior of D but with the following modification: on public random strings σ for which $Prob(D(P(x,w_1,\sigma)) = 1) < Prob(D(P(x,w_2,\sigma)) = 1)$, algorithm D' inverts the output of D so as to prevent a cancelation effect between σ with the above property and σ for which $Prob(D(P(x,w_1,\sigma)) = 1) > Prob(D(P(x,w_2,\sigma)) = 1)$.

On input z, a noninteractive proof for x using the public random string σ, D' operates as follows. First, ignoring z, algorithm D' performs the following *bias test* for σ, obtaining a "bias indicator" b. Algorithm D' generates $|x|^{k+1}$ independent strings from each of the distributions $P(x, w_1, \sigma)$ and $P(x, w_2, \sigma)$ (we note that this is possible because P is polynomial time, and D' can simulate P with the relevant auxiliary input). D' feeds these strings to D, obtaining from D two sequences of output bits, each of length $|x|^{k+1}$. If the first such sequence (corresponding to w_1) contains less 1 entries than the second (corresponding to w_2), then b is set to 1. Otherwise b is set to 0. Then D' feeds D with z and flips the output of D if and only if $b = 1$.

It is a simple matter to show that

$$\left|\sum_{\sigma} 2^{-|\sigma|}(Prob(D'(P(x,w_1,\sigma)) = 1) - Prob(D'(P(x,w_2,\sigma)) = 1))\right| > \frac{1}{2(|x|)^k}. \qquad \square$$

Remark. The above proof uses the fact that P is efficient (i.e., polynomial time). The equivalence between definitions might not hold if the prover is nonpolynomial.

Consider for example the zero knowledge proof system of section 2.2 in which the prover inverts one-way functions. It is witness indistinguishable in the sense of Lemma 3.3—this can be proved in a way similar to the proof of Lemma 3.4 below. However, it is not witness indistinguishable in the sense of Definition 3.2: the prover is deterministic, and hence for any particular σ, the use of different witnesses by the prover is distinguishable by examining a single bit location (that may depend on σ) of the prover's output. An averaging argument shows that there is some bit location that (for a polynomial fraction of the possible choices of σ) distinguishes between the two witnesses that the prover may be using.

LEMMA 3.4. *Any bounded NIZK proof system with polynomial time prover is also a bounded noninteractive witness indistinguishable proof system.*

Proof. Assume that the proof system is not witness indistinguishable. By Lemma 3.3, for some constant k and an infinite sequence of inputs there exists a distinguisher D that satisfies

$$\left|\sum_{\sigma} 2^{-|\sigma|}(Prob(D(P(x,w_1,\sigma))=1) - Prob(D(P(x,w_2,\sigma))=1))\right| > \frac{1}{(|x|)^k}.$$

Now the proof system cannot be zero knowledge. For consider any proposed simulator M. No matter what value $Prob(D(M(x))=1)$ takes on, it differs either from $Prob(D(P(x,w_1))=1)$ or from $Prob(D(P(x,w_1))=1)$ by at least $\frac{1}{2(|x|)^k}$. Since both the latter cases are valid distributions of noninteractive proofs for x, we conclude that D is a distinguisher which fails any simulator. ◻

DEFINITION 3.5. *A noninteractive proof system is* general witness indistinguishable *over R if for any positive constant c, for any $m \le n^c$, we have that the two ensembles $\{(P(x_1,w_1^1,\sigma), P(x_2,w_2^1,\sigma),\ldots,P(x_m,w_m^1,\sigma))\}$ and $\{(P(x_1,w_1^2,\sigma), P(x_2,w_2^2,\sigma),$ $\ldots,P(x_m,w_m^2,\sigma))\}$ are computationally indistinguishable (in the sense of Definition* 3.2*, where σ is identical in the two ensembles). In more detail,*

$$\forall D\ \forall c\ \exists N\ \forall n > N\ \forall m < n^c$$

whenever $x_i \in L_R \bigcap \{0,1\}^n$ and $w_i^1, w_i^2 \in w(x_i)$ for all $1 \le i \le m$, then

$$\sum_{\sigma} 2^{-|\sigma|} \cdot |Prob(D(P(x_1,w_1^1,\sigma),\ldots,P(x_m,w_m^1,\sigma))=1)$$

$$-Prob(D(P(x_1,w_1^2,\sigma),\ldots,P(x_m,w_m^2,\sigma))=1)| \ < \ \nu(n).$$

The probability space is that of P's random coin tosses.

LEMMA 3.6. *Any bounded noninteractive witness indistinguishable proof system with efficient provers is also general witness indistinguishable.*

Proof. Assume that for some constant c and infinitely many n the following holds: there exists a distinguisher D of size at most n^c, a positive integer m, where $m \le n^c$, a sequence $\mathcal{X} = (x_1, x_2, \ldots, x_m)$ of inputs (each of size n), and two sequences, $\mathcal{W}^1 = (w_1^1,\ldots,w_m^1)$ and $\mathcal{W}^2 = (w_1^2,\ldots w_m^2)$, of witnesses for the respective $x_i \in \mathcal{X}$, such that

$$\sum_{\sigma} 2^{-|\sigma|} \cdot |Prob(D(P(x_1,w_1^1,\sigma),\ldots,P(x_m,w_m^1,\sigma))=1)$$

$$-Prob(D(P(x_1,w_1^2,\sigma),\ldots,P(x_m,w_m^2,\sigma))=1)| \ > \ n^{-c},$$

where probabilities are taken over the random coin tosses of P. Then by the "hybrid" argument of [GM] (also known as "probability walk" argument), there must be a "polynomial jump" somewhere in the execution: there exists k, where $1 \leq k \leq m$, such that

$$\sum_{\sigma} 2^{-|\sigma|} \cdot |Prob(D(P(x_1, w_1^1, \sigma), \ldots, P(x_k, w_k^1, \sigma), \ldots, P(x_m, w_m^2, \sigma)) = 1)$$

$$-Prob(D(P(x_1, w_1^1, \sigma), \ldots, P(x_k, w_k^2, \sigma), \ldots, P(x_m, w_m^2, \sigma)) = 1)| \;>\; \frac{1}{mn^c}.$$

We now use the nonuniformity of the distinguishers to derive a contradiction. Since the proof system has efficient provers, the whole set of proofs $(P(x_1, w_1^1, \sigma), \ldots, P(x_{k-1}, w_{k-1}^1, \sigma), P(x_{k+1}, w_{k+1}^2, \sigma), \ldots, P(x_m, w_m^2, \sigma))$ can be simulated by a modified D', who has as auxiliary input $((x_1, w_1^1), \ldots, (x_{k_1}, w_{k-1}^1), (x_{k+1}, w_{k+1}^2), \ldots, (x_m, w_m^2))$. This random nonuniform polynomial time D' can now distinguish between proofs for x_k in which the prover uses w_k^1 and proofs in which the prover uses w_k^2. This contradicts our assumption that the original protocol was witness indistinguishable. □

3.3. The transformation. We assume the existence of pseudorandom bit generators (see [BM], [Yao]), which extend n-bit random seeds to $2n$-bit pseudorandom strings, computationally indistinguishable from strings of truly random $2n$ bits. The existence of pseudorandom generators follows from the assumption that one-way functions exist [ILL], [Ha].

Let (P, V) be any bounded NIZK proof system with polynomial time prover for the NP-complete language L_R. We construct $(\bar{\text{P}}, \bar{\text{V}})$, a general NIZK proof system for L_R, under the sole assumption that one-way functions exist.

Let $g : \{0,1\}^n \longrightarrow \{0,1\}^{2n}$ be a pseudorandom bit generator. We introduce two new NP languages. L_{R_g} is the NP language corresponding to the relation $R_g(y, s)$ iff $g(s) = y$. $L_{R_\#}$ is the NP language corresponding to the relation $R_\#(x\#y, w)$ iff *either* $R(x, w)$ *or* $R_g(y, w)$, where # is used as a special delimiting character in the alphabet of the inputs to $L_{R_\#}$.

We now describe the algorithm of $\bar{\text{P}}$ on input $(x, w) \in R$ and reference string σ.

1. Divide the CRS σ into two segments: the first $2n$ bits, denoted by y, and called the *reference statement*; the rest of the CRS is denoted by σ'.
2. Construct the instance $x\#y \in L_{R_\#}$. Observe that w, the witness that $\bar{\text{P}}$ has for $x \in L_R$, is a witness for $x\#y \in L_{R_\#}$.
3. Reduce $x\#y$ to an instance X of the NP-complete language L_R, using a publicly known reduction with efficient transformation of witnesses. (That is, any witness for the original instance can be efficiently transformed into a witness for the target instance, and vice versa. Known reductions to NP-complete languages have this property.) Reduce w, the witness for $x\#y \in L_{R_\#}$, to W, a witness for $X \in L_R$.
4. Send x, X, and $P(X, W, \sigma')$. (The last term, $P(X, W, \sigma')$, is a random variable that denotes the NIZK proof that the prover P from the system (P, V) would produce on input X, witness W, and reference string σ', depending on P's private random string.)

The verifier $\bar{\text{V}}$ accepts if the publicly known reduction (from $L_{R_\#}$ to L_R) gives X when applied to $x\#y$, and if furthermore V would have accepted $P(X, W, \sigma')$.

THEOREM 3.7. *Under the assumptions that (P, V) is a bounded NIZK proof system with efficient provers for L_R, that L_R is NP-complete, and that g is a pseudo-*

random generator, the above transformed scheme is a general NIZK proof system for L_R.

Proof. We first give an intuitive introduction to the full proof. The completeness, soundness, and zero knowledge properties of ($\bar{\mathrm{P}}$, $\bar{\mathrm{V}}$) are based on the corresponding properties of (P, V). Efficiency is preserved in the completeness property since from a witness to x, prover $\bar{\mathrm{P}}$ can derive a witness to X, and thus execute the bounded NIZK proof system (P, V). The soundness property follows from the fact that y is chosen as a truly random (rather than pseudorandom) string. Thus, for almost all possible choices of y, it is not in the range of g, and consequently $X \in L_R$ if and only if $x \in L_R$. The zero knowledge property requires more subtle analysis. The simulation of ($\bar{\mathrm{P}}$, $\bar{\mathrm{V}}$) is done by replacing the reference statement y by a pseudorandom string y'. This y' is generated by selecting at random an n bit seed s and computing $y' = g(s)$. Since g is a pseudorandom bit generator, this replacement is indistinguishable to polynomial time observers. Now any statement x with witness w is transformed into a statement X which also has s, the seed of y', as its witness. The simulator uses s instead of w in order to prove X. The concept of witness indistinguishability can now be used to show that this change of witnesses in the proof of X is indistinguishable to polynomial time observers, even if it is done polynomially many times. We now give a more detailed proof.

Completeness (while preserving efficiency). The reduction of $L_{R_\#}$ to L_R allows efficient transformation of witnesses. Thus $\bar{\mathrm{P}}$, who knows a witness for $x \in L_R$, and hence for $x\#y \in L_{R_\#}$, can also compute in polynomial time a witness for X and use it in order to perform the protocol. The reduction is also publicly known, and so $\bar{\mathrm{V}}$ can check that it was followed correctly. The completeness property then follows from the completeness property of (P, V).

Soundness. From the soundness property of (P, V) it follows that either $x \in L_R$ or y is in the range of the generator g (i.e., there is an s such that $y = g(s)$). But simple counting shows that the probability that the random string y of length $2n$ is in the range of g (i.e., has a seed of length n) is at most 2^{-n}, and thus with overwhelming probability indeed $x \in L_R$.

The completeness and soundness property trivially continue to hold even if polynomially many statements are proved.

Zero knowledge. For any large enough value of n, consider any sequence (x_1, w_1), (x_2, w_2), $\ldots$, (x_m, w_m), of inputs together with their respective witnesses, where m is polynomial in n. Assume that $\bar{\mathrm{P}}$ proves for these inputs membership in L_R (by using private coin tosses, the associated witnesses, and the CRS σ). We construct a simulator M which creates an ensemble indistinguishable from the ensemble that $\bar{\mathrm{P}}$ produces. M receives as input only the sequence of instances $\{x_i\}$, without their respective witnesses.

M randomly selects an n bit seed s and computes the $2n$-bit string $y' = g(s)$, to be used as the reference statement (instead of a random y used in reality). M generates a truly random reference string σ'. For each $1 \le j \le m$, M reduces $x_j\#y'$ to an instance X_j of L_R and derives from s a witness w' for this instance. Then M uses the proof system (P, V) and the reference string σ' to simulate a proof that $X \in L_R$, by using its knowledge of the seed s (rather than a witnesses it does not have to the statements x_j).

In order to prove that M's simulation is indistinguishable from $\bar{\mathrm{P}}$'s proofs, we construct a hybrid $\bar{\mathrm{M}}$, which constructs y' pseudorandomly as M does, but simulates the proofs using w_1, w_2, $\ldots$ as $\bar{\mathrm{P}}$ does.

LEMMA 3.8. *For any positive constant c, for any $m \leq n^c$, the two ensembles $\{(\sigma, \bar{P}(x_1, w_1, \sigma), \ldots, \bar{P}(x_m, w_m, \sigma))\}$ and $\{\bar{M}((x_1, w_1), (x_2, w_2), \ldots, (x_m, w_m))\}$ are computationally indistinguishable. In any sequence of instances that indexes the ensembles, all x_i are of the same length (denoted by n), and for all $1 \leq i \leq m$, $(x_i, w_i) \in R$.*

Proof. Assume otherwise, and let D be a distinguisher that distinguishes between $\bar{\text{M}}$'s output and $\bar{\text{P}}$'s output. We construct a distinguisher D' that distinguishes between truly random strings, and outputs of the generator g, contradicting its pseudorandomness. For infinitely many values of n, nonuniform algorithm D' has as auxiliary input the corresponding sequence (x_1, w_1), (x_2, w_2), $\ldots$, (x_m, w_m) for which D distinguishes between the two ensembles. In order to test whether a string y of length $2n$ was generated from the distribution of outputs of the generator g, D' generates a random reference string σ' and simulates $\bar{\text{P}}$'s action on the sequence (x_1, w_1), (x_2, w_2), $\ldots$, (x_m, w_m), with respect to the reference string composed of y and σ'. Now the verdict of D of whether the output was produced by $\bar{\text{P}}$ (which uses truly random y) or by $\bar{\text{M}}$ (which uses pseudorandom y) forms a statistical test as to whether y was truly random or generated by g. □

The proof of the following lemma is the heart of our argument that the transformed scheme is general zero knowledge.

LEMMA 3.9. *For any positive constant c, for any $m \leq n^c$, the two ensembles $\{\bar{M}((x_1, w_1), (x_2, w_2), \ldots, (x_m, w_m))\}$ and $\{M(x_1, x_2, \ldots, x_m)\}$ are computationally indistinguishable. In any sequence of instances that indexes the ensembles, all x_i are of the same length (denoted by n), and for all $1 \leq i \leq m$, $(x_i, w_i) \in R$.*

Proof. Consider what the simulators $\bar{\text{M}}$ and M actually do. They are giving NIZK proofs for a sequence of statements $X_1, \ldots, X_m$, derived from the sequence $x_1, \ldots, x_m$, by taking into account a reference statement y'. In more detail, they first generate a reference statement y', and thereafter each of them uses the truly random reference string σ' to execute NIZK protocols for the sequence $X_1, \ldots, X_m$, in the same way as a true prover in (P, V) would execute such protocols. The only difference between $\bar{\text{M}}$ and M is in the sequence of witnesses that they are using, where $\bar{\text{M}}$ uses the sequence $w_1, \ldots, w_m$, and M repeatedly uses s as a witness for each of the X_i (recall that s is the seed that was used in order to generate y').

Protocol (P, V) is zero knowledge. By Lemma 3.4 it is witness indistinguishable. By Lemma 3.6, even polynomially many executions of (P, V) on the same reference string σ' are witness indistinguishable. Hence the ensembles that $\bar{\text{M}}$ and M create are indistinguishable. □

From the two lemmas above it follows that the outputs of M and $\bar{\text{P}}$ are computationally indistinguishable, which completes the proof that the protocol is zero knowledge. □

Remarks.

1. In giving multiple NIZK proofs, the prover reuses public randomness (the CRS) over and over again. However, the prover must use "fresh" private randomness (internal random coin tosses) in each execution of the protocol. Otherwise, zero knowledge might not be preserved. In particular, when applying our transformation to the efficient bounded NIZK described in section 2.4, the truthful prover is expected to use a different trapdoor permutation for each input statement being proved.
2. The complexity of the pseudorandom bit generator g has major impact on the overall complexity of our transformation. The choice of g (and, in par-

ticular, the time required for a nondeterministic Turing machine to accept the language L_{R_g}) influences the size of X, the statement that the truthful prover eventually proves (which may be much larger than the size of x, the original statement that the prover wants to prove). There is a spectrum of constructions of pseudorandom bit generators, where the complexity of the construction typically depends on the strength of the underlying computational complexity assumptions (e.g., compare [BM] with [ILL, Ha]). In applying our transformation, one should first determine the computational complexity assumptions that were made in the construction of the particular bounded NIZK proof system, and based on them, select the simplest pseudorandom bit generator.

3. The term *bounded* NIZK proof systems indicates that the length of the CRS bounds the size of the (single) NP statement that can be proven. We have seen that using *general* NIZK proof systems, polynomially many NP statements, each of bounded length, can be proved. [BDMP] show that under the assumption that one-way functions (and hence encryption schemes) exist, general NIZK proof systems can be used in order to prove statements that are longer than the bound implied by the length of the CRS. For completeness, we sketch how this is done.
   On input of a satisfiable 3-SAT formula Φ (any other NP-statement can be reduced to 3-SAT, if necessary), the prover encrypts separately the satisfying value of each variable and for each clause $C \in \Phi$ constructs the string composed of the concatenation of the encrypted values of the three variables in C. Observe that the length of each such string does not depend on the length of Φ. Each string is treated as an NP-statement: "there exists a decryption of the encrypted values that would show that clause C is satisfiable." The prover gives a NIZK for each such statement separately, and the verifier verifies that each of the NIZK proofs is acceptable.

4. Security against adaptive attacks.

4.1. Definitions. NIZK proof systems are useful design primitives in the construction of cryptographic schemes, such as signature schemes [BG] and encryption schemes [NY]. It is often required that the cryptographic scheme will be robust against attacks of adaptive nature, which are the strongest types of attack. For example, a standard security requirement of signature schemes is that even after requesting signatures of polynomially many messages of his choice, the adversary is not able to forge a signature to any new message. In order to treat adaptive attacks we extend the security requirements of NIZK proof systems.

In a typical adaptive scenario, a polynomial time adversary A repeatedly selects statements and observes their noninteractive proofs. His goal is to come up with a statement x on which one of the three basic properties of NIZK proof systems is violated: either x is true but P cannot produce a noninteractive proof for it (violating the *completeness* condition) or x is false but there exists a noninteractive "proof" that convinces V to accept x (violating the *soundness* condition), or x is true and the adversary can extract useful information from the noninteractive proof that P gives (violating the *zero knowledge* property).

Definition 1.2 is strong enough to serve as the definition of *completeness* and *soundness* for the adaptive scenario. However, Definition 3.1 of *zero knowledge* needs to be modified. We define the concept of *adaptive zero knowledge* in a way similar to Bellare and Goldwasser [BG] (see also [GGM]'s test for pseudorandom functions).

We call this test *adaptive indistinguishability*, or the AI test (partially because of its origins as Turing's test for artificial intelligence). In our AI test, an adversary A is confronted with a blackbox B. His goal is to determine whether B contains a real prover P or whether it contains a simulator M. A first requests the random reference string σ from B. If P is inside the box, it replies with a truly random string. If M is inside the box, it replies with a string of its choice. Now, possibly based on σ, A generates a pair $(x, w) \in R$ and sends it to B. If P is inside the box, it produces a noninteractive proof $P(x, w, \sigma)$. If M is inside the box, w is "magically" filtered away, and M must simulate a noninteractive proof for x. This procedure of adaptively choosing theorems and receiving noninteractive proofs for them is repeated polynomially many times until A is ready to pass a decision: $'0'$ or $'1'$. (M, P) are said to pass the AI test if the probabilities that A outputs $'1'$ when M is inside the box and when P is inside the box are equal up to negligible additive terms.

DEFINITION 4.1. *A noninteractive adaptive proof system* (P, V) *is* adaptive zero knowledge *if there exists a random polynomial time simulator* M *such that* (M, P) *pass the AI test for any nonuniform polynomial time adversary* A.

We remark that in [BG]'s definition, A is not required to supply witnesses for $x \in L_R$ to the blackbox. This leaves open the question of how the real prover P (which is assumed to be efficient) comes up with a witness w for $x \in L_R$, to be used in producing a NIZK proof for this fact. One possibility is that some computationally unbounded agent produces this witness for P. Another is that A has to produce the witness. In our definition we choose the latter possibility, with the intention of using it only in applications where all parties are (nonuniform) polynomial time. We do not know if the theorems to follow (and, in particular, Theorem 4.4) hold also with respect to the stronger definition of zero knowledge, in which a computationally unbounded agent produces the witnesses.

PROPOSITION 4.2. *Any noninteractive proof system which is adaptive zero knowledge (Definition* 4.1*) is also general zero knowledge (Definition* 3.1*).*

Proof. The proof follows directly from the nonuniformity of the adversary in Definition 4.1. If there is a sequence of inputs with respective witnesses that serves to defeat the general zero knowledge property (according to Definition 3.1), then the adversary A (of Definition 4.1) can hold this same sequence as auxiliary input and defeat the adaptive zero knowledge property. ◻

A somewhat weaker condition than adaptive zero knowledge is *single statement adaptive zero knowledge*. For such proof systems, the adversary of the AI test is allowed to request only one noninteractive proof from B before passing his judgment as to what is inside the box.

PROPOSITION 4.3. *Any noninteractive proof system which is single statement adaptive zero knowledge is also bounded zero knowledge (Definition* 1.3*).*

Proof. The proof follows directly from the nonuniformity of the adversary. ◻

Remark. The converse of the above proposition is probably not true. For example, consider the NIZK proposed in [BDMP] for the language NQR. [BDMP] prove that the noninteractive proof system for this language is zero knowledge by producing a simulator M that constructs a CRS only after it sees the input x. In contrast, adaptive zero knowledge postulates that σ is generated first, and x is chosen only later. Despite the fact that [BDMP]'s NIZK proof system for the language NQR is bounded zero knowledge, it is not known to be single statement adaptive zero knowledge.

4.2. Robustness of our protocols. All theorems and lemmas that deal with the adaptive zero knowledge scenario are straightforward modifications of their counterparts that dealt with the nonadaptive scenario. For this reason, we only state our theorems, and give some "hints" to help the reader in modifying the proofs in previous sections so that they also apply to the adaptive case.

THEOREM 4.4. *The NIZK of section* 2 *is single statement adaptive zero knowledge.*

Proof. There are two parts to a proof that a certain protocol is zero knowledge. One part is to construct the simulator M. The other part is to prove that its output is indistinguishable from the output of the real prover.

We first address the problem of constructing the "adaptive" simulation. Consider the simulator M described in section 2. This simulator generates a reference string σ' independently of the common input x. This σ' can be used to simulate a noninteractive proof for any input x. Consequently, the same simulator can be used even if the input statement is chosen adaptively after the CRS is chosen.

The proof of indistinguishability follows the same arguments as those which are used in the proof of subsection 2.3.3. Recall that it was shown that if the proof system is not zero knowledge on input $(x, w) \in R$, then one could construct an efficient algorithm (denoted by C) that predicts the hard bits of the one-way permutation (or trapdoor permutation, if the prover is required to be efficient). This algorithm had (x, w) as auxiliary input and constructed a reference string σ in the course of its operation. For the case of adaptive zero knowledge, the end result would again be an efficient algorithm C that predicts the hard bits of the one-way permutation. However, this algorithm would not explicitly receive any $(x, w) \in R$ as auxiliary input. The reason for this is that in the adaptive scenario, the adversary A need not have a prespecified (x, w) that it uses in foiling the zero knowledge property. Instead, A may generate (x, w) only after observing σ. Likewise, we must allow C to generate (x, w) only after constructing σ, in a way similar to A. Hence, instead of supplying C with explicit (x, w) as auxiliary input, we supply it with the auxiliary input of A, which C can later use to generate (x, w) that depend on σ. $\square$

In the rest of this section we consider only NIZK proofs with efficient provers.

THEOREM 4.5. *The transformation of section* 3.3 *transforms efficient noninteractive single statement adaptive zero knowledge proof systems into efficient (general) noninteractive adaptive zero knowledge proof systems.*

Proof. Theorem 4.5 is proved in the same way as Theorem 3.7, which is its nonadaptive counterpart. We only sketch the modifications that are necessary.

The proof of the *completeness* and *soundness* conditions is straightforward. The main emphasis is on the proof of the *adaptive zero knowledge* property. Recall that in order to prove Theorem 3.7, we used the concept of *witness indistinguishability.* In order to prove Theorem 4.5, we define a corresponding notion of *adaptive witness indistinguishability.* Once this concept is defined, and its main properties are established (see Lemmas 4.7 and 4.8 below), the proof of Theorem 4.5 can be carried out in a way similar to the proof of Theorem 3.7.

In the AI test for witness indistinguishability a nonuniform polynomial time adversary A is confronted with a blackbox B that may contain one of two possible provers, denoted by P_1 and P_2. The provers differ in the witnesses that they select to use in producing NIZK proofs, and the goal of A is to determine whether B contains P_1 or P_2. A first requests the random reference string σ from B. Now, possibly based on σ, A generates a triplet (x, w^1, w^2), where $(x, w^1) \in L_R$ and $(x, w^2) \in L_R$ and sends the triplet to B. If P_1 is inside B, it uses only the first of the given witnesses for x to generate the noninteractive proof $P(x, w^1, \sigma)$. If P_2 is inside B, it uses only the second of the given witnesses to generate $P(x, w^2, \sigma)$. This procedure of adaptively choosing theorems and receiving noninteractive proofs for them is repeated polynomially many times until A is ready to pass a decision: $'0'$ or $'1'$. (P_1, P_2) are said to pass the AI test for witness indistinguishability if the probabilities that A outputs $'1'$ when P_1 is inside the box and when P_2 is inside the box are equal up to negligible additive terms.

Definition 4.6. *A noninteractive adaptive proof system* (P, V) *is* adaptive witness indistinguishable *if* (P_1, P_2) *pass the AI test for witness indistinguishability.*

In *single statement adaptive witness indistinguishability*, the adversary A is allowed to request only one noninteractive proof from B before passing his judgment of whether P_1 or P_2 is inside the box.

Lemma 4.7. *Any noninteractive proof system which is single statement adaptive zero knowledge is also single statement adaptive witness indistinguishable.*

The proof of Lemma 4.7 is similar to the proof Lemma 3.4 and is omitted.

The following lemma shows that adaptive witness indistinguishability is preserved under repeated applications of the noninteractive proof system with the same random reference string.

Lemma 4.8. *Any noninteractive proof system which is single statement adaptive witness indistinguishable is also adaptive witness indistinguishable.*

Proof. Assume that there exists an adversary A which adaptively generates triplets (x_i, w_i^1, w_i^2) (for $i \geq 1$) and can distinguish between P_1 and P_2. By the "hybrid" argument of [GM], there must be a "polynomial jump" somewhere in the execution: there exists k, such that if for $i < k$ the adversary uses the first of the two generated witnesses of each instance to produce $P(x_i, w_i^1, \sigma)$ by itself, then generates (x_k, w_k^1, w_k^2) and gives it to the blackbox, and finally (for $i > k$) uses the second of the two generated witnesses of each instance to produce $P(x_i, w_i^2, \sigma)$ by itself, then A can distinguish between the case that P_1 is inside the box and the case that P_2 is inside the box. This contradicts our assumption that the original protocol was single statement adaptive witness indistinguishable. □

The rest of the proof of Theorem 4.5 can be carried out in a way similar to the proof of Theorem 3.7. The details are omitted. □

5. Conclusions. We show how one can construct general NIZK proof systems under general computational complexity assumptions. Theoretically, NIZK proof systems have numerous cryptographic applications ([BG], [NY], and we are confident that more will follow). However, to be useful in practice, the efficiency of NIZK proof systems must be greatly improved. One parameter that deserves special attention is the length of the CRS. To prove Hamiltonicity of n-node graphs, our NIZK proof system requires $|\sigma| = \Omega(n^{11/2})$. Recently, Kilian [K94] presented considerably more efficient NIZK proof systems for circuit satisfiability, and this was further simplified and improved by Kilian and Petrank [KP], to a point where the complexity of NIZK proof systems for NP statements almost matches that of the most efficient known *interactive* zero knowledge proof systems.

The following question remains open: what are the minimal computational complexity assumptions that support bounded NIZK proof systems?

Recall that once bounded NIZK proof systems with efficient provers are constructed, the transformation to general NIZK proof systems requires only the assumption that one-way functions exist and are relatively efficient. Since the transformation to general NIZK proof systems requires only bounded noninteractive witness indistinguishable proof systems as a starting point, the above question can be reformulated with NIZK replaced by noninteractive witness indistinguishability.

Acknowledgments. We thank Moni Naor for helpful discussions, Oded Goldreich for his useful remarks on an early version of this manuscript, and the referees of this paper for greatly improving its readability.

REFERENCES

[BG] M. Bellare and S. Goldwasser, *New paradigms for digital signatures and message authentication based on non-interactive zero knowledge proofs*, in Advances in Cryptology-CRYPTO 89, Lecture Notes in Computer Science 435, Springer-Verlag, New York, 1989, pp. 194–211.

[BeMi] M. Bellare and S. Micali, *Non-interactive oblivious transfer and applications*, in Advances in Cryptology-CRYPTO 89, Lecture Notes in Computer Science 435, Springer-Verlag, New York, pp. 547–557.

[BY] M. Bellare and M. Yung, *Certifying permutations: Noninteractive zero-knowledge based on any trapdoor permutations*, J. Cryptology, 9 (1996), pp. 149–166.

[Blum] M. Blum, *How to prove a theorem so no one else can claim it*, in Proc. of the International Congress of Mathematicians, Berkeley, CA, 1986, pp. 1444–1451.

[BDMP] M. Blum, A. De Santis, S. Micali, and G. Persiano, *Noninteractive zero-knowledge*, SIAM J. Comput., 20 (1991), pp. 1084–1118.

[BFM] M. Blum, P. Feldman, and S. Micali, *Noninteractive zero-knowledge and its applications*, in Proc. of 20th ACM Symposium on Theory of Computing, ACM Press, NY, 1988, pp. 103–112.

[BM] M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudo-random bits*, SIAM J. Comput., 13 (1984), pp. 850–864.

[DMP87] A. De Santis, S. Micali, and G. Persiano, *Non-interactive zero-knowledge proof systems*, in Advances in Cryptology-CRYPTO 87, Lecture Notes in Computer Science 293, Springer-Verlag, New York, 1987, pp. 52–72.

[DMP88] A. De Santis, S. Micali, and G. Persiano, *Non-interactive zero-knowledge with preprocessing*, in Advances in Cryptology-CRYPTO 88, Lecture Notes in Computer Science 403, Springer-Verlag, New York, 1988, pp. 269–283.

[DP] A. De Santis and G. Persiano, *Zero-knowledge proofs of knowledge without interaction*, in Proc. of 33rd IEEE Annual Symposium on Foundations of Computer Scienc, 1992, pp. 427–436.

[DY] A. De Santis and M. Yung, *Cryptographic applications of the non-interactive metaproof and many-prover systems*, in Advances in Cryptology-CRYPTO 90, Lecture Notes in Computer Science, 537, Springer-Verlag, New York, 1990, pp. 366–377.

[FS] U. Feige and A. Shamir, *Witness indistinguishable and witness hiding protocols*, in Proc. of 22nd ACM Symposium on Theory of Computing, ACM Press, NY, 1990, pp. 416–426.

[G] O. Goldreich, *A Uniform-Complexity Treatment of Encryption and Zero-Knowledge*, TR-568, Computer Science Dept., Technion, Haifa, Israel, 1989.

[GGM] O. Goldreich, S. Goldwasser, and S. Micali, *How to construct random functions*, J. Assoc. Comput. Mach., 33 (1986), pp. 792–807.

[GILVZ] O. Goldreich, R. Impagliazzo, L. Levin, R. Venkatesan, and D. Zuckerman, *Security preserving amplification of hardness*, in Proc. of 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 318–326.

[GL] O. Goldreich and L. Levin, *A hard-core predicate for all oneway functions*, in Proc. of 21st ACM Symposium on Theory of Computing, ACM Press, NY, 1989, pp. 25–32.

[GMW] O. Goldreich, S. Micali, and A. Wigderson, *Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems*, J. Assoc. Comput. Mach., 38 (1991), pp. 691–729.

[GM] S. Goldwasser and S. Micali, *Probabilistic Encryption*, J. Comput. System Sci., 28 (1984), pp. 270–299.

[GMR] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof systems*, SIAM J. Comput., 18 (1989), pp. 186–208.

[Ha] J. Hastad, *Pseudo-random generators under uniform sssumptions*, in Proc. of 22nd ACM Symposium on Theory of Computing, ACM Press, NY, 1990, pp. 395–404.

[ILL] R. Impagliazzo, L. Levin, and M. Luby, *Pseudorandom generation from oneway functions*, in Proc. of 21st ACM Symposium on Theory of Computing, ACM Press, NY, 1989, pp. 12–24.

[K94] J. Kilian, *On the complexity of bounded-interaction and moninteractive zero-knowledge proofs*, in Proc. 35th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 466–477.

[KMO] J. Kilian, S. Micali, and R. Ostrovsky, *Minimum resource zero-knowledge proofs*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 474–479.

[KP] J. Kilian and E. Petrank, *An Efficient Non-Interactive Zero-Knowledge Proof System for NP with General Assumptions*, J. Cryptology, 11 (1998), pp. 1–27.

[LS] D. Lapidot and A. Shamir, *Publicly verifiable non-interactive zero-knowledge proofs*, in Advances in Cryptology-CRYPTO 90, Lecture Notes in Computer Science, 537, Springer-Verlag, New York, 1990, pp. 353–365.

[Naor] M. Naor, *Bit commitment using pseudorandomness*, J. Cryptology, 4 (1991), pp. 151–158.

[NY] M. Naor and M. Yung, *Public-key cryptosystems provably secure against chosen ciphertext attacks*, in Proc. of 22nd ACM Symposium on Theory of Computing, ACM Press, NY, 1990, pp. 427–437.

[RS] C. Rackoff and D. Simon, *Non-interactive zero-knowledge proofs of knowledge and chosen ciphertext attacks*, in Advances in Cryptology-CRYPTO 91, Lecture Notes in Computer Science, 576, Springer-Verlag, New York, 1991, pp. 433–444.

[Yao] A. C. Yao, *Theory and applications of trapdoor functions*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1982, pp. 80–91.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 29–64

TIGHT ANALYSES OF TWO LOCAL LOAD BALANCING ALGORITHMS [*]

BHASKAR GHOSH[†], F. T. LEIGHTON[‡], BRUCE M. MAGGS[§], S. MUTHUKRISHNAN[¶], C. GREG PLAXTON[‖][††], R. RAJARAMAN[‖][††], ANDRÉA W. RICHA[§], ROBERT E. TARJAN[#], AND DAVID ZUCKERMAN[‖][‡‡]

Abstract. This paper presents an analysis of the following load balancing algorithm. At each step, each node in a network examines the number of tokens at each of its neighbors and sends a token to each neighbor with at least $2d+1$ fewer tokens, where d is the maximum degree of any node in the network. We show that within $O(\Delta/\alpha)$ steps, the algorithm reduces the maximum difference in tokens between any two nodes to at most $O((d^2 \log n)/\alpha)$, where Δ is the global imbalance in tokens (i.e., the maximum difference between the number of tokens at any node initially and the average number of tokens), n is the number of nodes in the network, and α is the edge expansion of the network. The time bound is tight in the sense that for any graph with edge expansion α, and for any value Δ, there exists an initial distribution of tokens with imbalance Δ for which the time to reduce the imbalance to even $\Delta/2$ is at least $\Omega(\Delta/\alpha)$. The bound on the final imbalance is tight in the sense that there exists a class of networks that can be locally balanced everywhere (i.e., the maximum difference in tokens between any two neighbors is at most $2d$), while the global imbalance remains $\Omega((d^2 \log n)/\alpha)$. Furthermore, we show that upon reaching a state with a global imbalance of $O((d^2 \log n)/\alpha)$, the time for this algorithm to locally balance the network can be as large as $\Omega(n^{1/2})$. We extend our analysis to a variant of this algorithm for dynamic and asynchronous networks. We also present tight bounds for a randomized algorithm in which each node sends at most one token in each step.

Key words. load balancing, distributed network algorithms

AMS subject classification. 68Q22

PII. S0097539795292208

---

[*]Received by the editors September 22, 1995; accepted for publication (in revised form) November 7, 1997; published electronically September 14, 1999.
http://www.siam.org/journals/sicomp/29-1/29220.html

[†]Informix Software, Menlo Park, CA 94025 (ghosh@informix.com). The work of this author was done while the author was at the Department of Computer Science, Yale University, New Haven, CT 06520, and was supported by ONR grant 4-91-1576 and a Yale-IBM joint study.

[‡]Department of Mathematics and Laboratory for Computer Science, MIT, Cambridge, MA 02139 (ftl@math.mit.edu). The work of this author was supported by ARPA contracts N00014-91-J-1698 and N00014-92-J-1799.

[§]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (bmm@cs.cmu.edu, aricha@cs.cmu.edu). The work of these authors was supported in part by NSF National Young Investigator Award CCR-9457766 and ARPA contract F33615-93-1-1330.

[¶]Bell Labs, 700 Mountain Avenue, Murray Hill, NJ 07974 (muthu@research.bell-labs.com). This work was done while the author was at DIMACS, Rutgers University, Piscataway, NJ 08855. The work of this author was supported by DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, a National Science Foundation Science and Technology Center, under NSF contract STC-8809648.

[‖]Department of Computer Science, University of Texas, Austin, TX 78712 (plaxton@cs.utexas.edu, diz@cs.utexas.edu). Current address for Rajaraman: College of Computer Science, Northeastern University, Boston, MA 02115 (rraj@ccs.neu.edu). The work of the sixth author was done at the Department of Computer Science, University of Texas, Austin, TX.

[#]Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544 (ret@cs.princeton.edu), and NEC Research Institute. Research at Princeton University was supported in part by National Science Foundation grant CCR-8920505 and Office of Naval Research contract N0014-91-J-1463.

[††]The work of these authors was supported by Texas Advanced Research Program grant ARP-93-003658-461.

[‡‡]The work of this author was supported in part by NSF National Young Investigator Award CCR-9457799.

1. Introduction. A natural way to balance the workload in a distributed system is to have each workstation periodically poll the other stations to which it is connected and send some of its work to stations with less work pending. This paper analyzes the effectiveness of this local load balancing strategy in the simplified scenario in which each workstation has a collection of independent unit-size jobs (called *tokens*) that can be executed on any other workstation. We model a distributed system as a graph, where nodes correspond to workstations and edges correspond to connections between stations, and we assume that in one unit of time, at most one token can be transmitted across an edge of the graph in each direction. Our analysis addresses only the static load balancing aspect of this problem: we assume that each processor has an initial collection of tokens and that no tokens are created or destroyed while the tokens are being balanced.

We analyze the algorithms in this paper in terms of the initial *imbalance* of tokens, i.e., the maximum difference between the number of tokens at any node and the average number of tokens, which we denote Δ; the number of nodes in the graph, which we denote n; the maximum degree of the graph, d; and the node and edge expansion of the graph. We define the *node expansion* μ of a graph G to be the largest value such that every set S of $n/2$ or fewer nodes in G has at least $\mu|S|$ neighbors outside of S. We define the *edge expansion* α of a graph G to be the largest value such that for every set S of $n/2$ or fewer nodes in G, there are at least $\alpha|S|$ edges in G with one endpoint in S and the other not in S.

The performance of an algorithm is characterized by the time that it takes to balance the tokens and by the final balance that it achieves. We say that an algorithm *globally balances* (or just *balances*) to within t tokens if the maximum difference in the number of tokens between any two nodes in the graph is at most t. We say that an algorithm *locally balances* to within t tokens if the maximum difference in the number of tokens between any two neighboring nodes in the graph is at most t.

We analyze two different types of algorithms in this paper: single-port and multiport. In the *single-port* model, a node may send and/or receive at most one token in one unit of time. In the *multiport* model, a node may send and/or receive a token across all of its edges (there may be as many as d) in a single unit of time. Not surprisingly, the load balancing algorithms run faster in the multiport model. In practice, however, single-port nodes may be preferred to multiport nodes because they are easier and less costly to build.

1.1. Our results. This paper analyzes the simplest and most natural local algorithms in both the single-port and multiport models.

In the single-port algorithm, a matching is randomly chosen at each step. First, each (undirected) edge in the network is independently selected to be a *candidate* with probability $1/(4d)$. Then each candidate edge (u, v) for which there is another candidate edge (u, x) or (y, v) is removed from the set of candidates. The remaining candidates form a matching M in the graph. For each edge (u, v) in M, u sends a token to v if at the beginning of the step node u contains at least two more tokens than v. This algorithm was first analyzed in [14].

We analyze the performance of the single-port algorithm in terms of both the edge expansion and the node expansion of the graph. In terms of edge expansion, we show that the single-port algorithm balances to within $O((d \log n)/\alpha)$ tokens in $O(d\Delta/\alpha)$ steps with high probability. In terms of node expansion, the final imbalance is $O((\log n)/\mu)$ and the time is $O(d\Delta/\mu)$ with high probability. (To compare these bounds, note that $\mu \leq \alpha \leq d\mu$.) The time bounds are tight in the sense that for many values of n, d, α, μ, and Δ, there is a graph with n nodes, maximum degree d, edge expansion α or node expansion μ, and an initial placement of tokens with imbalance Δ such that the time (for any algorithm) to balance to within even $\Delta/2$ tokens is at least $\Omega(d\Delta/\alpha)$. Similarly, in terms of node expansion, there exist classes of graphs where the time to balance to within even $\Delta/2$ tokens is at least $\Omega(d\Delta/\mu)$.

The multiport algorithm is simpler and deterministic. At each step, a token is sent from node u to node v across edge (u, v) if at the beginning of the step node u contains at least $2d + 1$ more tokens than node v. This algorithm was first analyzed in [2].

As in the single-port case, we analyze the multiport algorithm in terms of both edge expansion and node expansion. In terms of edge expansion, the algorithm balances to within $O((d^2 \log n)/\alpha)$ tokens in $O(\Delta/\alpha)$ steps. This bound is tight in the sense that for any network with edge expansion α, and any value Δ, there exists an initial distribution of tokens with imbalance Δ such that the time to reduce the imbalance to even $\Delta/2$ is $\Omega(\Delta/\alpha)$. In terms of node expansion, the algorithm balances to within $O((d \log n)/\mu)$ tokens in $O(\Delta/\mu)$ time. This bound is tight in the sense that for many values of d, n, and μ, and any value Δ, there exists an n-node, maximum degree d graph with node expansion μ and an initial distribution of tokens with imbalance Δ for which the time to balance to within $\Delta/2$ tokens is $\Omega(\Delta/\mu)$.

Both the single-port and the multiport algorithms will eventually locally balance the network, the single-port algorithm to within one token and the multiport algorithm to within $2d$ tokens. However, even after reducing the global imbalance to a small value, the time for either of these algorithms to reach a locally balanced state can be quite large. In particular, we show that after reaching a state that is globally balanced to within $O((d \log n)/\mu)$ tokens, the multiport algorithm may take another $\Omega(n^{1/2})$ steps to reach a state that is locally balanced to within $2d$ tokens. For networks with large node expansion and small degree, e.g., $\mu = \Omega(1)$ and $d = O(1)$, and small initial imbalance, e.g., $\Delta = O((d \log^2 n)/\mu)$, the time to locally balance the network, $\Omega(n^{1/2})$, may be much larger than the time, $O(\Delta/\mu) = O((d \log^2 n)/\mu^2) = O(\log^2 n)$, to reach a state that is globally balanced to within $O((d \log n)/\mu)$ tokens. Moreover, there exist networks with diameter $\Theta(\log n/\mu)$ for which even after reducing the global imbalance to the asymptotically best possible value of $O(d \log n/\mu)$ tokens in optimal time, the multiport algorithm can still take a long time to locally balance to within d tokens. We prove similar bounds for the single-port algorithm.

Thus far we have described a network model in which the nodes are synchronized by a global clock (i.e., a *synchronous* network) and in which the edges are assumed not to fail. With minor modifications, however, the load balancing algorithms can be made to work in both asynchronous and dynamic networks. In a *dynamic* network, the set of edges in the network may vary at each time step. In any time step, a *live* edge is one that can transmit one message in each direction. We assume that at each time step, each node in a synchronous dynamic network knows which of its edges are live. In an *asynchronous* network, the topology is fixed, but an adversary determines the speed at which each edge operates at every instant of time. For every undirected edge between two nodes, we allow at most two messages to be in transit at any instant in time. These messages may travel in opposite directions across the edge, or both may travel in one direction while no message travels in the opposite direction. An edge is said to be *live* for a unit interval of time if every message that was in transit across the edge (in either direction) at the beginning of the interval is guaranteed to reach the end of the edge by the end of the interval. We analyze the performance of the multiport load balancing algorithm under the assumption that at each time step, the set of live edges has some edge expansion α or node expansion μ.

We also study the off-line load balancing problem, in which every node has knowledge of the global state of the network. This problem has been studied on static synchronous networks in [29]. We use their results to obtain tight bounds on off-line load balancing in terms of edge expansion and node expansion. For the single-port model, we prove that any network can be balanced off-line in $\lceil(1+\mu)\Delta/\mu\rceil$ steps so that no node has more than two tokens more than the average. This result can be used to show that any network can be balanced off-line to within three tokens in at most $2\lceil(1+\mu)\Delta/\mu\rceil$ steps in the single-port model. Moreover, there exists a network and an initial token distribution for which any single-port off-line algorithm takes more than $\lceil(1+\mu)\Delta/\mu\rceil$ steps to balance the network to within one token. Similarly, in the multiport model, any network can be balanced off-line in at most $\lceil\Delta/\alpha\rceil$ steps so that no node contains more than d tokens more than the average. Using this result, we show that any network can be balanced to within $d+1$ tokens in at most $2\lceil\Delta/\alpha\rceil$ steps. It is easy to observe that for any network there exists an initial token distribution such that any algorithm will take at least $\lceil\Delta/\alpha\rceil$ steps to balance the network to within one token.

1.2. Previous and related work. Load balancing has been studied extensively because it comes up in a wide variety of settings, including adaptive mesh partitioning [17, 39], fine-grain functional programming [16], job scheduling in operating systems [13, 25], and distributed tree searching [22, 26]. A number of models have been proposed for load balancing, differing chiefly in the amount of global information used by the load balancing algorithm [2, 11, 12, 14, 27, 31]. In these models, algorithms have been proposed for specific applications; also, proposed heuristics and algorithms have been analyzed using simulations and queueing-theoretic techniques [28, 35, 37]. In what follows, we focus on models that allow only local algorithms and on prior work that takes an analytical approach to the load balancing problem.

Local algorithms restricted to particular networks have been studied on counting networks [4, 23], hypercubes [20, 34], and meshes [17, 29]. Another class of networks on which load balancing has been studied is the class of expanders. Peleg and Upfal [32] pioneered this study by identifying certain small-degree expanders as being suitable for load balancing. Their work has been extended in [9, 18, 33]. These algorithms use either strong expanders to approximately balance the network or the AKS sorting network [3] to perfectly balance the network. Thus, they do not work on networks of arbitrary topology. Also, these algorithms work by setting up fixed paths through the network on which load is moved and therefore cannot cope with changes in the network topology. In contrast, our local algorithm works on any arbitrary dynamic network that remains connected.

On arbitrary topologies, load balancing has been studied under two models. In the first model, any amount of load can be moved across a link in any time step [8, 12, 14, 15, 19, 36]. The second model is the one that we adopt here, namely, one in which at most one unit load can be moved across a link in each time step. Load balancing algorithms for the second model were first proposed and analyzed in [2] for the multiport variant and in [14] for the single-port variant. The upper bounds established by them are suboptimal by a factor of $\Omega(\log(n\Delta))$ or $\Omega(\sqrt{n})$, respectively. We improve these results for both single-port and multiport variants.

As remarked earlier, our multiport results (and those in [2]) hold even for dynamic or asynchronous networks. In general, work on dynamic and asynchronous networks has been limited. In work related to load balancing, for instance, an end-to-end communication problem, namely, one in which messages are routed from a single source to a single destination, has been studied in [1, 7] on dynamic networks. Our scenario is substantially more involved since we are required to move load between several sources and destinations simultaneously. Another result on dynamic networks is the recent analysis of a local algorithm for the approximate multicommodity flow problem [5, 6]. While their result has several applications including the end-to-end communication problem mentioned above, it does not seem to extend to load balancing. Our result on load balancing is related to their work in the technique; however, our algorithm and analysis are simpler and we obtain optimal bounds for our problem.

The convergence of local load balancing algorithms is related to that of random walks on Markov chains. Indeed the convergence bounds in both cases depend on the expansion properties of the underlying graph, and they are established using potential function arguments. There are, however, two important differences. First, the analysis of the rapid convergence of random walks [21, 30] relies on averaging arbitrary probabilities across any edge. This corresponds to sending an arbitrary (possibly nonintegral) load along an edge, which is forbidden in our model. In this sense, the analysis in [12] (and all references in the unbounded capacity model) are similar to the random walk analysis. Second, our argument uses an exponential potential function. The analyses in [12, 21, 30], in contrast, use quadratic potential functions. Our potential function and our amortized analysis were necessary, since a number of previous attempts using quadratic potential functions yielded suboptimal results [2, 14] for local load balancing.

As mentioned earlier, we consider only the static aspect of load balancing. For a recent survey on the dynamic aspect of this problem (i.e., when tokens can be created or destroyed while the tokens are being balanced), see [40].

1.3. Outline. The remainder of this paper is organized as follows. Section 2 contains some definitions. Section 3.1 analyzes the performance of the single-port algorithm. Section 3.2 analyzes the performance of the multiport algorithm. In section 4, we show that the time to reach a locally balanced state can be quite large, even if the network starts in a state that is well balanced globally. Section 5 describes extensions to dynamic and asynchronous networks. Finally, section 6 presents tight bounds on off-line load balancing.

2. Preliminaries.

For any network $G = (V, E)$ with n nodes and edge expansion α, we denote the number of tokens at $v \in V$ by $w(v)$. We denote the average number of tokens by ρ, i.e., $\rho = (\sum_{v \in V} w(v))/n$. For simplicity, throughout this paper we assume that ρ is an integer. We assign a unique *rank* from $[1, w(v)]$ to every token at v. The *height* of a token is its rank minus ρ. The height of a node is the maximum among the heights of all its tokens.

Consider a partition of V given by $\{S_i\}$, where the index i is any integer (positive, negative, or zero) and S_i may be empty for any i. Let $S_{>j}$ be $\cup_{i>j} S_i$. Similarly, we define $S_{\geq j}$, $S_{<j}$, and $S_{\leq j}$. We define index i to be *good* if $|S_i| \leq \alpha|S_{>i}|/2d$. An index that is not good is called a *bad* index. Thus, index i is good if there are at least $\alpha|S_{>i}|/2$ edges from nodes in $S_{>i}$ to nodes in $S_{<i}$. To observe this, note that the number of edges out of $S_{>i}$ is at least $\alpha|S_{>i}|$. On the other hand, the number of edges coming out of S_i is at most $d|S_i|$, which is at most $\alpha|S_{>i}|/2$ if i is good. Therefore, at least $\alpha|S_{>i}|/2$ edges go from nodes in $S_{>i}$ to nodes in $S_{<i}$.

For any bad index i, it follows from the equality $|S_i| = |S_{>i-1}| - |S_{>i}|$ that $|S_{>i}| < |S_{>i-1}|/(1+\alpha/(2d))$. Consider the reduction in $|S_{>i}|$ as i increases. For each bad index, there is a reduction by a factor of $1/(1+\alpha/(2d))$. Hence, there can be at most $\lceil \log_{(1+\alpha/(2d))} n \rceil$ bad indices because $(1+\alpha/(2d))^{\log_{(1+\alpha/(2d))} n} \geq n$. It follows that at least half of the indices in $[1, 2\lceil \log_{(1+\alpha/(2d))} n \rceil]$ are good.

Finally, we note that for $0 \leq a \leq 1$, $1+a \geq e^{a-a^2/2} \geq e^a/2$. Thus $\ln(1+a) \geq a/2$, implying $\log(1+a) = \Theta(a)$. We will use this result several times in the sections to follow, without further justification.

3. Analysis for static synchronous networks.

3.1. The single-port model.

In this section, we analyze the single-port load balancing algorithm that is described in section 1.1.

THEOREM 3.1. *For an arbitrary network G with n nodes, maximum degree d, edge expansion α, and initial imbalance Δ, the single-port algorithm balances within $O((d \log n)/\alpha)$ tokens in $O((d\Delta)/\alpha)$ steps, with high probability.*

For the sake of analysis, before every step we partition the set of nodes according to how many tokens they contain. For every integer i, we denote the set of nodes having $\rho + i$ tokens as S_i. Consider the first T steps of the algorithm, with T to be specified later. It holds that either $|S_{>0}| \leq n/2$ at the start of at least half the steps, or $|S_{\leq 0}| \leq n/2$ at the start of at least half the steps. Without loss of generality, assume the former is true. Thus, every subset of nodes in $S_{>0}$ expands, and we will use this expansion property to show that the number of nodes that have at least $\rho + 2\log_{(1+\alpha/(2d))} n$ tokens rapidly goes to zero.

Recall that at least half of the indices in $[1, 2\lceil \log_{(1+\alpha/(2d))} n \rceil]$ are good in any time step. Therefore, there exists an index j in $[1, 2\lceil \log_{(1+\alpha/(2d))} n \rceil]$ that is good in at least half of those time steps in which $|S_{>0}| \leq n/2$. Hence j is good in at least $T/4$ steps.

With every token at height x we associate a potential of $\phi(x)$, where $\phi : N \to R$ is defined as follows:

$$\phi(x) = \begin{cases} 0 & \text{if } x \leq j, \\ (1+\nu)^x & \text{otherwise,} \end{cases} \tag{3.1}$$

where $\nu = \alpha/(cd)$ and $c > 1$ is a real constant to be specified later. The potential of the network is the sum of the potentials of all tokens in the network. While transmitting a token, every node sends its token with maximum height. Similarly, any token arriving at a node with height h is assigned height $h+1$. It follows from the definition of the potential function, and the fact that the height of a token never increases, that the potential of the network never increases. In the following, we show that during any step when j is good, the expected decrease in the potential of the network is at least an $\varepsilon\nu^2$ fraction of the potential before the step, where $\varepsilon > 0$ is a real constant to be specified later.

Before proving Theorem 3.1, we present an informal outline of the proof. For simplicity, let us assume that G is a constant-degree expander, i.e., $d = O(1)$ and $\mu = \Omega(1)$. Consider the scenario in which all of the indices greater than j are bad. In this situation, for indices greater than j, the size of the set $S_{\geq i}$ decreases exponentially with increasing i, and hence the number of tokens with height i decreases exponentially with increasing i. If the rate of growth of $\phi(x)$ with increasing x is smaller than the rate of decrease of $|S_{\geq i}|$ with increasing i, then the total potential due to tokens at height i dominates the total potential due to tokens at height greater than i. In such a case the potential of $S_{>j}$ is essentially a constant times the potential of tokens at height $j+1$. In addition, if the potential of tokens at height at most j is zero, then in every step when j is good, there is a constant fraction potential drop because a constant fraction of the nodes in $S_{>j}$ send tokens to $S_{<j}$ in such a step. The exponential function we have defined in (3.1) satisfies the properties described above for c sufficiently large.

In general, the indices greater than j may form any sequence of good and bad indices, provided that the upper bound on the number of bad indices is respected. We consider the indices greater than j in reverse order and show by an amortized analysis that for each index i we can view all indices greater than or equal to i as bad. If i is bad, then this view is trivially preserved; otherwise, the number of edges from $S_{>i}$ to $S_{<i}$ is at least $\alpha|S_{>i}|/2$ and hence there is a significant potential drop across the cut $(S_{\leq i}, S_{>i})$. This drop can be used to rearrange the potential of $S_{>i}$ in order to maintain the view that all indices greater than i are bad. We then invoke the argument for the case in which all indices greater than j are bad, and complete the proof.

Consider step t of the algorithm. Let Φ_t denote the potential of the network after step $t > 0$. Let M_i be the set of tokens that are sent from a node in $S_{>i}$ to a node in $S_{<i}$. Note that a token may appear in several different sets M_i. Let $m_i = |M_i|$. We say that a token p has an *i-drop* of $\phi(i+1) - \phi(i)$ if p moves from a node in $S_{>i}$ to a node in $S_{<i}$. Thus, the potential drop due to a token moving on an edge from node $u \in S_i$ to node $v \in S_{i'}$, $i > i'+1$, can be expressed as the sum of k-drops for $i' < k < i$. In Lemma 3.2, we use this notion of i-drops to relate the total potential drop in step t, Ψ, to the m_i's.

LEMMA 3.2.

$$\Psi = \left(\sum_{i>j} m_i \nu (1+\nu)^i\right) + m_j (1+\nu)^{j+1}.$$

Proof. Let M be the set of tokens that are moved from a node in $S_{>j}$. (Note that tokens that start from and end at nodes in $S_{>j}$ also belong to M.) For any token p, let $a(p)$ (resp., $b(p)$) be the height of p after (resp., before) step t.

$$\begin{aligned}
\Psi &= \sum_{p \in M} (\phi(b(p)) - \phi(a(p))) \\
&= \sum_{p \in M} \sum_{a(p) \leq i < b(p)} (\phi(i+1) - \phi(i)) \\
&= \sum_{i \geq j} \sum_{p \in M_i} (\phi(i+1) - \phi(i)) \\
&= \left(\sum_{i>j} \sum_{p \in M_i} (\phi(i+1) - \phi(i))\right) + \sum_{p \in M_j} (\phi(j+1) - \phi(j)) \\
&= \left(\sum_{i>j} \sum_{p \in M_i} \nu (1+\nu)^i\right) + \sum_{p \in M_j} (1+\nu)^{j+1} \\
&= \left(\sum_{i>j} m_i \nu (1+\nu)^i\right) + m_j (1+\nu)^{j+1}.
\end{aligned}$$

(The second equation holds since the sum of $\phi(i+1)-\phi(i)$ over i telescopes. For the third equation, we interchange the order of summation and use the fact that $\phi(i)$ is zero for all $i \le j$. The fourth equation is obtained by separating the case $i \ge j$ into two cases $i > j$ and $i = j$. For deriving the fifth equation, we use (i) for all $i > j$, $\phi(i+1) - \phi(i) = \nu(1+\nu)^i$, and (ii) $\phi(j) = 0$. The last equation follows from the definition of m_i.) $\square$

We now describe the amortized analysis, which we alluded to earlier in this section, that we use to prove Theorem 3.1. We associate a charge of $\varepsilon\nu^2\phi(h)$ with each token at height h. We show that we can pay for all of the charges using the expected potential drop $E[\Psi]$, which implies a lower bound on $E[\Psi]$. We consider the indices in $[j+1,\ell]$ in reverse order, where ℓ is the maximum token height. For every i in $[j,\ell]$, we maintain a "debt" term, given by Γ_i below, which is the difference between the charges due to tokens at height greater than i and the sum of i'-drops for $i' > i$. We will place an upper bound on $E[\Gamma_i]$ that lets us view all of the indices in $[i+1,\ell]$ as bad indices. In other words, we upper bound $E[\Gamma_i]$ by $\varepsilon\nu|S_{\ge i}|(1+\nu)^i$. It follows from this upper bound and the informal argument outlined earlier in this section that the expected total debt can be paid for by the expected drop across index j.

Formally, for any $i > j$, we define

$$\Psi_i = \sum_{k \ge i} m_k \nu (1+\nu)^k,$$

$$\Gamma_i = (\varepsilon\nu^2)\left(\sum_{p:b(p)\ge i} (1+\nu)^{b(p)}\right) - \Psi_i.$$

We also define

$$\Gamma = (\varepsilon\nu^2)\left(\sum_{p:b(p)>j} (1+\nu)^{b(p)}\right) - \Psi.$$

Note that $\Phi_{t-1} = \sum_{p:b(p)>j}(1+\nu)^{b(p)}$ is the total potential of $S_{>j}$ prior to step t.

In order to prove the upper bound on $E[\Gamma_i]$, we place a lower bound on $E[m_i]$ that is obtained from the following lemma of [14].

LEMMA 3.3 (see [14]). *For any edge $e \in E$, the probability that e is selected in the matching is at least* $1/(8d)$. $\square$

LEMMA 3.4. *There exists a real constant $\varepsilon > 0$ such that for all $i > j$, we have* $E[\Gamma_i] \le (\varepsilon\nu)|S_{\ge i}|(1+\nu)^i$.

Proof. The proof is by reverse induction on i. If $i > \ell$, then the claim holds trivially since Γ_i and $|S_{\ge i}|$ are both equal to zero. (Recall that ℓ denotes the maximum token height.) Therefore, for the base case we consider $i = \ell$. Since $m_\ell = 0$, we have $\Psi_\ell = 0$. Thus, $\Gamma_\ell = (\varepsilon\nu^2)|S_\ell|(1+\nu)^\ell \le (\varepsilon\nu)|S_{\ge \ell}|(1+\nu)^\ell$, since $\nu = \alpha/(cd) \le 1/c \le 1$ by our choice of c.

For the induction step we consider two cases, depending on whether i is good or bad. We begin with the case when i is good. By the definition of a good index, we have $|S_i| \le \alpha|S_{>i}|/2d$. Since each node has at most d adjacent edges, there are at most $\alpha|S_{>i}|/2$ edges adjacent to nodes in S_i. Therefore, there are at most $\alpha|S_{>i}|/2$ edges from $S_{>i}$ to S_i. By the expansion property of the graph, $S_{<i}$ has at least $\alpha|S_{<i}|$ edges to nodes in $S_{\ge i}$, so there are at least $\alpha|S_{>i}|/2$ edges from $S_{>i}$ to $S_{<i}$. By Lemma 3.3, we have $E[m_i] \ge \alpha|S_{>i}|/(16d)$.

We are now ready to place a bound on $E[\Gamma_i]$. By definition, Γ_i can be calculated by subtracting the sum of i-drops from Γ_{i+1} and adding the charges due to tokens at height i. Therefore, we have

$$\begin{aligned}
E[\Gamma_i] &= E[\Gamma_{i+1}] + (\varepsilon\nu^2)|S_{\geq i}|(1+\nu)^i - E[m_i]\nu(1+\nu)^i \\
&\leq E[\Gamma_{i+1}] + (\varepsilon\nu^2)|S_{\geq i}|(1+\nu)^i - c\nu^2|S_{>i}|(1+\nu)^i/16 \\
&\leq E[\Gamma_{i+1}] - (\nu^2)|S_{\geq i}|(1+\nu)^i(f(c,\alpha,d) - \varepsilon) \\
&\leq (\varepsilon\nu)|S_{>i}|(1+\nu)^{i+1} - (\nu^2)|S_{\geq i}|(1+\nu)^i(f(c,\alpha,d) - \varepsilon) \\
&\leq (\varepsilon\nu)|S_{\geq i}|(1+\nu)^i((1+\nu) - \nu(f(c,\alpha,d) - \varepsilon)/\varepsilon),
\end{aligned}$$

where $f(c,\alpha,d) = c/(16(1+\alpha/(2d)))$. (In the first equation, we use the fact the number of tokens p such that $b(p) = i$ is $|S_{\geq i}|$. The second equation follows from the lower bound on $E[m_i]$. The third equation follows from the fact that $|S_{>i}| \geq |S_{\geq i}|/(1+\alpha/(2d))$ whenever i is a good index. The fourth equation follows from the induction hypothesis. The last equation follows from the fact that $|S_{>i}| \leq |S_{\geq i}|$.)

The second case is when i is bad. Thus $|S_i| > \alpha|S_{>i}|/(2d)$. We now place an upper bound on $E[\Gamma_i]$ as follows.

$$\begin{aligned}
E[\Gamma_i] &\leq E[\Gamma_{i+1}] + (\varepsilon\nu^2)|S_{\geq i}|(1+\nu)^i \\
&\leq (\varepsilon\nu)|S_{>i}|(1+\nu)^{i+1} + (\varepsilon\nu^2)|S_{\geq i}|(1+\nu)^i \\
&\leq (\varepsilon\nu)|S_{\geq i}|(1+\nu)^i((1+\nu)/(1+c\nu/2) + \nu).
\end{aligned}$$

(In the first equation, we use the fact the number of tokens p such that $b(p) = i$ is $|S_{\geq i}|$. The second equation follows from the induction hypothesis. The third equation follows from the fact that $|S_{\geq i}| > (1+\alpha/(2d))|S_{>i}|$ whenever i is a bad index.)

We now complete the induction step by determining values for c and ε such that the following equations hold:

$$((1+\nu) - \nu(f(c,\alpha,d) - \varepsilon)/\varepsilon) \leq 1, \tag{3.2}$$

$$(1+\nu)/(1+c\nu/2) + \nu \leq 1. \tag{3.3}$$

We set c to be any constant greater than or equal to $(\alpha/d) + 4$ (e.g., $c = 5$). For this choice of c, $\nu = \alpha/(cd) \leq (c-4)/c$, and hence $2\nu + c\nu^2/2 \leq c\nu/2$. Therefore, we have

$$\begin{aligned}
(1+\nu)/(1+c\nu/2) + \nu &= (1 + 2\nu + c\nu^2/2)/(1+c\nu/2) \\
&\leq (1+c\nu/2)/(1+c\nu/2) \\
&= 1.
\end{aligned}$$

Thus, (3.3) is satisfied. Since $\alpha \leq d$, we find that $f(c,\alpha,d) \geq c/24$. We now set $\varepsilon = c/48$ to establish (3.2). (For example, $c = 5$ and $\varepsilon = 5/48$.) □

We are now in a position to bound $E[\Gamma]$ on those steps in which j is good. By applying Lemma 3.4 with $i = j+1$, we obtain that $E[\Gamma_{j+1}] \leq (\varepsilon\nu)|S_{\geq j+1}|(1+\nu)^{j+1}$. If j is good, then by the definitions of Γ, Γ_{j+1}, and Ψ, we have

$$\begin{aligned}
E[\Gamma] &= E[\Gamma_{j+1}] - E[m_j](1+\nu)^{j+1} \\
&\leq E[\Gamma_{j+1}] - \alpha|S_{>j}|(1+\nu)^{j+1}/(16d) \\
&\leq (\varepsilon\nu)|S_{>j}|(1+\nu)^{j+1} - \alpha|S_{>j}|(1+\nu)^{j+1}/(16d) \\
&= \nu|S_{>j}|(1+\nu)^{j+1}(\varepsilon - c/16) \\
&\leq 0.
\end{aligned}$$

(The second equation follows from the fact that $E[m_j] \geq \alpha|S_{>j}|/16d$ whenever j is good. The third equation follows from the upper bound on $E[\Gamma_{j+1}]$. The fifth equation holds since $c/16 \geq \varepsilon$.)

We now derive a lower bound on the expected drop in the potential of the network during a sequence of T steps. By the definitions of Ψ and Γ, we have $\Phi_t = \Phi_{t-1} - \Psi$ and $\Gamma = \varepsilon\nu^2\Phi_{t-1} - \Psi$. If j is good during step t, we have $E[\Gamma] \leq 0$, and therefore $E[\Phi_t] \leq \Phi_{t-1}(1 - \varepsilon\nu^2)$, where the expectation is taken over the random matching selected in step t. Since j is good in at least $T/4$ steps, we obtain that $E[\Phi_{t+T}] \leq \Phi_t(1 - \varepsilon\nu^2)^{T/4}$, where the expectation is taken over all the random matchings in the T steps. By setting $T = \lceil(4\ln 4)/(\varepsilon\nu^2)\rceil$, we obtain $E[\Phi_{t+T}] \leq \Phi_t/4$. By Markov's inequality, the probability that $\Phi_{t+T} \geq \Phi_t/2$ is at most $1/2$. Therefore, using standard Chernoff bounds [10], we can show that in $T' = 8aT\lceil(\log\Phi_0 + \log n)\rceil$ steps, $\Phi_{T'} > 1$ with probability at most $O(1/(\Phi_0)^a + 1/n^a)$ for any constant $a > 0$.

If Δ is at most $2\log_{(1+\alpha/(2d))} n$, then the claim of the theorem holds trivially. Accordingly, we assume that Δ is greater than $2\log_{(1+\alpha/(2d))} n$ in what follows. Since Φ_0 is at least $(1+\nu)^{\Delta}$, Φ_0 is at least $n^{2/c}$. Therefore, $1/(\Phi_0)^a$ is inverse-polynomial in n. Since $\Phi_0 \leq n(1+\nu)^{\Delta+1}/\nu$, we have $\log\Phi_0 \leq (\Delta+1)(\nu) + \log n - \log\nu$. Therefore, for $T' = O(\Delta d/\alpha + d^2\log n/\alpha^2)$, we have $\Phi_{T'} < 1$ with high probability, which implies that after T' steps $|S_{>2\log_{(1+\alpha/(2d))} n}| = 0$ with high probability.

To establish balance in the number of tokens below the average, we use an averaging argument to show that after T' steps $|S_{<-2\log_{(1+\alpha/(2d))} n}| \leq n/2$ with high probability and then repeat the above arguments with the potential redefined appropriately. This proves Theorem 3.1.

3.2. The multiport model. In this section, we analyze the deterministic multiport algorithm described in section 1.1.

THEOREM 3.5. *For an arbitrary network G with n nodes, maximum degree d, edge expansion α, and initial imbalance Δ, the multiport algorithm load balances to within $O((d^2\log n)/\alpha)$ tokens in $O(\Delta/\alpha)$ steps.*

The proof of Theorem 3.5 is similar to that of Theorem 3.1. We assign a potential to every token, where the potential is exponential in the height of the token. We then show by means of an amortized analysis that a suitable rearrangement of the potential reduces every instance of the problem to a special instance that we understand well.

For the sake of analysis, before every step we partition the set of nodes according to how many tokens they contain. For every integer i, we denote the set of nodes having between $\rho - d + 2id$ and $\rho + d - 1 + 2id$ tokens as S_i. (Recall that ρ is the average number of tokens per node.) Consider the first T steps of the algorithm with T to be specified later. Without loss of generality, we assume that $|S_{>0}| \leq n/2$ holds in at least half of these steps. As shown in section 2, there exists an index j in $[1, 2\lceil\log_{(1+\alpha/(2d))} n\rceil]$ that is good in at least half of those steps in which $|S_{>0}| \leq n/2$. Hence in T steps of the algorithm, j is good in at least $T/4$ steps.

With every token at height h we associate a potential of $\phi(h)$, where $\phi : N \to R$ is defined as follows:

$$\phi(x) = \begin{cases} 0 & \text{if } x \leq 2jd, \\ (1+\nu)^x & \text{otherwise,} \end{cases}$$

where $\nu = \alpha/(cd^2)$ and $c > 0$ is a constant to be specified later. The potential of the network is the sum of the potentials of all tokens in the network.

While transmitting some number (say, m) of tokens in a particular step, a node sends the m highest-ranked tokens. Similarly, if m tokens arrive at a node during a step, they are assigned the m highest ranks within the node. Thus, tokens that do not move retain their ranks after the step. We now describe what specific ranks we assign to tokens that move during any step t. Let u be a node in $S_{<i}$ with height h at the start of step t. Let A (resp., B) be the set of tokens that u receives from nodes in $S_{>i}$ (resp., $S_{\leq i}$). We assign new ranks to tokens in A and B such that the rank of every token in A is less than that of every token in B. Let C be the set of tokens in A that attain height at most $h + (d/2)$ after the step. Since $|A| \leq d$, by the choice of our ranking, we have $|C| \geq |A|/2$. We call C the set of *primary* tokens. We also note that for any node v with height h all tokens leaving v during a step are at height at least $h - d + 1$ prior to the step.

It follows from the definition of the potential function and the fact that the height of a token never increases that the network potential never increases. In the following we show that whenever j is good the potential of $S_{>j}$ decreases by a factor of $\varepsilon\nu^2 d^2$, where $\varepsilon > 0$ is a real constant to be specified later. (For the sake of simplicity, we assume that d is even. If d is odd, we can replace d by $d+1$ in our argument without affecting the bounds by more than constant factors.)

For any token p, let $a(p)$ (resp., $b(p)$) be the index i such that S_i contains p after (resp., before) the step. (Note that the indexing is done prior to the step.) Let M_i be the set of primary tokens received by nodes in $S_{<i}$. Let $m_i = |M_i|$. Note that m_i is at least half the number of edges connecting nodes in $S_{<i}$ and nodes in $S_{>i}$. This is because a token is sent along every one of the edges connecting $S_{<i}$ and $S_{>i}$ and at least half the tokens received by any node in $S_{<i}$ from nodes in $S_{>i}$ are primary tokens. Lemma 3.6 establishes the relationship between the total potential drop Ψ in step t and the m_i's.

LEMMA 3.6.

$$\Psi \geq \left(\frac{1}{2} \sum_{i>j} m_i \nu d (1+\nu)^{(2i-1)d} \right) + m_j (1+\nu)^{2jd+1}.$$

Proof. Let M be the set of primary tokens that are moved from nodes in $S_{>j}$. (Note that primary tokens that start from a node in $S_{>j}$ and end at a node in $S_{>j}$ are in M.) Let p be a token in M. By the definition of a primary token, the height of p prior to the step is at least $2b(p)d - 2d + 1$ and the height after the step is at most $2a(p)d + 3d/2$. Moreover, p belongs to M_i for all i such that $a(p) < i < b(p)$.

$$\begin{aligned}
\Psi &\geq \sum_{p \in M} [\phi(2b(p)d - 2d + 1) - \phi(2a(p)d + 3d/2)] \\
&\geq \sum_{p \in M} \sum_{a(p) < i < b(p)} [\phi(2(i+1)d - 2d + 1) - \phi(2(i-1)d + 3d/2)] \\
&= \sum_{i \geq j} \sum_{p \in M_i} [\phi(2(i+1)d - 2d + 1) - \phi(2(i-1)d + 3d/2)] \\
&= \sum_{i > j} \sum_{p \in M_i} [\phi(2(i+1)d - 2d + 1) - \phi(2(i-1)d + 3d/2)] \\
&\quad + \sum_{p \in M_j} [\phi(2(j+1)d - 2d + 1) - \phi(2(j-1)d + 3d/2)] \\
&\geq \left(\frac{1}{2} \sum_{i>j} \sum_{p \in M_i} \nu d (1+\nu)^{2id-d} \right) + \sum_{p \in M_j} (1+\nu)^{2jd+1}
\end{aligned}$$

$$\geq \left(\frac{1}{2} \sum_{i>j} m_i \nu d (1+\nu)^{2id-d} \right) + m_j (1+\nu)^{2jd+1}.$$

(The first equation follows from the lower bound (resp., upper bound) on the height of a token p in M before (resp., after) the step. For the second equation, note that $2id - 2d + 1 \leq 2(i-1)d + 3d/2$. Therefore, $\phi(2id - 2d + 1) \leq \phi(2(i-1)d + 3d/2)$. The second equation now follows since the sum telescopes. The third equation is obtained by interchanging the sums and noting that $\phi(x)$ is 0 for $x \leq 2jd$. The fourth equation is obtained by partitioning M into the subsets $M \setminus M_j$ and M_j. The fifth equation is derived using the following calculations: (i) $\phi(2id+1) - \phi(2id - d/2) \geq ((1+\nu)^{d/2} - 1)(1+\nu)^{2id-d/2} \geq \nu d (1+\nu)^{2id-d}/2$, (ii) $\phi(2jd+1) = (1+\nu)^{2jd+1}$, and (iii) $\phi(2jd - d/2) = 0$. The last equation follows from the definition of m_i.) □

We establish Theorem 3.5 by means of an amortized analysis similar to the one used in section 3.1. We associate a charge of $\varepsilon \nu^2 d^2 \phi(h)$ with every token at height h. We show that we can pay for all of the charges using the potential drop Ψ and thus place a lower bound on Ψ. We consider the sets S_i in reverse order and maintain a "debt" term Γ_i for each i. Informally, Γ_i indicates the difference between the total charges due to tokens at height at least $2id - d$ and the current upper bound on the potential drop. Our amortized analysis terminates by showing that the total debt Γ is at most zero.

We now formally define Γ_i and Γ. For any token p, let $h(p)$ denote the height of p prior to the step. Thus $2b(p)d - d \leq h(p) \leq 2b(p)d + d - 1$. For $i > j$ and for a suitable constant $\varepsilon > 0$ to be specified later, we define

$$\Psi_i = \frac{1}{2} \sum_{k \geq i} m_k \nu d (1+\nu)^{2kd-d} \text{ and}$$

$$\Gamma_i = (\varepsilon \nu^2 d^2) \left(\sum_{p: h(p) \geq 2id-d} (1+\nu)^{h(p)} \right) - \Psi_i.$$

We also define

$$\Gamma = (\varepsilon \nu^2 d^2) \left(\sum_{p: h(p) > 2jd} (1+\nu)^{h(p)} \right) - \Psi.$$

For any step t', let $\Phi_{t'}$ denote the total potential after step t'. Thus, $\Phi_{t-1} = \sum_{p:h(p)>2jd} (1+\nu)^{h(p)}$ is the total potential prior to step t.

Lemma 3.7. *There exists a real constant $\delta > 0$ such that for all $i > j$, we have*

$$\Gamma_i \leq (\delta \nu d^2) |S_{\geq i}| (1+\nu)^{2id-d}.$$

Proof. The proof is by reverse induction on i. Let ℓ be the maximum token height. Consider first the case when $i > \lfloor (\ell + d)/2d \rfloor$. Since $2id - d > \ell$, there is no token with height at least $2id - d$. Hence $\Gamma_i \leq 0$ and $|S_{\geq i}| = 0$. Thus, the desired claim holds. We now consider $i = \lfloor (\ell + d)/2d \rfloor$. Since $\Psi_i = 0$, we have

$$\begin{aligned}\Gamma_i &\leq (2\varepsilon \nu^2 d^3) |S_{\geq i}| (1+\nu)^{\ell} \\ &\leq (2\varepsilon \nu^2 d^3) |S_{\geq i}| (1+\nu)^{2d(i+1)-d} \\ &\leq (\delta \nu d^2) |S_{\geq i}| (1+\nu)^{2id-d}.\end{aligned}$$

(The first equation holds because (i) each node in S_i has at most $2d$ tokens with height at least $2id-d$, and (ii) $h(p) \leq \ell$ for each token p. The second equation follows from the fact that $\ell < 2(i+1)d-d$. The third equation is obtained by choosing δ and ε such that $\delta > 2\varepsilon\nu d(1+\nu)^{2d}$. Note that for c sufficiently large, $(1+\nu)^{2d}$ can be set to an arbitrarily small constant.)

For the induction step we consider two cases. If i is good, then $|S_i| \leq \alpha|S_{>i}|/(2d)$ and $m_i \geq \alpha|S_{>i}|/4$. Therefore, we have

$$\begin{aligned}\Gamma_i &\leq \Gamma_{i+1} + (2\varepsilon\nu^2 d^3)|S_{\geq i}|(1+\nu)^{2id+d-1} - m_i\nu d(1+\nu)^{2id-d}/2\\ &\leq \Gamma_{i+1} + (2\varepsilon\nu^2 d^3)|S_{\geq i}|(1+\nu)^{2id+d-1} - c\nu^2 d^3|S_{>i}|(1+\nu)^{2id-d}/8\\ &\leq \Gamma_{i+1} - (\nu^2 d^3)|S_{\geq i}|(1+\nu)^{2id-d}(f(c,\alpha,d) - 2\varepsilon(1+\nu)^{2d})\\ &\leq (\delta\nu d^2)|S_{>i}|(1+\nu)^{2(i+1)d-d} - (\nu^2 d^3)|S_{\geq i}|(1+\nu)^{2id-d}(f(c,\alpha,d)-4\varepsilon)\\ &\leq (\delta\nu d^2)|S_{\geq i}|(1+\nu)^{2id-d}((1+\nu)^{2d} - \nu d(f(c,\alpha,d)-4\varepsilon)/\delta),\end{aligned}$$

where $f(c,\alpha,d) = c/(8(1+\alpha/(2d)))$. (The first equation holds because (i) each node in S_i has at most $2d$ tokens with height at least $2id-d$, and (ii) $h(p) \leq 2id+d-1$ for each token p that contributes to Γ_i and not to Γ_{i+1}. The third equation follows from the fact that $|S_{>i}| \geq |S_{\geq i}|/(1+\alpha/(2d))$. The fourth equation follows from the induction hypothesis and the equation $(1+\nu)^{2d} \leq 2$ for c sufficiently large. The last equation is derived using straightforward algebra.)

The second case is when i is bad. Thus $|S_i| > \alpha|S_{>i}|/(2d)$. We have

$$\begin{aligned}\Gamma_i &\leq \Gamma_{i+1} + (2\varepsilon\nu^2 d^3)|S_{\geq i}|(1+\nu)^{2id+d-1}\\ &\leq (\delta\nu d^2)|S_{>i}|(1+\nu)^{2(i+1)d-d} + 2\varepsilon\nu^2 d^3|S_{\geq i}|(1+\nu)^{2id+d-1}\\ &\leq (\delta\nu d^2)|S_{\geq i}|(1+\nu)^{2id-d}((1+\nu)^{2d}/(1+\alpha/(2d)) + 2\varepsilon\nu d(1+\nu)^{2d}/\delta).\end{aligned}$$

We now set c, δ, and ε such that $c > 4$, $c/12 - 4\varepsilon \geq 4\delta$, and $c/4 - 2\varepsilon/\delta \geq 4$. (One set of choices is $c = 50$, $\delta = 1$, and $\varepsilon = 1/24$.) Since $\alpha \leq d$, we have $f(c,\alpha,d) \geq c/12$. Since $c > 4$, we have $2\nu d < 1/2$, and hence $(1+\nu)^{2d} \leq 1 + \sum_{i>0}(2\nu d)^i = 1 + 2\nu d/(1-2\nu d) \leq 1 + 4\nu d$. Thus,

$$((1+\nu)^{2d} - \nu d(f(c,\alpha,d)-4\varepsilon)/\delta) \leq 1 + 4\nu d - 4\nu d \leq 1.$$

Since $\alpha/(2d) \leq 1/2$, we have $1/(1+\alpha/(2d)) \leq 1 - \alpha/2d + (\alpha/2d)^2 \leq 1 - \alpha/(2d) + \alpha/(4d) = 1 - \alpha/(4d)$, and hence

$$\begin{aligned}(1+\nu)^{2d}/(1+\alpha/(2d)) + 2\varepsilon\nu d(1+\nu)^{2d}/\delta &= (1+\nu)^{2d}(1/(1+\alpha/(2d)) + 2\varepsilon\nu d/\delta)\\ &\leq (1+\nu)^{2d}(1 - \alpha/4d + 2\varepsilon\nu d/\delta)\\ &= (1+\nu)^{2d}(1 - c\nu d/4 + 2\varepsilon\nu d/\delta)\\ &\leq (1+4\nu d)(1-4\nu d) < 1.\end{aligned}$$

(The second equation follows from the upper bound on $1/(1+\alpha/(2d))$. The fourth equation follows from the upper bound of $(1+4\nu d)$ on $(1+\nu)^{2d}$.)

Thus, in both cases, $\Gamma_i \leq (\delta\nu d^2)|S_{\geq i}|(1+\nu)^{2id-d}$. This completes the induction step. ▯

COROLLARY 3.8. *If j is good in step t, then we have $\Psi \geq \varepsilon\nu^2 d^2 \Phi_{t-1}$.*

Proof. Applying Lemma 3.7 with $i = j+1$, it follows that $\Gamma_{j+1} \leq (\delta\nu d^2)|S_{\geq j+1}|(1+\nu)^{2(j+1)d-d}$. If j is good, then $|S_{\geq j}| \leq (1+\alpha/(2d))|S_{>j}| \leq 3|S_{>j}|/2$ and $m_j \geq$

$\alpha|S_{>j}|/2$. Therefore,

$$\begin{aligned}\Gamma &\leq \Gamma_{j+1} + \varepsilon\nu^2 d^3 |S_{\geq j}|(1+\nu)^{2jd+d-1} - \alpha|S_{>j}|(1+\nu)^{2jd+1}/2 \\ &\leq (\delta\nu d^2)|S_{>j}|(1+\nu)^{2(j+1)d-d} \\ &\quad + (3\varepsilon\nu^2 d^3)|S_{>j}|(1+\nu)^{2jd+d-1}/2 - \alpha|S_{>j}|(1+\nu)^{2jd+1}/2 \\ &\leq (\nu d^2)|S_{>j}|(1+\nu)^{2(j+1)d-d}(\delta + 3\varepsilon\alpha/(2cd) - c/4) \\ &\leq 0\end{aligned}$$

for c, δ, and ε as chosen in the proof of Lemma 3.7. (In the first equation, the term $\varepsilon\nu^2 d^3|S_{\geq j}|(1+\nu)^{2jd+d-1}$ is an upper bound on the contribution to Γ_j by tokens in $S_{\geq j}$ since (i) tokens with height at least $2jd+d$ contribute to Γ_{j+1}, and (ii) each node in $S_{\geq j}$ has $d-2 \leq d$ tokens with height in the interval $[2jd+1, 2jd+d-1]$. Also, the third term in the first equation is the second term in the right-hand side of the equation of Lemma 3.6. In the second equation, we use the upper bounds on Γ_{j+1} and $|S_{\geq j}|$. The third equation follows from the choice of c, δ, and ε and the fact that for $c > 4$, we have $(1+\nu)^d \leq (1+\alpha/(cd^2))^d \leq (1+1/(cd))^d < (1+1/(4d))^d \leq e^{1/4} \leq 2$.)

By the definitions of Γ and Ψ, we have $\Phi_t \leq \Phi_{t-1} - \Psi$ and $\Gamma = \varepsilon\nu^2 d^2 \Phi_{t-1} - \Psi$. If j is good during step t, then $\Gamma \leq 0$ and the desired claim follows. ◻

By Corollary 3.8, if j is good during step t, then we have

$$\Phi_t \leq \Phi_{t-1}(1 - \varepsilon\nu^2 d^2).$$

After $T = \lceil 4 \ln \Phi_0/(\varepsilon\nu^2 d^2)\rceil$ steps, we have $\Phi_T \leq \Phi_0(1-\varepsilon\nu^2 d^2)^{T/4} < 1$. Since the height of each node is at most Δ initially, $\Phi_0 \leq n\sum_{2jd<i\leq\Delta}(1+\nu)^i \leq n(1+\nu)^{\Delta+1}/\nu$, $\ln \Phi_0 = O(\Delta\nu + \log n)$. Substituting $\alpha/(cd^2)$ for ν, we obtain that within $O(\Delta/\alpha + d^2 \ln n/\alpha^2)$ steps, $|S_{>2\log_{(1+\alpha/(2d))} n}| \leq |S_{>j}| = 0$.

We use an averaging argument to show that after T steps, $|S_{<-2\log_{(1+\alpha/(2d))} n}| \leq n/2$. By redefining the potential function and repeating the above analysis in the other direction, we obtain that in another T steps $|S_{<-4\log_{(1+\alpha/(2d))} n}| = 0$. This completes the proof of Theorem 3.5.

3.3. Results in terms of node expansion. The proofs of Theorems 3.1 and 3.5 can be easily modified to analyze the algorithm in terms of the node expansion μ of the graph instead of the edge expansion α. Recall that μ and α are related by the following inequalities: $\alpha/d \leq \mu \leq \alpha$. The primary modifications that need to be done to obtain bounds in terms of node expansion are to change the definition of a good index and to set ν appropriately. We call index i good if $|S_i| \leq \mu|S_{>i}|/2$. We set $\nu = \mu/c$ (resp., $\nu = \mu/(cd)$) for the single-port model (resp., multiport model).

By an argument similar to the one used in section 2, we obtain that the number of bad indices is at most $\lceil \log_{(1+\mu)} n\rceil$. (In fact, the argument in section 2 uses α/d as a lower bound on μ.) This bound on the number of bad indices leads to an upper bound of $O((\log n)/\mu)$ (resp., $O(d(\log n)/\mu)$) on the final imbalance obtained by the single-port algorithm (resp., multiport algorithm). For a bound on the number of steps, note that while deriving a bound on the potential drop in sections 3.1 and 3.2, we use the edge expansion α to obtain a lower bound on the number of tokens leaving sets $S_{>i}$. Since the best lower bound on α in terms of node expansion is μ, our time bounds here are obtained by substituting μ for α in the time bounds of Theorems 3.1 and 3.5, respectively. We thus obtain Theorems 3.9 and 3.11. Finally, Corollary 3.10 (resp., Corollary 3.12) follows from Theorems 3.1 and 3.9 (resp., Theorems 3.5 and 3.11).

THEOREM 3.9. *For an arbitrary network G with n nodes, maximum degree d, node expansion μ, and initial imbalance Δ, the single-port algorithm balances to within $O((\log n)/\mu)$ tokens in $O(d\Delta/\mu)$ steps with high probability.*

COROLLARY 3.10. *If $\Delta \geq (d \log n)/\mu$, the single-port algorithm balances to within $O(\log n/\mu)$ tokens in $O((d\Delta)/\alpha)$ steps with high probability. If $\Delta < (d \log n)/\mu$, the single-port algorithm balances to within $O(\log n/\mu)$ tokens in $O((d\Delta)/\mu)$ steps with high probability.*

THEOREM 3.11. *For an arbitrary network G with n nodes, maximum degree d, node expansion μ, and initial imbalance Δ, the multiport algorithm balances to within $O((d \log n)/\mu)$ tokens in $O(\Delta/\mu)$ steps.*

COROLLARY 3.12. *If $\Delta \geq (d^2 \log n)/\mu$, the multiport algorithm balances to within $O((d \log n)/\mu)$ tokens in $O(\Delta/\alpha)$ steps. If $\Delta < (d^2 \log n)/\mu$, the multiport algorithm balances to within $O((d \log n)/\mu)$ tokens in $O(\Delta/\mu)$ steps.*

4. Local load balancing can be expensive.

Here we show that upon reaching a state with small global imbalance, the algorithms presented in this paper may still take many steps until they reach a locally balanced state. More specifically, in section 4.1, we show that locally load balancing to within $2d$ tokens using the multiport algorithm of [2] described in section 1.1 can take $\Omega(\sqrt{n})$ more time than globally load balancing to within $O((d \log n)/\mu)$ tokens. We extend this bound to the single-port algorithm presented in [14]; i.e., upon reaching a state where the network is globally balanced to within $O((\log n)/\mu)$ tokens, the expected number of additional steps this algorithm may take to perform local balancing to within one token is $\Omega(d\sqrt{n})$. Furthermore, in section 4.2, we show that in the single-port case, the network may be one step away from being locally balanced to within one token but have an expected running time of $\Omega(\mu\sqrt{n})$ for reaching a locally balanced state. Finally, we prove upper bounds on the time each algorithm takes to reach a locally balanced state in section 4.3.

All results in this section are stated in terms of the node expansion of the network, rather than in terms of its edge expansion. This is done for the sake of making our arguments more intuitive and clear. Similar bounds can be derived in terms of edge expansion.

For any positive n and any μ, $0 < \mu < 1/72$, we present an example of a graph $G = (V, E)$ on n nodes with node expansion at least μ and with maximum degree d that depends on μ, where, given some initial distribution of tokens, locally balancing is difficult.

First, we define the node set V of G. Let μ_0 be equal to $\sqrt{8\mu}$ (note that $0 < \mu_0 < 1/3$). Let V be equal to $(\cup_{i=0}^{k} L_i) \cup (\cup_{i=0}^{k-1} R_i)$, where L_i and R_i are disjoint sets of $(1+\mu_0)^i$ nodes each and k will be specified shortly. For simplicity, we shall ignore integrality constraints on the number of nodes in each set. We could be more formal by setting the size of each set L_i or R_i to be $\lceil (1+\mu_0)^i \rceil$, but this would only make the calculations in this section more involved without changing the asymptotic results. For convenience, let L_{-1} (resp., R_{-1}) denote R_0 (resp., L_0) and R_k denote L_k. Note that each L_i and each R_i has size $(1+\mu_0)^i = \mu_0(\sum_{j=0}^{i-1} |L_j|)+1 = \mu_0(\sum_{j=0}^{i-1} |R_j|)+1$. Thus, setting $n = \sum_{j=0}^{k} |L_j| + \sum_{j=0}^{k-1} |R_j|$ and solving for k, we have $k = \Theta((\log n)/\log(1 + \mu_0)) = \Theta((\log n)/\log(1 + \mu)) = \Theta((\log n)/\mu)$. For simplicity, we assume that $k = (\log n)/\log(1 + \mu)$ and that k is even.

Given μ, we can choose the maximum degree d of G, independent of n, such that the following construction of the edges in G is possible. We obtain a similar structure for the R_i's by replacing L_j by R_j below. Let the only node in L_0 be adjacent to

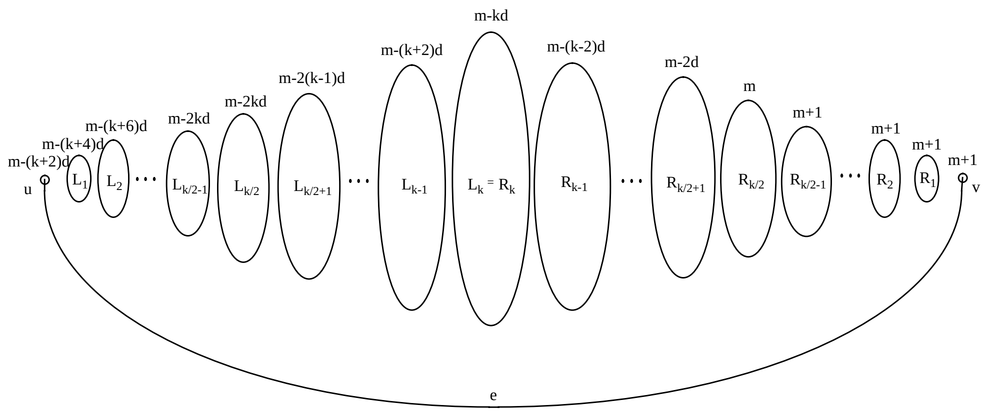


FIG. 4.1. *The initial distribution of tokens on G for the first case.*

every node in L_1. For all i, $0 \le i \le k$, we insert the edges between nodes in L_i such that (i) there are at most $d/2$ such edges adjacent to any node in L_i, and (ii) every subset S of L_i of size less than or equal to $2|L_i|/3$ has at least $\mu_0|S|$ neighbors in $L_i \setminus S$ (see [24, 38] for a proof that such a construction is possible). Also, let each node in L_i have $d(1+\mu_0)/(2(2+\mu_0))$ neighbors in L_{i+1} and each node in L_{i+1} have $d/(2(2+\mu_0))$ neighbors in L_i, $0 < i \le k-1$. For simplicity, we again ignore integrality constraints. Let S be any subset of L_i. There are $(|S|d(1+\mu_0))/(2(2+\mu_0))$ edges between S and L_{i+1}, and each node in L_{i+1} has $d/(2(2+\mu_0)$ neighbors in L_i. Thus, S has at least $(1+\mu_0)|S|$ neighbors in L_{i+1}.

Now we consider how L_{i+1} "expands" into L_i. We can use an approach similar to that of [24, 38] to show that we can choose the edges between L_i and L_{i+1}, respecting the degree constraints, such that any subset S of L_{i+1} of size less than or equal to $3|L_{i+1}|/(4(1+\mu_0))$ has at least $(1+\mu_0)|S|$ neighbors in L_i. This construction is possible since $(1+\mu_0)3|L_{i+1}|/(4(1+\mu_0)) = 3(1+\mu_0)|L_i|/4 < |L_i|$. The same analysis as in [24], but for a bipartite graph with node sets of sizes $|L_{i+1}|$ and $|L_{i+1}|/(1+\mu_0)$ and of regular node degrees $d/(2(2+\mu_0))$ and $d(1+\mu_0)/(2(2+\mu_0))$, respectively, applies here.

To complete the edge construction of G, let u be the only node in L_0, let v be the only node in R_0, and add the edge $e = (u, v)$ to the set E. Note that the diameter of G is $\Theta(k) = \Theta((\log n)/\mu)$.

We give a pictorial representation of the sets R_i's and L_i's in Figure 4.1. The initial distribution of tokens in Figure 4.1 (given by the quantities above the ovals representing each set L_i or R_i) may be ignored for the moment.

We still need to show that the graph G has node expansion at least μ, as claimed.

THEOREM 4.1. *The graph G, constructed as described, has node expansion at least μ.*

Proof. We will show how to account for the node expansion of any subset of G of at most $n/2$ nodes. Let U be a subset of V of size at most $n/2$. We will show that there exists a set of at least $\mu|U| = \mu_0^2|U|/8$ nodes outside of U that are all adjacent to nodes in U. For any subsets X and Y of V, we define the neighborhood of X in Y, $N_Y(X)$, as the subset of nodes in Y, but not in X, that are adjacent to some node in X, i.e., $N_Y(X) = \{y \in (Y \setminus X) \ : \ (x,y) \in E, x \in X\}$. If the set Y is not specified, assume $Y = V$. Let $U_i^L = U \cap L_i$ and $U_i^R = U \cap R_i$ for all $0 \le i \le k$. We consider two cases, according to whether the size of U_k^L is greater than $2|L_k|/3$ or not.

Case 1. If $|U_k^L| > 2|L_k|/3$, then let W^L (resp., W^R) be the union of the sets U_j^L (resp., U_j^R) of size greater than $2|L_j|/3$ (resp., $2|R_j|/3$) such that there is no U_q^L (resp., U_q^R), $q > j$, of size less than or equal to $2|L_q|/3$ (resp., $2|R_q|/3$). Let ℓ (resp., r) be the minimum index of a set U_j^L in W^L (resp., U_t^R in W^R). In case no such j (resp., t) exists, let $\ell = 0$ (resp., $r = 0$). Let $\mathcal{S}$ denote $(\cup_{j=\ell}^{k} L_j) \cup (\cup_{j=r}^{k-1} R_j)$, and let $W = W^L \cup W^R$. Note that since $|U_j^L| > 2|L_j|/3$ for all $j \geq \ell$ and $|U_j^R| > 2|R_j|/3$ for all $j \geq r$, there are at most $3|W|/2$ nodes in $\mathcal{S}$. Furthermore, since the set U has at most $n/2$ nodes, there are at most $3n/4$ nodes in $\mathcal{S}$. Hence, there are at least $n/4$ nodes that are not in $\mathcal{S}$, and so we must have either $\sum_{i=0}^{\ell-1} |L_i| \geq n/8$ or $\sum_{i=0}^{r-1} |R_i| \geq n/8$. Assume without loss of generality that the former is true. This implies that $|L_\ell| > \mu_0 n/8$.

We will account for the node expansion of U using the neighborhood of U_ℓ^L in $L_{\ell-1} \setminus U_{\ell-1}^L$. If $|U_\ell^L| < 3|L_\ell|/(4(1+\mu_0))$ (implying that $\mu_0 < 1/8$, since $|U_\ell^L| > 2|L_\ell|/3$), then

$$\begin{aligned}|N_{L_{\ell-1}}(U_\ell^L) \setminus U_{\ell-1}^L| &\geq \frac{(1+\mu_0)2|L_\ell|}{3} - \frac{2|L_{\ell-1}|}{3} = \frac{2|L_\ell|}{3}\left((1+\mu_0) - \frac{1}{(1+\mu_0)}\right)\\ &> \frac{2|L_\ell|\mu_0(2+\mu_0)}{3(1+\mu_0)} > \mu_0|L_\ell| > \frac{\mu_0^2 n}{8}\end{aligned}$$

(the second-to-last inequality follows from $(2+\mu_0)/(1+\mu_0) > 3/2$). Otherwise, any subset of $|U_\ell^L|$ of size $3|L_\ell|/(4(1+\mu_0))$ has at least $(1+\mu_0)3|L_\ell|/(4(1+\mu_0))$ neighbors in $L_{\ell-1}$. Thus

$$\begin{aligned}|N_{L_{\ell-1}}(U_\ell^L) \setminus U_{\ell-1}^L| &\geq \frac{(1+\mu_0)3|L_\ell|}{4(1+\mu_0)} - \frac{2|L_{\ell-1}|}{3} = |L_\ell|\left(\frac{3}{4} - \frac{2}{3(1+\mu_0)}\right)\\ &> \frac{9\mu_0|L_\ell|}{12(1+\mu_0)} > \frac{\mu_0|L_\ell|}{2} > \frac{\mu_0^2 n}{16}.\end{aligned}$$

We obtained the second-to-last inequality by substituting $1+\mu_0$ by $4/3$ in the denominator of the left-hand side of the inequality.

Hence, in Case 1 we have at least $\mu_0^2 n/16 \geq \mu_0^2|U|/8$ nodes not in U that are adjacent to the nodes in U.

Case 2. If $|U_k^L| \leq 2|L_k|/3$, then each set U_i^L and each set U_j^R is considered in exactly one of the following subcases. We prove the results that follow for the sets U_j^L's only. Similar results hold if we replace U_j^L by U_j^R and L_j by R_j in the two subcases below.

Case 2.1. Let i be the maximum index such that $|U_i^L| > 2|L_i|/3$ and $|U_{i+1}^L| \leq 2|L_{i+1}|/3$. If $|U_{i+1}^L| \leq |L_{i+1}|/3$, then the neighbors of U_i^L in L_{i+1} that are not in U_{i+1}^L can account for the node expansion of $\cup_{j=0}^{i+1} U_j^L$, because

$$\begin{aligned}|N_{L_{i+1}}(U_i^L) \setminus U_{i+1}^L| &> \frac{(1+\mu_0)2|L_i|}{3} - \frac{|L_{i+1}|}{3} = \frac{|L_{i+1}|}{3}\\ &> \frac{\mu_0}{3}\left(\sum_{j=0}^{i}|L_j|\right) \geq \frac{\mu_0}{4}\left(\left(\sum_{j=0}^{i}|L_j|\right) + |L_{i+1}| - 1\right)\\ &\geq \frac{\mu_0}{4}\left(\sum_{j=0}^{i+1}|U_j^L|\right)\end{aligned}$$

(the second-to-last inequality follows from the fact that $|L_{i+1}| = \mu_0(\sum_{j=0}^{i} |L_j|) + 1 \leq [(\sum_{j=0}^{i} |L_j|)/3] + 1$, since $\mu_0 < 1/3$). Otherwise, $|L_{i+1}|/3 < |U_{i+1}^L| \leq 2|L_{i+1}|/3$, and the neighborhood of U_{i+1}^L in L_{i+1} can account for the node expansion of $\cup_{j=0}^{i+1} U_j^L$, since

$$|N_{L_{i+1}}(U_{i+1}^L)| \geq \mu_0 |U_{i+1}^L| > \frac{\mu_0 |L_{i+1}|}{3} > \frac{\mu_0^2}{3}\left(\sum_{j=0}^{i} |L_j|\right)$$
$$\geq \frac{\mu_0^2}{4}\left(\sum_{j=0}^{i} |L_j| + |L_{i+1}| - 1\right) \geq \frac{\mu_0^2}{4}\left(\sum_{j=0}^{i+1} |U_j^L|\right).$$

Case 2.2. Now we consider every U_j^L, $i+2 \leq j \leq k$, that we did not account for in Case 2.1. Any set U_j^L that was not considered in Case 2.1 has size less than or equal to $2|L_j|/3$ by the choice of i in Case 2.1. Thus the neighborhood of each U_j^L in L_j, $i < j \leq k$, has size at least $\mu_0 |U_j^L|$, and so it accounts for the node expansion of U_j^L.

It follows from Cases 1 and 2 that U has at least $(\mu_0^2 |U|)/8 = \mu|U|$ neighbors outside of U in G. ◻

We group the sets R_i's and L_i's into $\mathcal{L}$ and $\mathcal{R}$, groups of $k/2$ consecutive sets, and $\mathcal{M}$, a group of $k+1$ consecutive sets (note that we have $2k+1$ distinct sets). Let $\mathcal{L} = \{L_0, L_1, \ldots, L_{k/2-1}\}$, $\mathcal{R} = \{R_0, R_1, \ldots, R_{k/2-1}\}$, and $\mathcal{M} = \{L_{k/2}, L_{k/2+1}, \ldots, L_{k-1}, L_k\ (= R_k), R_{k-1}, \ldots, R_{k/2}\}$. Our choice for $\mathcal{L}$, $\mathcal{M}$, and $\mathcal{R}$ is such that the number of sets in $\mathcal{L}$ is close to half the number of sets in $\mathcal{M}$.

4.1. It may be expensive to locally balance *G*. We give an initial distribution of tokens on G that has global imbalance of $\Theta((d \log n)/\mu)$. Then we show that the multiport algorithm will take $\Omega(\sqrt{n})$ steps to locally balance G to within $2d$ tokens. Suppose we have the following initial distribution of tokens on G: For every node z in $\mathcal{R}$, $w(z) = m+1$, where m is an integer such that $m \geq 2kd$; for all z in R_i, R_i in $\mathcal{M}$, let $w(z) = m - 2(i-k/2)d$; for all z in L_i, L_i in $\mathcal{M}$, let $w(z) = m - 2(3k/2 - i)d$; for all z in L_i, L_i in $\mathcal{L}$, let $w(z) = m - 2(i + k/2 + 1)d$. Then w is globally balanced to within $\Theta(dk) = \Theta((d \log n)/\mu)$ tokens, but it is not locally balanced to within $2d$ tokens, since $w(v) - w(u) = (k+2)d + 1 \geq 2d + 1$. See Figure 4.1.

We will maintain the invariant that at any step of the multiport algorithm, every node in L_i (resp., R_i) has the same number of tokens for all $0 \leq i \leq k$. The following lemma shows that this invariant holds.

LEMMA 4.2. *Suppose every node in L_i (resp., R_i) had the same number of tokens at the start of the multiport algorithm for all $0 \leq i \leq k$. Then every node in L_i (resp., R_i) has the same number of tokens at any step of the algorithm for all $0 \leq i \leq k$.*

Proof. We prove this lemma using induction, and without loss of generality, we will prove it for the sets L_i only. Suppose that every node in L_i had the same number of tokens at time step $t-1$. A node x in L_i sends a token to one of its neighbors y in L_{i+1} only if it has at least $2d+1$ more tokens than y. Thus if at time t, x sends a token to some y in L_{i+1}, then it sends a token to all of its neighbors in L_{i+1}, since all of them had the same number of tokens at time $t-1$. Note that x has at least $2d+1$ tokens and x has at most d neighbors. Hence at time t, every edge between L_i and L_{i+1} is traversed by a token. Since every node in L_{i+1} is adjacent to the same number of nodes in L_i, they all receive the same number of tokens from L_i. We can use a similar argument for tokens that move from L_i to L_{i-1}. For $i = k$, consider only tokens moving from L_k ($= R_k$) to L_{k-1} (and R_{k-1}). No token moves between any two nodes in L_i, since all nodes in L_i had the same number of tokens at time $t-1$ (thus we can ignore the edges inside each set L_i and R_i). □

Now we prove the main theorem in this section for the multiport model.

THEOREM 4.3. *The multiport algorithm may take $\Omega(\sqrt{n})$ steps to locally balance G, even if G is globally balanced to within $\Theta((d \log n)/\mu)$ tokens initially.*

Proof. Assume we have a initial token distribution on G as defined above. The number of nodes in $\mathcal{R}$, as well as in $\mathcal{L}$, is proportional to $|R_{k/2-1}| = (1+\mu_0)^{k/2-1} = (1+\mu_0)^{\frac{\log n}{2\log(1+\mu)}-1} > \sqrt{n}/(1+\mu_0) > \sqrt{n}/2$. We claim that in order for G to be locally balanced to within $2d$ tokens, we need to move at least $\sqrt{n}/2$ tokens from $\mathcal{R}$ to $\mathcal{L}$ across edge e. Since at most one token at a time can traverse e, this will require time $\Omega(\sqrt{n})$. Our proof proceeds as follows.

(1) Since every node in R_j (resp., L_j) for $0 \le j \le k/2-1$ is identical with respect to both the number of tokens it has (by Lemma 4.2) and the number of neighbors it sees in R_{j-1} and R_{j+1} (resp., L_{j-1} and L_{j+1}), we observe that the tokens in G flow as follows. Tokens will be sent from v to u across edge e until u has $2d+1$ more tokens than a node in L_1. Then, every node in L_1 receives a token from u. This process continues until the nodes in L_1 each have at least $2d+1$ more tokens than a node in L_2. Then every node in L_1 will send a token to each of its neighbors in L_2 (by Lemma 4.2, every node in L_2 receives the same number of tokens from the nodes in L_1). Continuing in this fashion, the flow of tokens in $\mathcal{L}$ will proceed only from left to right, i.e., tokens never move from L_i to L_{i-1}, or inside L_i, for all L_i in $\mathcal{L}$. In parallel, as the number of tokens in v gets small, the nodes in R_1 will all send a token to v. When the nodes in R_1 have each sent $2d+1$ tokens to v, the nodes in R_2 will all send a token to each of its neighbors in R_1, etc. Thus, as in $\mathcal{L}$, the tokens also flow only from left to right in $\mathcal{R}$ (i.e., tokens never move from R_i to R_{i+1}, or inside R_i, for all R_i in $\mathcal{R}$). Thus, no token ever moves from $\mathcal{R}$ to $\mathcal{M}$ or from $\mathcal{M}$ to $\mathcal{L}$.

(2) Now we show that only after $\sqrt{n}/2$ steps have elapsed can we have (i) $w(x) - w(y) \le 2d$ for all x in L_i for all y in L_{i+1} for all L_i in $\mathcal{L}$, i.e., $\mathcal{L}$ is locally balanced, and (ii) $w(u) > m - (k+2)d$. Suppose we reach such a configuration at some time t. Then every node in $\mathcal{L}$ has at least one more token than it had initially (since $w(u) = m-(k+2)d$ and $w(x) - w(y) = 2d$ for all x in L_i, y in L_{i+1}, and L_i in $\mathcal{L}$, initially). That is, we have at least $|\mathcal{L}| \ge \sqrt{n}/2$ "extra" tokens in $\mathcal{L}$ at time t, all of which have reached $\mathcal{L}$ by traversing e from v to u, since no token moves from $\mathcal{M}$ to $\mathcal{L}$. Hence $t \ge \sqrt{n}/2$.

(3) We also show that only after $\sqrt{n}/2$ steps have elapsed can we have (i) $w(y) - w(x) \le 2d$ for all x in R_i for all y in R_{i+1} for all R_i in $\mathcal{R}$, i.e., $\mathcal{R}$ is locally balanced, and (ii) $w(v) \le m - kd$. A counting argument (similar to the one above) on the number of tokens in $\mathcal{R}$ and the fact that no token is ever sent from $\mathcal{R}$ to $\mathcal{M}$ is sufficient to show this.

From (2) and (3), we conclude that on each of the first $\sqrt{n}/2$ steps the following holds: Either $w(u) \le m-(k+2)d$ and $w(v) > m - kd$, and so v sends a token to u, or the subnetwork induced by $\mathcal{L} \cup \mathcal{R}$ is not $2d$-locally balanced. Thus G is not locally balanced before the first $\sqrt{n}/2$ steps. Hence the algorithm takes $\Omega(\sqrt{n})$ time to locally balance G. □

A similar result holds for the single-port model. Assume we have the following initial distribution of tokens: for every node z in $\mathcal{R}$, $w(z) = m+1$, where m is an integer such that $m \ge k$; for all z in R_i, R_i in $\mathcal{M}$, let $w(z) = m+k/2-i$; for all z in L_i,

L_i in $\mathcal{M}$, let $w(z) = m-(3k/2-i)$; for all z in L_i, L_i in $\mathcal{L}$, let $w(z) = m-(i+k/2+1)$.

The arguments used in the proof of Theorem 4.3 can be easily modified to hold for the single-port model with the initial distribution of tokens defined above, since the lower bound on the number of steps required to reach a locally balanced state is given only in terms of how many tokens traverse the edge e. Lemma 4.2, which is used to show that no token moves from $\mathcal{M}$ to $\mathcal{L}$ without traversing edge e, no longer holds in the single-port model. Instead, we prove Lemma 4.4, which implies that no token moves from $\mathcal{M}$ to $\mathcal{L}$ without traversing edge e, as stated in Corollary 4.5. Recall that in the single-port algorithm, a token moves from node x to node y at some step only if edge (x, y) is selected to be in the matching, and x has at least two more tokens than y, at that step.

We first prove Lemma 4.4, from which we derive Corollary 4.5. Let M denote the set of tokens in $\mathcal{M}$ either that were initially in $\mathcal{M}$ or that moved from $\mathcal{R}$ to $\mathcal{M}$ without using the edge e (i.e., tokens that moved from $\mathcal{R}$ to $\mathcal{M}$ through some node in $R_{k/2}$) at any step of the single-port algorithm. Without loss of generality, assume that if a node in $\mathcal{M}$ sends a token at step t of the algorithm, it will send a token that is not in M if it has one for all t.

LEMMA 4.4. *At any step of the single-port algorithm for any node x in $\mathcal{M}$, the number of tokens on x that belong to M is at most the total number of tokens on x initially.*

Proof. By definition, a token in M is either a token that was in $\mathcal{M}$ initially or a token that moved from $\mathcal{R}$ to $\mathcal{M}$ through some node in $R_{k/2}$. Suppose, for the sake of contradiction, that at step t, a node x in $\mathcal{M}$ has one more token in M than it had initially. Assume x had b tokens initially. There exists a sequence of nodes $x = x_1, \ldots, x_p$ such that (i) x_i is adjacent to x_{i+1} in G, (ii) x_i had at least $b+i$ tokens at time t_i, (iii) $t_p < \cdots < t_1 = t$, and (iv) x_p is in $R_{k/2}$ and x_i, $i \neq p$, is not in $R_{k/2}$. There are two cases to consider.

(1) If no x_i is in $\mathcal{L}$ (i.e., every node x_i is in $\mathcal{M}$), then let q be the distance in $\mathcal{M}$ from x to x_p. Thus x_p has at least $b+p \geq b+q+1 = m+1$ tokens at time t_p. But no node in $R_{k/2}$ can have $m+1$ tokens, since $m+1$ is the maximum number of tokens in G initially, and no node in $R_{k/2}$ had that many tokens initially.

(2) Otherwise, let x_j (resp., $x_{j'}$) be the first (resp., last) node in the sequence that is not in $\mathcal{M}$. Then x_{j-1} and $x_{j'+1}$ belong to $L_{k/2}$. Let q be the distance from v to x_{j-1} in $\mathcal{M}$. Then $x_{j'+1}$ has at least $b+p \geq b+q+2 \geq (m-k)+q+1$ tokens at step $t_{j'+1}$. Thus x_p has at least $b+q+k+2 \geq m+q+1 \geq m+1$ tokens at step t_p, a contradiction (see item (1)). □

COROLLARY 4.5. *No token initially in $\mathcal{M} \cup \mathcal{R}$ ever moves from $\mathcal{M}$ to $\mathcal{L}$ without traversing edge e.*

Proof. By Lemma 4.4, no node x in $L_{k/2}$ will ever have more tokens in M than it had initially. Since the number of tokens on x at the beginning of the algorithm, $m-k$, is minimal (over the entire network), it follows that x will have exactly $m-k$ tokens that belong to M at any step of the algorithm. Thus x will never send a token that belongs to M to any other node in G. Since a token can move from $\mathcal{M}$ to $\mathcal{L}$ only through some node in $L_{k/2}$, the corollary follows. □

Any edge is selected independently with probability $O(1/d)$ at each iteration of the single-port algorithm. Thus an edge e is selected, on average, an $O(1/d)$ fraction of the time. Hence, we can show that it will take $\Omega(d\sqrt{n})$ expected time for G to be locally balanced to within one token in the single-port model, even if initially G is globally balanced to within $O((\log n)/\mu)$ tokens, as stated in the theorem below.

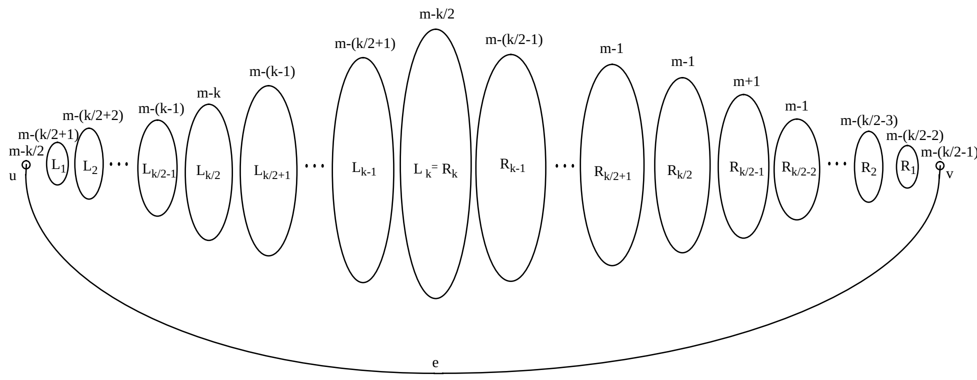


FIG. 4.2. *The initial distribution of tokens on G for the second case.*

THEOREM 4.6. *The single-port algorithm may take $\Omega(d\sqrt{n})$ expected number of steps to locally balance G to within one token, even if G is globally balanced to within $O((\log n)/\mu)$ tokens initially.*

4.2. The single-port algorithm may diverge from an almost locally balanced state. In this section we consider the single-port model only. Suppose we have the following distribution of tokens on G: for all z in $R_{k/2-1}$, let $w(z) = m + 1$ (where m is an integer such that $m \geq k$), and for all z in R_i, $i \leq k/2 - 2$ (note that $R_i \in \mathcal{R}$), let $w(z) = m - (k/2 - i - 1)$; for all z in $R_{k/2}$, let $w(z) = m - 1$; for all z in R_i, $i \geq k/2 + 1$ (note that $R_i \in \mathcal{M}$), let $w(z) = m - (i - k/2)$; for all z in L_i, L_i in $\mathcal{M}$, let $w(z) = m - (3k/2 - i)$; for all z in L_i, L_i in $\mathcal{L}$, let $w(z) = m - (k/2 + i)$. Thus w is globally balanced to within $O((\log n)/\mu)$ tokens but it is not locally balanced to within one token, since $w(x) - w(y) = 2$, for any x in $R_{k/2-1}$ and y in $R_{k/2} \cup R_{k/2-2}$. See Figure 4.2.

The intuition for this case is that if all tokens move in the "right direction" initially, we reach a locally balanced state in a single time step. Otherwise, if a large number of tokens move in the "wrong direction" in the first step, it will take many steps until we reach such a state. If every node in $R_{k/2-1}$ is matched with some node in $R_{k/2}$ (we can construct G such that every node in $R_{k/2-1}$ has a distinct neighbor in $R_{k/2}$), G reaches local balance in a single time step. On the other hand, if some tokens move across a matching between the nodes in $R_{k/2-1}$ and $R_{k/2-2}$, then these tokens will continue moving "down" (nondecreasing indices of R_i) and will never move "up." The expected size of such a matching will be $\Omega(|R_{k/2-1}|/d)$ (each node in $R_{k/2-1}$ has $d/(2(2 + \mu_0)) \geq d/5$ neighbors in $R_{k/2-2}$). Using an analysis similar to that of section 4.1 for the single-port model, we see that no token that was initially in $\mathcal{M} \cup \mathcal{R}$ moves from $\mathcal{M}$ to $\mathcal{L}$ without traversing edge e and that either $w(v) > m - k/2 + 1$ and $w(u) \leq m - k/2$ or $\mathcal{L} \cup \mathcal{R}$ is not locally balanced for each of the $\Omega(|R_{k/2-1}|/d)$ initial steps. Thus, once any of the tokens that moved from $R_{k/2-1}$ to $R_{k/2-2}$ in the first step reaches v, it will eventually traverse e onto u.

Hence, eventually all tokens that moved from $R_{k/2-1}$ to $R_{k/2-2}$ in the initial step will reach u. Since $|R_{k/2-1}| \geq \mu\sqrt{n}/((1 + \mu_0)^2)$ and $1 < (1 + \mu_0)^2 < 2$, the expected number of edges between nodes in $R_{k/2-1}$ and nodes in $R_{k/2-2}$ in a selected matching is $\Omega(|R_{k/2-1}|/d) = \Omega(\mu\sqrt{n}/d)$. The expected time for edge e to be in a selected matching is at least d. Thus the expected time for G to be locally balanced to within one token is $\Omega(\mu\sqrt{n})$. This result is stated in the following theorem.

THEOREM 4.7. *There exists an initial distribution of tokens from which the network G can be locally balanced to within one token in one step, but for which the expected number of steps required by the single-port algorithm to locally balance G to within one token is $\Omega(\mu\sqrt{n})$.*

4.3. Convergence to a locally balanced state. We now prove that if the graph G is globally balanced to within Δ tokens, in $O(n\Delta^2/d)$ subsequent steps the multiport algorithm locally balances G to within $2d$ tokens. Define the potential Φ of the network as $\sum_{v\in V}(w(v)-\rho)^2$. If the network is globally balanced to within Δ tokens, then $\Phi = O(n\Delta^2)$. At any step, if there exists an edge (u,v) such that $|w(u)-w(v)| \geq 2d+1$, then a token is transmitted along (u,v) resulting in a potential drop of at least d. Thus, within $O(n\Delta^2/d)$ steps the network is globally balanced. Similarly, it is not difficult to show that if the graph G is globally balanced to within Δ tokens, then the single-port algorithm locally balances to within one token in $O(nd\Delta^2)$ subsequent steps with high probability, since a token transmitted along an edge results in a potential drop of at least one and an edge is selected with probability at least $1/(8d)$.

5. Extension to dynamic and asynchronous networks. In this section, we extend our results of section 3.2 for the multiport model to dynamic and asynchronous networks. We first prove that a variant of the local multiport algorithm is optimal on dynamic synchronous networks in the same sense as for static synchronous networks. We then use a result of [2] that relates the dynamic synchronous and asynchronous models to extend our results to asynchronous networks.

In the dynamic synchronous model, the edges of the network may fail or succeed dynamically. An edge $e \in E$ is *live* during step t if e can transmit a message in each direction during step t. We assume that in each step each node knows which of its adjacent edges are live. The local load balancing algorithm for static synchronous networks can be modified to work on dynamic synchronous networks. The algorithm presented here is essentially the same as in [2].

Since edges may fail dynamically, a node u may have no knowledge of the height of a neighboring node v and hence may be unable to decide whether to send a token to v. In our algorithm, which we call **DS**, every node u maintains an estimate $e^u(v)$ of the number of tokens at v for every neighbor v of u. (The value of $e^u(v)$ at the start of the algorithm is arbitrary.) In every step of the algorithm, each node u performs the following operations:

(1) For each live neighbor v of u, if $w(u) - e^u(v) > 12d$, u sends a message consisting of $w(u)$ and a token; otherwise, u sends a message consisting only of $w(u)$. Next, $w(u)$ is decreased by the number of tokens sent.

(2) For each message received from a live neighbor v, $e^u(v)$ is updated according to the message, and if the message contains a token, $w(u)$ is increased by one.

Unlike the algorithm for static networks, the above algorithm may (temporarily) worsen the imbalance since a node may have an old estimate of the height of one of its neighbors. Two anomalies may occur while executing **DS**: (i) a token sent by u to v may gain height as it is possible for $w(u)-e^u(v)$ to be greater than $12d$ even if $w(u)$ is at most $w(v)$, and (ii) node u may not send a token to v as it is possible for $w(u)-e^u(v)$ to be at most $12d$ even if $w(u) - w(v)$ is much larger than $12d$. Consequently, the analysis for dynamic networks is more difficult than for static networks. We employ a more complicated amortized analysis to account for the above anomalies.

For every integer i, let S_i denote the set of nodes that have at least $\rho - 12d + 24id$ and at most $\rho + 12d - 1 + 24id$ tokens. Consider T steps of **DS**. We assume without loss of generality that $|S_{>0}| \leq n/2$ at the start of at least $T/2$ steps. As shown in section 2, there exists an index j in $[1, 2\lceil \log_{(1+\alpha/(2d))} n \rceil]$ that is good in at least half of those steps in which $|S_{>0}| \leq n/2$. (Recall that index i is good if $|S_i| \leq \alpha |S_{>i}|/2d$.) If index j is good at the start of step t, we call t a *good* step. For any token p, let $h_t(p)$ denote the height of p after step t, $t > 0$. For convenience, we denote the height of p at the start of **DS** by $h_0(p)$. Similarly, for $t \geq 0$, we define $h_t(u)$ for every node u and $e_t^u(v)$ for every edge (u, v).

With every token at height h, we associate a potential of $\phi(h)$, where $\phi : N \to R$ is defined as follows:

$$\phi(x) = \begin{cases} 0 & \text{if } x \leq 24jd - 11d, \\ (1+\nu)^x & \text{otherwise,} \end{cases}$$

where $\nu = \alpha/(cd^2)$ and $c > 0$ is a constant to be specified later. Let Φ_t denote the total potential of the network after step t. Let Ψ_t denote the potential drop during step t.

We analyze **DS** by means of an amortized analysis over the steps of the algorithm. Let E_t be the set $\{(u, v) : (u, v) \text{ is live during step } t,\ u \in S_{>j}, \text{ and } h_{t-1}(u) - h_{t-1}(v) \geq 24d\}$. For every step t, we assign an amortized potential drop of

$$\hat{\Psi}_t = \frac{1}{50} \sum_{\substack{(u,v)\in E_t \\ h_{t-1}(u) > h_{t-1}(v)}} (\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d)).$$

The definition of $\hat{\Psi}_t$ is analogous to the amount of potential drop that we use in step t in the argument of section 3.2 for the static case. By modifying that argument slightly and choosing appropriate values for the constants c and ε, we show the following lemma.

LEMMA 5.1. *If the live edges of G have an edge expansion of α during every step of* **DS**, *then for every good step t we have $\hat{\Psi}_t \geq \varepsilon \nu^2 d^2 \Phi_{t-1}$, where ε is an appropriately chosen constant.*

Proof (sketch). Let M_i denote the set of live edges between nodes in $S_{<i}$ and nodes in $S_{>i}$. Let $m_i = |M_i|$. For any node u, let $g(u)$ represent the group to which u belongs prior to step t. We now place a lower bound on $\hat{\Psi}_t$ which is analogous to that on Ψ in Lemma 3.6 of section 3.2. By the definition of $\hat{\Psi}_t$, we have

$$\begin{aligned}
\hat{\Psi}_t &= \frac{1}{50} \sum_{\substack{(u,v)\in E_t \\ h_{t-1}(u) > h_{t-1}(v)}} (\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d)) \\
&\geq \frac{1}{50} \sum_{\substack{(u,v)\in E_t \\ h_{t-1}(u) > h_{t-1}(v)}} \sum_{g(v)<i<g(u)} (\phi(24(i+1)d - 13d) - \phi(24(i-1)d + 13d)) \\
&= \frac{1}{50} \sum_{i\geq j} \sum_{\substack{(u,v)\in M_i \\ h_{t-1}(u) > h_{t-1}(v)}} (\phi(24(i+1)d - 13d) - \phi(24(i-1)d + 13d)) \\
&= \frac{1}{50} \sum_{i>j} \sum_{\substack{(u,v)\in M_i \\ h_{t-1}(u) > h_{t-1}(v)}} (\phi(24(i+1)d - 13d) - \phi(24(i-1)d + 13d)) \\
&\quad + \frac{1}{50} \sum_{\substack{(u,v)\in M_j \\ h_{t-1}(u) > h_{t-1}(v)}} \phi(24(j+1)d - 13d)
\end{aligned}$$

$$\geq \frac{22}{50}\sum_{i>j}\sum_{\substack{(u,v)\in M_i\\ h_{t-1}(u)>h_{t-1}(v)}} \nu d(1+\nu)^{24id-11d} + \frac{1}{50}\sum_{\substack{(u,v)\in M_j\\ h_{t-1}(u)>h_{t-1}(v)}} (1+\nu)^{24jd+11d}$$

$$\geq \frac{22}{50}\sum_{i>j} m_i \nu d(1+\nu)^{24id-11d} + \frac{1}{50} m_j (1+\nu)^{24jd+11d}.$$

(For the second equation, note that $24id-13d \leq 24(i-1)d+13d$. Therefore, $\phi(24id-13d) \leq \phi(24(i-1)d+13d)$. The second equation now follows since the sum telescopes. The third equation is obtained by interchanging the sums and noting that $\phi(x)$ is zero for $x \leq 24jd-11d$. The fourth equation is obtained by partitioning the set M into subsets $M \setminus M_j$ and M_j. The fifth equation uses the following calculations: (i) $\phi(24id+11d)-\phi(24id-11d) \geq ((1+\nu)^{22d}-1)(1+\nu)^{24id-11d} \geq 22d(1+\nu)^{24id-11d}$, (ii) $\phi(24jd+11d)=(1+\nu)^{24jd+11d}$, and (iii) $\phi(24jd-11d)=0$. The last equation follows from the definition of m_i.)

We next establish claims similar to Lemma 3.7 and Corollary 3.8 of section 3.2 by modifying the constants in the proofs. Thus we have $\hat{\Psi}_t \geq \varepsilon\nu^2 d^2 \Phi_{t-1}$ for an appropriately chosen constant ε. □

The following lemma relates the amortized potential drops to the actual potential drops.

Lemma 5.2. *For any initial load distribution and any step $t' > 0$, we have*

$$\sum_{t\leq t'} \Psi_t \geq \left(\sum_{t\leq t'} \hat{\Psi}_t\right) - 2\Phi_0 - n^2\phi(24jd). \tag{5.1}$$

In order to prove Lemma 5.2, we need to address two issues that arise in the dynamic setting: (i) potential gains, i.e., when a token gains height, and (ii) the lack of a potential drop across edges that join nodes differing by at least $24d$ tokens. We show that for any of the above events to occur, "many" tokens should have lost height in previous steps. We use a part of this prior potential drop to account for (i) and (ii). At a high level, our proof follows the lines of Lemma 3 of [2]. However, since the potential functions involved are different, the two proofs differ considerably in the details. We have included a complete proof of Lemma 5.2 in Appendix B.

The main result follows from Lemmas 5.1 and 5.2. We first show that within $O(1/(\varepsilon\nu^2 d^2))$ steps, there is a step when the actual potential of the network either decreases by a factor of 2 or falls below a threshold value.

Lemma 5.3. *Let t be any integer such that at least $7/(\varepsilon\nu^2 d^2)$ of the first t steps are good. There exists $t' \leq t$ such that $\Phi_{t'} \leq \max\{\Phi_0/2, n^2\phi(24jd)\}$.*

Proof. If $\Phi_0 \leq n^2\phi(24jd)$, then the claim is proved for $t=0$. For the remainder of the proof, we assume that $\Phi_0 \geq n^2\phi(24jd)$. If $\Phi_{t'} \leq \Phi_0/2$ for any $t' < t$, the claim is proved. Otherwise, for all $t' < t$, we have $\Phi_{t'} > \Phi_0/2$. In this case, we obtain

$$\begin{aligned}\Phi_t &= \Phi_0 - \sum_{t'<t}\Psi_{t'}\\ &\leq 3\Phi_0 + n^2\phi(24jd) - \sum_{t'<t}\hat{\Psi}_{t'}\\ &\leq 4\Phi_0 - \sum_{\substack{t'<t\\ t'\text{good}}} (\varepsilon\nu^2 d^2)\Phi_{t'}\\ &\leq \Phi_0/2.\end{aligned}$$

(To obtain the second equation, we invoke Lemma 5.2. For the third equation, we invoke Lemma 5.1 and use the inequalities $\Phi_0 \geq n^2\phi(24jd)$ and $\hat{\Psi}_{t'} \geq 0$ for every t'. For the last equation, we use the fact that at least $7/(\varepsilon\nu^2 d^2)$ of the t steps are good and the equation $\Phi_{t'} > \Phi_0/2$ for every $t' < t$.) ∎

THEOREM 5.4. *For an arbitrary network G with n nodes, degree d, and initial imbalance Δ, if the live edges at every step t of G have edge expansion α, then the dynamic synchronous multiport algorithm load balances to within $O(d^2(\log n)/\alpha)$ tokens in $O(\Delta/\alpha)$ steps.*

Proof. We first place an upper bound on the number t of steps such that the height of each node at the end of step t is $O(d^2(\log n)/\alpha^2)$. If Δ is at most $d^2(\log n)/\alpha^2$, then a trivial bound is 0.

We now consider the case when Δ is at least $d^2(\log n)/\alpha^2$. By repeatedly invoking Lemma 5.3, we obtain that within $T = \lceil(7/(\varepsilon\nu^2 d^2))\rceil\lceil\log\Phi_0\rceil$ good steps, there exists a step after which the potential of the network is at most $n^2\phi(24jd)$. (Note that the fact that Lemma 5.2 holds for arbitrary initial values of the estimates, the $e^u(v)$'s, is crucial here.) Since at least $T/4$ of the first T steps are good, there exists $t \leq 4\lceil(7/(\varepsilon\nu^2 d^2))\rceil\lceil\log\Phi_0\rceil$ such that $\Phi_t \leq n^2\phi(24jd)$. Since $\Phi_0 \leq n(1+\nu)^{(\Delta+1)}/\nu$, we have $\log\Phi_0 \leq \log n + (\Delta+1)\log(1+\nu) - \log\nu$. Since $\nu = \alpha/(cd^2)$ and $\log(1+\nu) < \nu$, we have $t = O((\Delta/\alpha) + d^2(\log n)/\alpha^2) = O(\Delta/\alpha)$.

Let h be the maximum height of any node after step t. We thus have

$$\begin{aligned}\phi(h) &\leq \Phi_t\\ &\leq n^2(1+\nu)^{24jd}.\end{aligned}$$

Therefore, if $\phi(h) > 0$, then $h \leq \log_{(1+\nu)}(n^2(1+\nu)^{24jd})$. If $\phi(h) = 0$, then $h \leq 24jd - 11d$. In either case,

$$\begin{aligned}h &\leq 24jd + (2\log n)/\log(1+\nu)\\ &\leq 24jd + (4\log n)/\nu\\ &= O((d^2\log n)/\alpha).\end{aligned}$$

(The right-hand side of the first equation is an expansion of $\log_{(1+\nu)}(n^2(1+\nu)^{24jd})$. The second equation holds since $\log(1+\nu) < \nu/2$ for c appropriately large. The final inequality follows from the relations $\nu = \alpha/(cd^2)$ and $j = O((d\log n)/\alpha)$.)

Thus, at the end of step t, no node has more than $a = \rho + h$ tokens. We now prove by contradiction that for every step after step t, no node has more than $a + d$ tokens. Let t' be the first step after step t such that there exists some node u with more than $a + d$ tokens. (If no such t' exists, the claim holds trivially.) Of the $d+1$ highest tokens received by u after step t, at least 2 tokens (say p and q) were last sent by the same neighbor v of u. Without loss of generality, we assume that p arrived at u before q. Let t_1 be the step when p was last sent by v to u. Therefore, we have $e^v_{t_1}(u) \geq h_{t_1}(p) - d \geq a - d$. Hence q can be sent to u only when v has height at least $a + 11d$, which contradicts our choice of t'.

We have shown that after $O(\Delta/\alpha)$ steps, no node ever has more than $\rho + O((d^2\log n)/\alpha)$ tokens. An easy averaging argument shows that there exists $k = O((d\log n)/\alpha)$ such that after every step $t' \geq t$, $|S_{<-k}| \leq n/2$. By defining an appropriate potential function for tokens with heights below the average and repeating the analysis done for $S_{>j}$, we show that in another $O(\Delta/\alpha)$ steps, all nodes have more than $\rho - O(d^2(\log n)/\alpha)$ tokens. ∎

As suggested in [2], a simple variant of **DS** can be defined for asynchronous networks. As shown in [2], the analysis for the dynamic synchronous case can be used for asynchronous networks to yield the same time bounds. Hence, the multiport local load balancing algorithm balances to within $O(d^2 \log n/\alpha)$ tokens in time $O(\Delta/\alpha)$ on asynchronous networks.

6. Tight bounds on off-line load balancing. In this section, we analyze the load balancing problem in the off-line setting for both single-port and multiport models. We derive nearly tight bounds on the minimum number of steps required to balance on arbitrary networks in terms of the node and edge expansion of the networks. We assume that the network is synchronous.

We first consider the network $G = (V, E)$ under the single-port model. For any subset X of V, let $\overline{X}$ denote $V \setminus X$, $m(X)$ denote the number of edges in a maximum matching between X and $\overline{X}$, $A(X)$ denote the set $\{v \in \overline{X} : \exists x \in X \text{ such that } (x, v) \in E\}$, and $B(X)$ denote the set $\{x \in X : \exists y \in A(X) \text{ such that } (x, y) \in E\}$. For subsets X and Y of V, let $M(X, Y)$ denote the set of edges with one endpoint in X and the other in Y.

LEMMA 6.1. *For any network $G = (V, E)$ with node expansion μ and any subset X of V, we have $m(X) \le \mu \min\{|X|, |\overline{X}|\}/(1 + \mu)$. Moreover, for any subset X of V, $m(X \cup A(X)) \le |A(X)|$.*

Proof. Without loss of generality, assume that $|X| \le |\overline{X}|$. Consider the bipartite graph $H = (B(X), A(X), M(X, \overline{X}))$. A maximum matching in H is equal to a maximum flow in the graph $I = (B(X) \cup A(X) \cup \{s, t\}, M(X, \overline{X}) \cup \{(s, x) : x \in B(X)\} \cup \{(x, t) : x \in A(X)\})$ from source s to sink t. (All of the edges of I have unit capacity.) We will show that every cut C of I separating s and t is of cardinality at least $\mu|X|/(1 + \mu)$.

Consider any cut $C = (S, T)$ with $s \in S$ and $t \in T$. The set of edges in C is $M(S, T)$. Let $Y = T \cap B(X)$ and $Z = T \cap A(X)$. The capacity of C, given by $|M(S, T)|$, can be lower bounded as follows.

$$\begin{aligned}
|M(S,T)| &= |Y| + |M(Y, A(X) \setminus Z)| + |M(B(X) \setminus Y, Z)| + |A(X) \setminus Z| \\
&\ge |Y| + |M(B(X) \setminus Y, Z)| + |A(X) \setminus Z| \\
&\ge |A(X \setminus Y)| \\
&\ge \mu |X \setminus Y| \\
&= \mu(|X| - |Y|) \\
&\ge \mu|X|/(1 + \mu).
\end{aligned}$$

(For the third equation, see Figure 6.1. Three subsets of nodes contribute to the set $A(X \setminus Y)$: (i) the set of nodes in Y that have an edge to a node in $X \setminus Y$, (ii) the set of nodes in Z that have an edge to a node in $X \setminus Y$, and (iii) the set of nodes in $A(X) \setminus Z$ that have an edge to a node in $X \setminus Y$. The size of the three sets is bounded by $|Y|$, $|M(B(X) \setminus Y, Z)|$, and $|A(X) \setminus Z|$, respectively. The fourth equation follows from the definition of $A(X \setminus Y)$. The fifth equation holds since Y is a subset of X. The last equation holds since $|Y| \le |M(S, T)|$.)

For the second part of the lemma, we note that since all of the neighbors of X are in $A(X)$, any node in $X \cup A(X)$ that connects to some node outside of $X \cup A(X)$ is in $A(X)$. Therefore, $m(X \cup A(X)) \le |A(X)|$. $\square$

Theorem 1 of [29] obtains tight bounds on the off-line complexity of load balancing in terms of the function m. We restate the theorem using our notation and

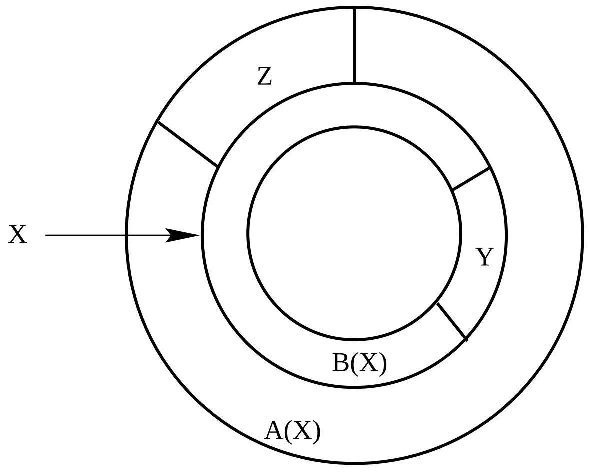


FIG. 6.1. *The sets X, Y, Z, $A(X)$, and $B(X)$ in the proof of Lemma* 6.1.

terminology. Before stating the theorem, we need one additional notation. For any subset X of nodes of any network, let $I(X)$ denote the number of tokens held by nodes in X in the initial distribution.

THEOREM 6.2 (see [29]). *Consider a network $G = (V, E)$ in the single-port model. The network G can be balanced in at most $\max_{\emptyset \subsetneq X \subsetneq V} \lceil (I(X) - \rho|X|)/m(X) \rceil$ steps so that every node has at most $\lceil \rho \rceil + 1$ tokens. Moreover, any algorithm takes at least $\max_{\emptyset \subsetneq X \subsetneq V} \lceil (I(X) - \rho|X|)/m(X) \rceil$ steps to balance the network so that every node has at most $\lceil \rho \rceil$ tokens.* $\square$

Theorem 6.2 and Lemma 6.1 imply the following result.

LEMMA 6.3. *Assume the single-port model. Any network G with node expansion μ and initial imbalance Δ can be balanced in at most $\lceil \Delta(1+\mu)/\mu \rceil$ steps so that every node has at most $\lceil \rho \rceil + 1$ tokens. Moreover, there exist a network G and an initial load distribution with imbalance Δ such that any algorithm takes at least $\lceil \Delta(1+\mu)/\mu \rceil$ steps to balance G such that every node has at most $\lceil \rho \rceil$ tokens.*

Proof. If $I(X)$ is the total number of tokens belonging to nodes in X in the initial distribution, then we have $-\Delta|X| \leq I(X) - \rho|X| \leq \Delta|X|$ for all X. Moreover, $|I(X) - \rho|X|| = |I(\overline{X}) - \rho|\overline{X}||$. Therefore, for all X, $|I(X) - \rho|X|| = \Delta \min\{|X|, |\overline{X}|\}$. By Lemma 6.1, $m(X)$ is at least $\mu \min\{|X|, |\overline{X}|\}/(1 + \mu)$ for all X. Thus, the first claim of Theorem 6.2 establishes the first claim of the desired lemma.

For the second claim of the lemma, given any μ, we construct the following network $G = (V, E)$ with node expansion μ. The node set V is partitioned into three sets X, Y, and Z such that (i) $|Y| = \mu|X|$ and (ii) $|Z| = |X|(1 + \mu)^2/(1 - \mu)$. Let n and x denote $|V|$ and $|X|$, respectively. Thus, n equals $x(1 + \mu + (1 + \mu)^2/(1 - \mu)) = 2x(1 + \mu)/(1 - \mu)$. The edge set E is the union of the sets $X \times X$, $X \times Y$, $Y \times Y$, $Y \times Z$, and $Z \times Z$.

We now show that the node expansion of G is μ. Consider any nonempty subset U of V of size at most $n/2$, and let X', Y', and Z' denote $U \cap X$, $U \cap Y$, and $U \cap Z$, respectively. Let $n(U)$ denote the number of neighbors of U that lie outside of U. We need to show that $n(U)$ is at least $\mu|U|$.

We consider two cases: (i) Y' and Z' are both empty, and (ii) Y' is nonempty or Z' is nonempty. In the first case, $U = X'$. Therefore, $n(U) \geq |Y| = \mu x \geq \mu|U|$. In the second case, we have

$$\begin{aligned} n(U) &\geq |Z| - |Z'| \\ &\geq |Z| - |U| \\ &\geq x((1+\mu)^2/(1-\mu) - (1+\mu)/(1-\mu)) \\ &= x\mu(1+\mu)/(1-\mu) \\ &\geq \mu|U|. \end{aligned}$$

(The second equation holds since Z' is a subset of U. For the third equation, note that $|U| \leq n/2 = x(1+\mu)/(1-\mu)$. The last equation follows from the upper bound of $x(1+\mu)/(1-\mu)$ on $|U|$.)

We now apply the second claim of Lemma 6.1 to the subset X. Since $A(X) = Y$, $m(X \cup Y) = \mu x = \mu|X \cup Y|/(1+\mu)$. Given any Δ, consider the initial token distribution in which each node in $X \cup Y$ has $\rho + \Delta$ tokens and each node in Z has $\rho - \Delta(1-\mu)/(1+\mu)$ tokens, where ρ is any integer that is at least $\Delta(1-\mu)/(1+\mu)$. (Note that the average number of tokens is ρ.) By applying the second claim of Theorem 6.2, we obtain that the number of steps to balance G so that each node has at most ρ tokens is at least $(I(X \cup Y) - \rho|X \cup Y|)/m(X \cup Y) \geq \Delta|X \cup Y|/m(X \cup Y) \geq \Delta(1+\mu)/\mu$. Since the number of steps is an integer, the desired claim follows. □

By using the techniques of [29], we can modify the proof of Lemma 6.3 to show that any network G with node expansion μ and initial imbalance Δ can be globally balanced to within 3 tokens in at most $2\lceil\Delta(1+\mu)/\mu\rceil$ steps. The extra factor of 2 is required because even after balancing the network so that each node has at most $\lceil\rho\rceil + 1$ tokens, there may exist a node with considerably fewer than ρ tokens. It takes an additional $\lceil\Delta(1+\mu)/\mu\rceil$ steps to bring the network to a state in which the global imbalance is at most 3.

Lemma 6.3 implies that the time bound achieved by the single-port algorithm (see Theorems 3.1 and 3.9) is not optimal for all networks. An example of a network for which the single-port algorithm is not optimal is the hypercube, which has maximum degree $\log n$, edge expansion 1, and node expansion $\Theta(1/\sqrt{\log n})$. The local algorithm balances in $\Omega(\Delta \log n)$ time, while there exists an $O(\Delta\sqrt{\log n} + \log^2 n)$ time load balancing algorithm for the hypercube [34] which is optimal for Δ sufficiently large. For the class of constant-degree networks, however, the time taken by the single-port algorithm to reduce the global imbalance to $O(\log n/\mu)$ (see Theorem 3.9) is within a constant factor of the time taken by any algorithm to completely balance the network (see Lemma 6.3).

The proofs of Theorem 1 of [29] and Lemma 6.3 can be modified to establish the following result for the multiport model.

LEMMA 6.4. *Assume the multiport model. Any network G with edge expansion α and initial imbalance Δ can be balanced in at most $\lceil\Delta/\alpha\rceil$ steps so that every node has at most $\lceil\rho\rceil + d$ tokens. Moreover, for every network G, there exists an initial load distribution with imbalance Δ such that any algorithm takes at least $\lceil\Delta/\alpha\rceil$ steps to balance G so that every node has at most $\lceil\rho\rceil$ tokens.*

Proof (sketch). We prove that there exists an off-line algorithm that balances to within d tokens in at most $T = \max_{\emptyset \subset X \subset V} \lceil \frac{|I(X) - \rho|X||}{|M(X,\overline{X})|} \rceil$ steps. For all $X \subseteq V$, we have (i) $|I(X) - \rho|X|| \leq \Delta \min\{|X|, |\overline{X}|\}$ (see proof of Lemma 6.3), and (ii) $|M(X,\overline{X})| \geq \alpha \min\{|X|, |\overline{X}|\}$. It follows from (i) and (ii) that $T \leq \lceil\Delta/\alpha\rceil$.

We modify the proofs of Theorem 1 and Lemma 4 of [29] (where the single-port model was assumed) to establish the desired claims for the multiport model. We transform the load balancing problem on G to a network flow problem on a directed graph $H = (V', E')$, which is constructed as follows. Let V_i be $\{\langle v, i\rangle : v \in V\}$, $0 \le i \le T$. Let E_i be $\{(\langle u, i\rangle, \langle v, i+1\rangle) : (u, v) \in E \text{ or } u = v\}$, $0 \le i < T$. We set V' to $\{s\} \cup \bigcup_{0 \le i \le T} V_i \cup \{t\}$ and E' to $\{(s, \langle v, 0\rangle) : v \in V\} \cup \bigcup_{0 \le i < T} E_i \cup \{(\langle v, T\rangle, t) : v \in V\}$. For any v in V, the capacity of the edge $(s, \langle v, 0\rangle)$ is $w(v)$. For any (u, v) in E, the capacity of any edge $(\langle u, i\rangle, \langle v, i+1\rangle)$, $0 \le i < T$, is 1. For any v in V, the capacity of any edge $(\langle v, i\rangle, \langle v, i+1\rangle)$, $0 \le i < T$, is ∞. For any v in V, the capacity of the edge $(\langle v, T\rangle, t)$ is $\lceil \rho \rceil + d$.

We show that the value of the maximum integral flow in H is equal to the total number of tokens N in V, from which it follows that there exists an off-line algorithm that balances to within d tokens in T steps. Consider any cut $C = (S, T)$ of H separating $s \in S$ and $t \in T$. Let $S_i = S \cap V_i$ and $D(S_i) = \{v \in V : \langle v, i\rangle \in S_i\}$. If $S_0 = \emptyset$, or $S_T = V_T$, or there is an edge of infinite capacity, then the capacity of C is at least N. Otherwise, the number of edges from V_i to V_{i+1} that belong to the cut is at least $|M(D(S_i), \overline{D(S_i)})| - d(|S_{i+1}| - |S_i|)$. Moreover, since there is no edge with infinite capacity in C, $D(S_i)$ is a subset of $D(S_{i+1})$. Thus the capacity of C is at least

$$\begin{aligned}
&I(D(V_0) \setminus D(S_0)) + \left(\sum_{i=0}^{T-1} \Big(|M(D(S_i), \overline{D(S_i)})| - d(|S_{i+1}| - |S_i|)\Big)\right) + (\lceil \rho \rceil + d)|S_T| \\
&\ge I(D(V_0) \setminus D(S_0)) + \left(\sum_{i=0}^{T-1} ((I(D(S_i)) - \rho|S_i|)/T - d(|S_{i+1}| - |S_i|))\right) + (\lceil \rho \rceil + d)|S_T| \\
&\ge I(D(V_0) \setminus D(S_0)) + \left(\sum_{i=0}^{T-1} ((I(D(S_0)) - \rho|S_T|)/T - d(|S_T| - |S_0|))\right) + (\lceil \rho \rceil + d)|S_T| \\
&\ge I(D(V_0) \setminus D(S_0)) + I(D(S_0)) - \rho|S_T| + d|S_0| + \lceil \rho \rceil |S_T| \\
&\ge N.
\end{aligned}$$

(In the first equation, (i) $I(D(V_0) \setminus D(S_0))$ is the capacity of the edges from s to V_0 that belong to the cut, (ii) $|M(D(S_i), \overline{D(S_i)})| - d(|S_{i+1}| - |S_i|)$ is the capacity of the edges from V_i to V_{i+1} that belong to the cut, and (iii) $(\lceil \rho \rceil + d)|S_T|$ is the capacity of the edges from S_T to t that belong to the cut. The second equation follows from the definition of T and the fact that $|D(S_i)| = |S_i|$. For the third equation, note that $D(S_0)$ is a subset of $D(S_i)$ for all i and $|S_T| \ge |S_i|$ for all i. The fourth equation holds since the sum of $|S_{i+1}| - |S_i|$ telescopes. The final equation is obtained since $I(D(V_0)) = N$.) Since the capacity of the cut $(\{s\}, V' \setminus \{s\})$ equals N, the maximum flow in H is N.

To prove the second part of the lemma, given any network G with a partition (V_1, V_2) of its nodes such that $|V_1| \le n/2$ and $|M(V_1, V_2)| = \alpha|V_1|$, we define an initial load distribution with average ρ in which each node in V_1 has $\rho + \Delta$ tokens and each node in V_2 has $\rho - \Delta|V_1|/|V_2|$ tokens. The desired claim holds since at least $\Delta|V_1|$ tokens need to leave the set V_1. $\square$

Lemma 6.4 implies that the local multiport algorithm is asymptotically optimal for *all* networks. As in the single-port case, we can modify the above proof to obtain upper bounds on the off-line complexity of globally balancing a network. We can show that any network G with edge expansion α and initial imbalance Δ can be globally balanced to within $d + 1$ tokens in at most $2\lceil \Delta/\alpha \rceil$ steps.

Appendix A. Some technical inequalities. Let ν equal $\alpha/(cd^2)$. For the following we set c large enough so that $(1 + \nu)^{12d} \le 3/2$. The function ϕ is defined in section 5.

LEMMA A.1. *For any integer x, if $\phi(x) > 0$, then $\phi(x+12d) \le 3\phi(x)/2$.*

Proof. Since $\phi(x) > 0$, we have $\phi(x+12d) = (1+\nu)^{12d}\phi(x) \le 3\phi(x)/2$. (Note that if $\phi(x) = 0$, then $\phi(x+12d)$ may not equal $(1+\nu)^{12d}\phi(x)$.) ∎

LEMMA A.2. *For any integer x we have*

$$\max\{\phi(24jd), \phi(x-12d)\} \ge 2\phi(x)/3.$$

Proof. If $\phi(x-12d) > 0$, then $2\phi(x)/3 \le \phi(x-12d)$ by Lemma A.1. Otherwise, $x - 12d \le 24jd - 11d$, which implies that $x \le 24jd + d$. Therefore, $\phi(x) \le \phi(24jd+d) \le \phi(24jd)(1+\nu)^d \le 3\phi(24jd)/2$. ∎

LEMMA A.3. *For any integers x and y, if $\phi(x) > 0$ and $x - y \ge 11d$, then we have $\phi(x) - \phi(y) \ge 2(\phi(x+11d) - \phi(y))/5$.*

Proof.

$$\begin{aligned}
2(\phi(x+11d) - \phi(y))/5 &= 2(\phi(x+11d) - \phi(x))/5 \\
&\quad +2(\phi(x) - \phi(y))/5 \\
&\le 2(1+\nu)^{11d}(\phi(x) - \phi(x-11d))/5 \\
&\quad +2(\phi(x) - \phi(y))/5 \\
&\le 2(1+\nu)^{11d}(\phi(x) - \phi(y))/5 \\
&\quad +2(\phi(x) - \phi(y))/5 \\
&\le \phi(x) - \phi(y).
\end{aligned}$$

(In the second equation we use $x - 11d \ge y$. In the last equation we use $(1+\nu)^{11d} \le 3/2$.) ∎

Appendix B. Proof of Lemma 5.2. We define a notion of "goodness" of the tokens. Initially, all tokens are unmarked. After any step t, for every token p that is moved along an edge, p is marked *good* if $h_{t-1}(p) - h_t(p) \ge 6d$; otherwise, p is marked *bad*. The marking of tokens that do not move is unchanged.

LEMMA B.1. *For any two bad tokens p_1 and p_2 present at any node v at the start of any step t, if p_1 and p_2 are last sent to v by the same neighbor u of v, then $|h_t(p_1) - h_t(p_2)| > 4d$.*

Proof. Let t_1 (resp., t_2) be the step during which p_1 (resp., p_2) is last sent to v. Without loss of generality, we assume $t_1 < t_2 < t$. Thus we have $h_t(p_1) < h_t(p_2)$. Since u's estimate of the number of tokens at v is updated in step t_1, we have $e^u_{t_1}(v) \ge \rho + h_{t_1}(p_1) - d$. (Note that $e^u_{t_1}(v)$ is u's estimate of the number of tokens at v after step t_1.) Since p_1 remains at v during the interval $[t_1, t_2)$, we find that $e^u_{t'}(v) \ge \rho + h_{t'}(p_1) - d$ for every step t' in $[t_1, t_2)$. In particular, we have $e^u_{t_2-1}(v) \ge \rho + h_{t_2-1}(p_1) - d$. Since u sends p_2 to v in step t_2, $h_{t_2-1}(p_2) \ge h_{t_2-1}(u) - d \ge e^u_{t_2-1}(v) - \rho + 11d \ge h_{t_2-1}(p_1) + 10d$. Since p_2 is bad, we also have $h_{t_2}(p_2) > h_{t_2-1}(p_2) - 6d \ge h_{t_2-1}(p_1) + 4d$. Since $h_t(p_2) = h_{t_2}(p_2)$ and $h_t(p_1) = h_{t_2-1}(p_1)$, the lemma follows. ∎

COROLLARY B.2. *At any time, for any node u and integer $i > 0$, there are at most d bad tokens with heights in $(i, i+4d]$.* ∎

Proof of Lemma 5.2. Consider an arbitrary step t of the algorithm. For every token p transferred from u to v in step t, we assign some credit to every edge adjacent to u or v. Specifically, if p is marked good after step t we assign an *outgoing credit* of $9(\phi(h_{t-1}(p)) - \phi(h_t(p)))/(20d)$ units to every edge adjacent to u and an *incoming credit* of the same amount to every edge adjacent to v. If p is marked bad we assign an outgoing credit of $(\phi(h_t(p)+d) - \phi(h_t(p)))/(20d) + (\phi(h_{t-1}(p)) - \phi(h_{t-1}(p)-d))$

units to every edge adjacent to u and an incoming credit of the same amount to every edge adjacent to v. Also, for each edge (u, v), we assign an *initial credit* of $2\max\{\phi(24jd), \phi(h_0(u) - d) + \phi(h_0(v) - d)\}$ units at the start of the analysis. The total initial credit I is bounded as follows:

$$\begin{aligned} I &\le 2\binom{n}{2}\phi(24jd) + \sum_{(u,v)\in E} 2(\phi(h_0(u) - d) + \phi(h_0(v) - d)) \\ &\le n^2\phi(24jd) + \sum_{u\in V}\sum_{0\le \ell<d} 2\phi(h_0(u) - \ell) \\ &\le n^2\phi(24jd) + 2\Phi_0. \end{aligned}$$

(The first equation follows from the fact that the maximum of two quantities is at most the sum of the particular quantities. We also note that each undirected edge (u, v) appears at most once in the summation. For the second equation, we note that each node has at most d edges. Hence for any node u, the term $2\phi(h_0(u)-d)$ appears in at most d terms of the sum. We complete the derivation of the second equation by observing that $\phi(h_0(u) - \ell)$ is at least $\phi(h_0(u) - d)$ for $0 \le \ell < d$. The third equation is obtained by the fact that $\sum_{0\le\ell<d} \phi(h_0(u) - \ell)$ is at most $\phi(u)$.) The above bound on I corresponds to the negative term in (5.1).

We now show that by using the above accounting method we can account for the amortized potential drop of $(\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d))/50$ units at step t for every edge $(u, v) \in E_t$. To accomplish this, for every live edge (u, v) ((u, v) not necessarily in E_t), we consider three cases: (i) a token p sent from u to v is marked good, (ii) a token p sent from u to v is marked bad, (iii) no token is sent from u to v.

We first consider case (i). When a token p is marked good after being sent along (u, v), we use the actual potential drop of p to pay for the amortized drop D_1 associated with (u, v) as well as the total credit D_2 assigned to the edges adjacent to u or v due to the transfer of a good token.

$$\begin{aligned} D_1 + D_2 &\le (\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d))/50 + 2d[9(\phi(h_{t-1}(p)) - \phi(h_t(p)))]/(20d) \\ &\le (\phi(h_{t-1}(p)) - \phi(h_t(p)))/50 + 9(\phi(h_{t-1}(p)) - \phi(h_t(p)))/10 \\ &\le \phi(h_{t-1}(p)) - \phi(h_t(p)). \end{aligned}$$

(The first term in the right-hand side of the first equation is the amortized potential drop. The second term is an upper bound on D_2, since the number of edges adjacent to either u or v is at most $2d$. The second equation follows from the fact that $h_{t-1}(p)$ is at least $h_{t-1}(u) - d$ and $h_t(p)$ is at most $h_t(u) + d$.)

We now consider case (ii). In this case we need to account for (1) if $h_t(p) > h_{t-1}(p)$, an amount equal to the potential increase of $D_1 = \phi(h_t(p))-\phi(h_{t-1}(p))$ units, and (2) a credit of at most $(\phi(h_t(p) + d) - \phi(h_t(p)))/10 + (\phi(h_{t-1}(p)) - \phi(h_{t-1}(p) - d))/10$ units. We pay for $(\phi(h_{t-1}(p)) - \phi(h_t(p)))/10$ units of the credit using the potential change. The remainder of the credit we need to account for is at most the sum of $D_2 = (\phi(h_t(p) + d) - \phi(h_t(p)))/10$ and $D_3 = (\phi(h_t(p)) - \phi(h_{t-1}(p) - d))/10$. (Note that this is true regardless of whether the potential of p decreases in step t.)

We have two subcases, depending on whether t is the first step in which (u, v) is live (subcase (a)) or not (subcase (b)). In subcase (a), if $h_0(u) \ge h_t(p) - d$, the initial credit C_0 associated with (u, v) is at least $2\max\{\phi(24jd), \phi(h_t(p) - 2d)\}$. Since $\phi(h_t(p) - 2d) \ge \phi(h_t(p) - 12d)$, it follows from Lemma A.2 that $3C_0/4 \ge \phi(h_t(p)) \ge D_1$. Since $\phi(h_t(p) - 2d) \ge \phi(h_t(p) - 11d)$, $C_0/4 \ge \phi(h_t(p) + d)/3 \ge \phi(h_t(p) + d)/10 + \phi(h_t(p))/10 \ge D_2 + D_3$. Therefore, we have $C_0 \ge D_1 + D_2 + D_3$.

We now consider subcase (a) under the assumption that $h_0(v) \le h_t(p) - d$. In order to do the accounting, we use part of the incoming credit associated with the edge (u, v) due to the set X of good tokens of v with heights in the interval $(h_0(v), h_t(p) - d]$. (Note that each token in X is marked and thus has contributed incoming credit to all edges adjacent to v.) Since each token x in X is good, the height of the token before the transfer to node v was at least $h_t(q) + 6d$. Therefore, the incoming credit assigned to (u, v) by a token x in X is at least $9(\phi(h_t(q) + 6d) - \phi(h_t(q)))/(20d)$ units. For each token x in X, we use $c_x = 8(\phi(h_t(q) + 6d) - \phi(h_t(q)))/(20d)$ units of this incoming credit. Let C_1 denote $\sum_{x \in X} c_x$. We obtain the following lower bound C_1. By invoking Corollary B.2, we obtain

$$\begin{aligned}
C_1 &\ge \frac{8}{20d} \sum_{1 \le i \le \lfloor \frac{h_t(p) - d - h_0(v)}{4d} \rfloor} \sum_{1 \le k \le 3d} (\phi(h_t(p) - d - 4id + k + 6d) \\
&\qquad\qquad - \phi(h_t(p) - d - 4id + k + d)) \\
&\ge \frac{8}{20d} \sum_{1 \le k \le 3d} \sum_{1 \le i \le \lfloor \frac{h_t(p) - d - h_0(v)}{4d} \rfloor} (\phi(h_t(p) - d - 4id + k + 6d) \\
&\qquad\qquad - \phi(h_t(p) - 4id + k)) \\
&\ge \frac{8}{20d} \sum_{1 \le k \le 3d} \Bigg(\phi(h_t(p) - d - 4d + k + 6d) \\
&\qquad\qquad - \phi(h_t(p) - 4d \left\lfloor \frac{h_t(p) - d - h_0(v)}{4d} \right\rfloor + k) \Bigg) \\
&\ge \frac{8}{20d} \sum_{1 \le k \le 3d} (\phi(h_t(p) + d) - \phi(h_0(v) + 8d)) \\
&= 6(\phi(h_t(p) + d) - \phi(h_0(v) + 8d))/5.
\end{aligned}$$

(In the first equation we partition the interval $(h_0(v), h_t(p) - d]$ into subintervals of $4d$ consecutive integers starting from $h_t(p) - d$. The last subinterval may have fewer than $4d$ integers; if so, we ignore the last subinterval in the sum. The second summation in the first equation is a lower bound on the sum of c_x over each good token x in each subinterval. To obtain the second summation, we invoke Corollary B.2, which implies that there are at least $3d$ good tokens in every subinterval of $4d$ tokens. The second equation is obtained by interchanging the order of summation. For the third equation, we use the fact that $\phi(h_t(p) - d - 4(i-1)d + k + 6d) \ge \phi(h_t(p) - 4id + k)$ and then note that the sum telescopes. For the fourth inequality, note that (i) the index k is at least 0 and at most $3d$, and (ii) $h_t(p) - 4d\lfloor \frac{h_t(p) - d - h_0(v)}{4d} \rfloor \le h_0(v) + 5d$.)

Since p is marked bad after step t, we have $h_t(p) > h_{t-1}(p) - 6d$. Therefore,

$$\begin{aligned}
C_0 + C_1 &\ge 2 \max\{\phi(24jd), \phi(h_0(v) - d)\} + 6(\phi(h_t(p) + d) - \phi(h_0(v) + 8d))/5 \\
&\ge 6\phi(h_t(p) + d)/5 \\
&\ge \phi(h_t(p)) - \phi(h_{t-1}(p)) + (\phi(h_t(p) + d) - \phi(h_t(p)))/10 \\
&\quad + (\phi(h_t(p)) - \phi(h_{t-1}(p) - d))/10 \\
&\ge D_1 + D_2 + D_3.
\end{aligned}$$

(The first equation states the lower bounds on C_0 and C_1 obtained above. For the second equation, we invoke Lemma A.2 as follows: $2 \max\{\phi(24jd), \phi(h_0(v) - d)\} \ge 4\phi(h_0(v) + 11d)/3 \ge 6\phi(h_0(v) + 8d)/5$. The third equation is obtained from the following three observations: (i) $\phi(h_t(p)+d) \geq \phi(h_t(p)) - \phi(h_{t-1}(p))$, (ii) $\phi(h_t(p)+d)/10 \geq (\phi(h_t(p)+d) - \phi(h_t(p)))/10$, and (iii) $\phi(h_t(p)+d)/10 \geq (\phi(h_t(p)) - \phi(h_{t-1}(p) - d))/10$.)

We use a similar argument as above to handle subcase (b), where t is not the first step in which (u, v) is live. The set X is the set of good tokens of v with heights in the interval $(e^u_{t-1}(v) - \rho, h_t(p) - d]$. Let c_x and C_1 be defined as in subcase (a). That is, c_x equals $8(\phi(h_t(x)+6d) - \phi(h_t(x)))/(20d)$ units of the incoming credit assigned to (u, v) by a token x in X, and C_1 equals $\sum_{x \in X} c_x$. We will show that $11C_1/12 \geq D_1 + D_3$ and $C_1/12 \geq D_2$, and hence obtain that $C_1 \geq D_1 + D_2 + D_3$.

We first show that $11C_1/12 \geq D_1 + D_3$. If $h_t(p) \leq h_{t-1}(p) - d$, then D_1 and D_3 are both nonpositive and hence the desired claim holds trivially. We now assume that $h_t(p) > h_{t-1}(p) - d$. Let y denote $e^u_{t-1}(v) - \rho + 8d$. We observe that since u sent a token to v during step t, $y = e^u_{t-1}(v) - \rho + 8d \leq h_{t-1}(u) - 4d \leq h_{t-1}(p) - 3d$. Since p is a bad token, we have $y \leq h_{t-1}(p) - 3d < h_t(p) - 2d$. As in subcase (a), we divide the interval $(e^u_{t-1}(v) - \rho, h_t(p) - d]$ into subintervals consisting of $4d$ consecutive integers. Note that $e^u_{t-1}(v) - \rho \leq h_t(p) - 11d$ and hence the number of subintervals is at least 1. We obtain the following lower bound on $11C_1/12$.

$$\begin{aligned}
11C_1/12 &\geq (11/12) \cdot 6(\phi(h_t(p)+d) - \phi(y))/5 \\
&\geq 11(\phi(h_t(p)+d) - \phi(h_{t-1}(p) - 2d))/10 \\
&\geq (\phi(h_t(p)) - \phi(h_{t-1}(p))) + (\phi(h_t(p)) - \phi(h_{t-1}(p) - d))/10 \\
&= D_1 + D_3.
\end{aligned}$$

(The first equation is obtained in the same manner as the upper bound on C_1 in subcase (a). While the interval considered in subcase (a) is $(h_0(v), h_t(p) - d]$, we consider here the interval $(e^u_{t-1}(v) - \rho, h_t(p) - d] = [y - 8d, h_t(p) - d]$. Hence, the term $\phi(h_0(v) + 8d)$ obtained in the lower bound on C_1 in subcase (a) is replaced by $\phi(y)$ above. The second equation is obtained from the upper bound on y.)

We now show that $C_1/12 \geq D_2$. Since a token is sent by u to v in step t, $e^u_{t-1}(u) - \rho \leq h_{t-1}(u) - 12d \leq h_{t-1}(p) - 11d$. Moreover, since p is a bad token, $h_{t-1}(p) \leq h_t(p) - 6d$. Therefore, $e^u_{t-1}(u) - \rho \leq h_t(p) - 5d$. It follows that $(h_t(p) - 5d, h_t(p) - d]$ is a subinterval of $(e^u_{t-1}(u) - \rho, h_t(p) - d]$. Hence, C_1 can be lower bounded by adding c_x over all good tokens x whose height is in $(h_t(p) - 5d, h_t(p) - d]$. By Corollary B.2, at least $3d$ of the tokens in $[h_t(p) - 5d, h_t(p) - d]$ are good. We thus obtain

$$\begin{aligned}
C_1/12 &\geq (3d/12) \cdot 8(\phi(h_t(p)+d) - \phi(h_t(p) - d))/(20d) \\
&= (\phi(h_t(p)+d) - \phi(h_t(p)))/10 \\
&\geq D_2.
\end{aligned}$$

(For the first equation, note that $c_x = 8(\phi(h_t(x)+6d) - \phi(h_t(x)))/(20d) \geq 8(\phi(h_t(p)+d) - \phi(h_t(p) - d))/(20d)$ for $h_t(x)$ in $[h_t(p) - 5d, h_t(p) - d]$. The last equation follows from the definition of D_2.)

To complete the proof for case (ii), we show that for any token x of v, any incoming credit assigned by x to edge (u, v) that is used at step t for case (ii) is not used again for case (ii). To prove this, we note that for any x in X, for every further step $t' > t$ until x is transferred by u, we have $h_{t'}(x) \geq e^u_{t'-1}(v) - \rho$. While establishing case (ii) for step t, we use only the incoming credit assigned by tokens in $(e^u_{t'-1}(v) - \rho, h_{t'}(p) - d]$. Hence the incoming credit assigned by x to edges adjacent to u that is used at step t will never be used again.

We need to consider case (iii) only under the assumption that $(u, v) \in E_t$, i.e., (u, v) is live in step t. In this case we account for $D = (\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d))/50$ units of potential. Again we consider two subcases depending on whether t is the last step in which (u, v) is live (subcase (a)) or not (subcase (b)). We first consider subcase (a). If $h_0(u) \geq h_{t-1}(u) - 12d$, then we use $C_0 = 2\max\{\phi(24jd), \phi(h_0(u) - d)\}$ units of the initial credit associated with (u, v). Since $h_{t-1}(u) - d \leq h_0(u) - d + 12d$, it follows from Lemma A.2 that $C_0 \geq 4\phi(h_{t-1}(u) - d)/3 \geq \phi(h_{t-1}(u) - d)/50 \geq D$.

We now consider subcase (a) of case (iii) under the assumption that $h_0(u) < h_{t-1}(u) - 12d$. In addition to C_0, we also use part of the incoming credit associated with the set of tokens $Y = \{y : y \text{ is a token of } u \text{ and } h_0(u) < h_t(y) \leq h_{t-1}(u)\}$. Specifically, for every token y in Y, we use $(\phi(h_t(y) + d) - \phi(h_t(y)))/(20d)$ units of incoming credit that is assigned to (u, v) by y. Note that since $h_t(y) > h_0(u)$, token y has moved and hence has assigned some incoming credit to (u, v). If y is good, this credit is at least $9(\phi(h_t(y) + 6d) - \phi(h_t(y)))/(20d)$ units; otherwise, this credit is at least $(\phi(h_t(y) + d) - \phi(h_t(y)))/(20d)$. Moreover, if y is a good token, at most $8(\phi(h_t(y) + 6d) - \phi(h_t(y)))/(20d)$ units of incoming credit were used in the analysis of case (ii). If y is a bad token, none of the incoming credit was used in the analysis of case (ii). In either case, at least $(\phi(h_t(y) + d) - \phi(h_t(y)))/(20d)$ units of incoming credit still remain. Let this credit be denoted C_1. We obtain the following lower bound on $C_0 + C_1$:

$$\begin{aligned}
C_0 + C_1 &\geq C_0 + \sum_{h_0(u) < k \leq h_{t-1}(u)} (\phi(k + d) - \phi(k))/(20d) \\
&= C_0 + \frac{1}{20d} \sum_{1 \leq i \leq d} (\phi(h_{t-1}(u) + i) - \phi(h_0(u) + i)) \\
&\geq C_0 + (\phi(h_{t-1}(u)) - \phi(h_0(u) + d))/20 \\
&\geq \phi(h_{t-1}(u))/20 \\
&\geq D.
\end{aligned}$$

(The second equation holds since the sum in the first equation can be expressed as a sum of d telescoping sums. For the third equation we invoke Lemma A.2 and obtain that $C_0 \geq 2\phi(h_0(u) + 11d)/3 \geq \phi(h_0(u) + d)/20$.)

We now consider subcase (b) of (iii). Recall that by the definition of E_t, u is in $S_{>j}$ at the start of step t. Therefore, $h_{t-1}(u) \geq 24(j+1)d - 12d \geq 24jd + 12d$. Since no token was sent along (u, v) in step t, we have $e^u_{t-1}(v) - \rho > h_{t-1}(u) - 12d$ ($\geq 24jd$). By the definition of E_t, we also have $h_{t-1}(u) \geq h_{t-1}(v) + 24d$. It follows that $e^u_{t-1}(v) - \rho > h_{t-1}(v) + 12d$. Since the last step in which (u, v) was live, at least $e^u_{t-1}(v) - \rho - h_{t-1}(v)$ tokens have left v. We use the outgoing credit assigned to (u, v) due to these token transmissions. Consider a token x that is transmitted by v in step t'. If x is marked good after the step, then the outgoing credit assigned by x to (u, v) is at least $9(\phi(h_{t'-1}(p)) - \phi(h_{t'}(p)))/(20d) \geq 9(\phi(h_{t'-1}(p)) - \phi(h_{t'-1}(p) - 6d))/(20d)$ units. Otherwise, the outgoing credit assigned by x to (u, v) is at least $(\phi(h_{t'-1}(p)) - \phi(h_{t'-1}(p) - d))/(20d)$ units. In either case, the outgoing credit is at least $(\phi(h_{t'-1}(p)) - \phi(h_{t'-1}(p) - d))/(20d)$ units. We thus obtain the following lower bound on the total outgoing credit C_2 assigned to (u, v) by at least $e^u_{t-1}(v) - \rho - h_{t-1}(v)$ tokens.

$$C_2 \geq \sum_{h_{t-1}(v) < k \leq e^u_{t-1}(v) - \rho} (\phi(k) - \phi(k - d))/(20d)$$

$$\begin{aligned}
&= \frac{1}{20d} \sum_{1 \le i \le d} (\phi(e^u_{t-1}(v) - \rho - d + i) - \phi(h_{t-1}(v) - d + i)) \\
&\ge (\phi(e^u_{t-1}(v) - \rho - d) - \phi(h_{t-1}(v)))/20 \\
&\ge (\phi(e^u_{t-1}(v) - \rho + 11d) - \phi(h_{t-1}(v) + d))/50 \\
&\ge (\phi(h_{t-1}(u) - d) - \phi(h_{t-1}(v) + d))/50 \\
&= D.
\end{aligned}$$

(The second equation holds since the sum in the first equation can be expressed as a sum of d telescoping sums. For the third and fourth inequalities, we first note that since no token is sent by u to v in step t, we have $e^u_{t-1}(v) - \rho > h_{t-1}(u) - 12d \ge 24jd - d$. The third equation now follows from Lemma A.3 and the fact that $\phi(e^u_{t-1}(v) - \rho - d) > 0$. The fourth equation follows directly from the lower bound on $e^u_{t-1}(v) - \rho$.)

We note that the outgoing credit assigned to edge (u, v) in the above analysis of case (iii) is used at most once in case (iii). To prove this, we observe that after step t, the value of $e^u(v)$ is updated by u to $h_{t-1}(v) + \rho$. Therefore, if case (iii) of the analysis subsequently uses any outgoing credit assigned by a token x that leaves v and whose height in v is in $(h_{t-1}(v), e^u_{t-1}(v)]$, then x reached v after step t. Hence, the outgoing credit assigned by the $e^u_{t-1}(v) - h_{t-1}(v)$ tokens that are used in the analysis for step t are not used again for a later step. $\square$

REFERENCES

[1] Y. Afek, E. Gafni, and A. Rosen, *The slide mechanism with applications in dynamic networks*, in Proceedings of the 11th ACM Symposium on Principles of Distributed Computing, 1992, pp. 35–46.

[2] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao, *Approximate load balancing on dynamic and asynchronous networks*, in Proceedings of the 25th Annual ACM Symposium on the Theory of Computing, 1993, pp. 632–641.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, *Sorting in $c \log n$ parallel steps*, Combinatorica, 3 (1983), pp. 1–19.

[4] J. Aspnes, M. Herlihy, and N. Shavit, *Counting networks and multiprocessor co-ordination*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 348–358.

[5] B. Awerbuch and T. Leighton, *A simple local-control approximation algorithm for multicommodity flow*, in Proceedings of the 34th IEEE Annual Symposium on Foundations of Computer Science, 1993, pp. 459–468.

[6] B. Awerbuch and T. Leighton, *Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks*, in Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, 1994, pp. 487–496.

[7] B. Awerbuch, Y. Mansour, and N. Shavit, *End-to-end communication with polynomial overhead*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 358–363.

[8] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice–Hall, Englewood Cliffs, NJ, 1989.

[9] A. Broder, A. M. Frieze, E. Shamir, and E. Upfal, *Near-perfect token distribution*, in Proceedings of the 19th International Colloquium on Automata, Languages and Programming, 1992, pp. 308–317.

[10] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Stat., 23 (1952), pp. 493–507.

[11] Y. C. Chow and W. Kohler, *Models for dynamic load balancing in a heterogeneous multiple processor system*, IEEE Trans. Comput., C–28 (1980), pp. 57–68.

[12] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[13] D. Eager, E. Lazowska, and J. Zahorjan, *Adaptive load sharing in homogeneous distributed systems*, IEEE Trans. Software Engrg., SE–12 (1986), pp. 662–675.

[14] B. Ghosh and S. Muthukrishnan, *Dynamic load balancing on parallel and distributed networks by random matchings*, in Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, 1994, pp. 226–235.

[15] B. Ghosh, S. Muthukrishnan, and M. H. Schultz, *First and second order diffusive methods for rapid, coarse, distributed load balancing*, in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 72–81.

[16] B. Goldberg and P. Hudak, *Implementing functional programs on a hypercube multiprocessor*, in Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications, vol. 1, 1989, pp. 489–503.

[17] A. Heirich and S. Taylor, *A Parabolic Theory of Load Balance*, Research Report Caltech-CS-TR-93-25, Caltech Scalable Concurrent Computation Lab, Pasadena, CA, 1993.

[18] K. T. Herley, *A note on the token distribution problem*, Inform. Process. Lett., 28 (1991), pp. 329–334.

[19] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, *Analysis of a graph coloring based distributed load balancing algorithm*, J. Parallel Distrib. Comput., 10 (1990), pp. 160–166.

[20] J. JáJá and K. W. Ryu, *Load balancing and routing on the hypercube and related networks*, J. Parallel Distrib. Comput., 14 (1992), pp. 431–435.

[21] M. R. Jerrum and A. Sinclair, *Conductance and the rapid mixing property for Markov chains: The approximation of the permanent resolved*, in Proceedings of the 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 235–244.

[22] R. Karp and Y. Zhang, *A randomized parallel branch-and-bound procedure*, J. ACM, 40 (1993), pp. 765–789.

[23] M. Klugerman and C. G. Plaxton, *Small depth counting networks*, in Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 417–428.

[24] T. Leighton, C. E. Leiserson, and D. Kravets, *Theory of Parallel and VLSI Computation*, Research Seminar Series Report MIT/LCS/RSS 8, MIT Laboratory for Computer Science, MIT, Cambridge, MA, 1990.

[25] F. C. H. Lin and R. M. Keller, *The gradient model load balancing method*, IEEE Trans. Software Engrg., SE–13 (1987), pp. 32–38.

[26] R. Lüling and B. Monien, *Load balancing for distributed branch and bound algorithms*, in Proceedings of the 6th International Parallel Processing Symposium, 1992, pp. 543–549.

[27] R. Lüling and B. Monien, *A dynamic distributed load balancing algorithm with provable good performance*, in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 164–172.

[28] R. Lüling, B. Monien, and F. Ramme, *Load balancing in large networks: A comparative study*, in Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, IEEE Press, Piscataway, NJ, 1991, pp. 686–689.

[29] F. Meyer auf der Heide, B. Oesterdiekhoff, and R. Wanka, *Strongly adaptive token distribution*, in Proceedings of the 20th International Colloquium on Automata, Languages and Programming, 1993, pp. 398–409.

[30] M. Mihail, *Conductance and convergence of Markov chains—a combinatorial treatment of expanders*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 526–531.

[31] L. M. Ni, C. Xu, and T. B. Gendreau, *Distributed drafting algorithm for load balancing*, IEEE Trans. Software Engrg., SE–11 (1985), pp. 1153–1161.

[32] D. Peleg and E. Upfal, *The generalized packet routing problem*, Theoret. Comput. Sci., 53 (1987), pp. 281–293.

[33] D. Peleg and E. Upfal, *The token distribution problem*, SIAM J. Comput., 18 (1989), pp. 229–243.

[34] C. G. Plaxton, *Load balancing, selection and sorting on the hypercube*, in Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 64–73.

[35] J. Stankovic, *Simulations of three adaptive, decentralized controlled, job scheduling algorithms*, Computer Networks, 8 (1984), pp. 199–217.

[36] R. Subramanian and I. D. Scherson, *An analysis of diffusive load balancing*, in Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, 1994, pp. 220–225.

[37] A. N. Tantawi and D. Towsley, *Optimal static load balancing in distributed computer systems*, J. ACM, 32 (1985), pp. 445–465.

[38] E. Upfal, *An $O(\log N)$ deterministic packet routing scheme*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 241–250.

[39] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3 (1991), pp. 457–481.

[40] C. Z. Xu and F. C. M. Lau, *Iterative dynamic load balancing in multicomputers*, J. Oper. Res. Soc., 45 (1994), pp. 786–796.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 65–80

AN ALGORITHM FOR SHORTEST PATHS IN BIPARTITE DIGRAPHS WITH CONCAVE WEIGHT MATRICES AND ITS APPLICATIONS*

XIN HE† AND ZHI-ZHONG CHEN‡

Abstract. The traveling salesman problem on an n-point convex polygon and the minimum latency tour problem for n points on a straight line are two basic problems in graph theory and have been studied in the past. Previously, it was known that both problems can be solved in $O(n^2)$ time. However, whether they can be solved in $o(n^2)$ time was left open by Marcotte and Suri [*SIAM J. Comput.*, 20 (1991), pp. 405–422] and Afrati et al. [*Informatique Theorique Appl.*, 20 (1986), pp. 79–87], respectively.

In this paper we show that both problems can be solved in $O(n \log n)$ time by reducing them to the following problem: Given an edge-weighted complete bipartite digraph $G = (X, Y, E)$ with $X = \{x_0, \ldots, x_n\}$ and $Y = \{y_0, \ldots, y_m\}$, we wish to find the shortest path from x_0 to x_n in G. This new problem requires $\Omega(nm)$ time to solve in general, but we show that it can be solved in $O(n + m \log n)$ time if the weight matrices A and B of G are both concave, where for $0 \le i \le n$ and $0 \le j \le m$, $A[i, j]$ and $B[j, i]$ are the weights of the edges (x_i, y_j) and (y_j, x_i) in G, respectively. As demonstrated in this paper, the new problem is a powerful tool and we believe that it can be used to solve more problems.

Key words. graph algorithm, shortest path, traveling salesman problem, minimum latency tour problem, concave matrix

AMS subject classifications. 68Q25, 68R10

PII. S0097539797322255

1. Introduction.

The traveling salesman problem (TSP) is a classical problem of combinatorial optimization. It has been the testing ground of many new algorithmic ideas during the past half-century: dynamic programming, linear programming, genetic algorithms, etc. The TSP is NP-hard and even nonapproximatable. This has motivated researchers to look at its special cases. It turns out that various special cases of the TSP remain NP-hard but are approximatable. Among the special cases solvable in polynomial time, the TSP for points on a convex polygon is well known. In this special case, we are given a set S of n points on the boundary of a convex polygon C and two points x and y in S and are requested to compute a shortest tour starting at x, visiting all the points in $S - \{x, y\}$, and ending at y. Here, the distance between two points in S is the Euclidean distance between them. This special case can be solved in $O(n^2)$ time via dynamic programming [10], but whether it can be solved in $o(n^2)$ time was an open question (posed in [10]). In this paper, we give an affirmative answer to this open question. More specifically, we show that the TSP for points on a convex polygon can be solved in $O(n \log n)$ time.

The minimum latency problem (MLP) is as follows: We are given a metric space M on n points $\{x_1, \ldots, x_n\}$ and are requested to compute a tour in M starting at x_1

*Received by the editors June 4, 1997; accepted for publication (in revised form) January 28, 1998; published electronically September 14, 1999. A preliminary version of this work was presented as *Shortest path in complete bipartite digraph problem and its applications*, 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, 1997.
http://www.siam.org/journals/sicomp/29-1/32225.html

†Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260 (xinhe@cse.buffalo.edu). The research of this author was supported in part by NSF grant CCR-9205982.

‡Department of Mathematical Sciences, Tokyo Denki University, Hatoyama, Saitama 350-03, Japan.

which minimizes the sum of the arrival times at the n points. More precisely, if T is a tour starting at x_1 we say that the *latency* of x_i with respect to T is the distance traveled in T before reaching x_i; and the latency of T is the sum of the latencies of the n points. The goal is then to find a tour of minimum latency. The MLP is a well-studied problem in the operations research literature, where it is also known as the "delivery-man problem" and the "traveling repairman problem" (see [5] for more discussions and references). Although it looks similar to the TSP, the MLP is very different from the TSP in nature [5]. Generally, the MLP is NP-complete [5]. Even for points on a tree or on a convex polygon, it is not known whether the MLP is in P or NP-complete [5]. The case where points are on a straight line was considered in [1, 5]. This case is interesting since it is exactly the following *disk head scheduling problem*: A disk head moves along a straight line L. The head must visit a set of n points on L in order to satisfy disk access requests. The time needed to travel is proportional to the distance being traveled. Once the head reaches a point, the disk access time can be ignored (since the disk rotating speed is much higher than the head moving speed). We want to find a tour of the head such that the average delay (or equivalently, the total delay) of all requests is minimized. The MLP for this special case can be solved in $O(n^2)$ time via dynamic programming [1, 5]. However, whether it can be solved in $o(n^2)$ time was an open question [1]. In this paper, we answer this question in the affirmative by giving an $O(n \log n)$-time algorithm for it.

We obtain the two results mentioned above by efficient reductions to a *single* problem called the *shortest path in bipartite digraph* (SPBD) problem, which is defined as follows. Let $G = (X, Y, E)$ be a complete bipartite digraph with $X = \{x_0, x_1, \ldots, x_n\}$ and $Y = \{y_0, y_1, \ldots, y_m\}$. Each edge $e \in E$ is associated with a real-valued weight $w(e)$. We use $x_i \to y_j$ and $y_j \to x_i$ to denote the edges. Let $A[0..n, 0..m]$ be the matrix with $A[i,j] = w(x_i \to y_j)$ and $B[0..m, 0..n]$ be the matrix with $B[i,j] = w(y_i \to x_j)$. The weight of a (directed) path P in G is defined as $w(P) = \sum_{e \in P} w(e)$. Given such a digraph G, the SPBD problem is to find a path P in G from x_0 to x_n such that $w(P)$ is minimized. For arbitrary weight matrices, we must examine all the edges of G in order to find the shortest path. Thus we need at least $\Omega(nm)$ time to solve the problem. A matrix $M[0..n, 0..m]$ is called *concave* if the following hold:

$$M[i_1, j_1] + M[i_2, j_2] \le M[i_2, j_1] + M[i_1, j_2] \quad \text{for} \quad 0 \le i_1 \le i_2 \le n \quad \text{and} \quad 0 \le j_1 \le j_2 \le m. \tag{1.1}$$

Concave matrices were first discussed in [12] and have been very successfully used in solving various problems (see [2, 3, 4, 6, 7, 8, 9, 11, 12, 13] and the references cited within). Given two matrices $A[0..n, 0..m]$ and $B[0..m, 0..n]$, the product matrix $W[0..n, 0..n] = A \times B$ is defined by

$$W[i,j] = \min_{0 \le k \le m} (A[i,k] + B[k,j]). \tag{1.2}$$

For the SPBD problem we require that G contains no negative cycles since otherwise the shortest path of G is not well defined. If both A and B are concave, as we will prove later, this requirement is satisfied if all the entries on the main diagonal of the product matrix $W = A \times B$ are nonnegative. In section 4 we will prove the following theorem.

THEOREM 1.1. *Given two concave matrices A and B such that all the entries on the main diagonal of the product matrix $W = A \times B$ are nonnegative, the SPBD problem defined by A and B can be solved in $O(n + m \log n)$ time.*

Even for this special case, no algorithm with $o(nm)$ running time was previously known. In this theorem we assume that the matrices A and B are not explicitly given. Rather, an entry is computed in constant time when it is needed. This is true when we apply our SPBD algorithm to solve the TSP and the MLP.

The rest of this paper is organized as follows. In section 2 we present the reduction from the TSP for points on a convex polygon to the SPBD problem. In section 3 we reduce the MLP for points on a straight line to the SPBD problem. In section 4 we prove Theorem 1 by giving an $O(n + m \log n)$-time algorithm for the SPBD problem. Section 5 concludes the paper.

2. The TSP for points on a convex polygon.

Let C be a convex polygon and Z be the set of the corner vertices of C. The members of Z will be called *points*. For two points z_1 and z_2 in Z, let $d(z_1, z_2)$ denote the Euclidean distance between z_1 and z_2. The points in Z induce an edge-weighted complete graph G_Z, where the weight on each edge $\{z_1, z_2\}$ is $d(z_1, z_2)$. We identify each edge $\{z_1, z_2\}$ with the line segment whose endpoints are z_1 and z_2. The weight of a path P in G_Z is the sum of the weights on the edges in P and is denoted by $w(P)$. Fix two points x and y in Z. Hereafter, a Hamiltonian path in G_Z always means one from x to y. Our goal is to compute an optimal Hamiltonian path in G_Z, i.e., a Hamiltonian path in G_Z of minimum weight. An easy geometric argument shows that every optimal path P must be simple, i.e., no two edges of P cross each other.

Let $x = x_0, x_1, \ldots, x_n = y$ be the points in Z that lie on the boundary of C from x to y in the clockwise order. Let C_X be the portion of the boundary of C which starts at x, includes $x_1, \ldots, x_{n-1}$, and ends at y. Similarly, let $x = y_0, y_1, \ldots, y_m = y$ be the points in Z that lie on the boundary of C from x to y in the counterclockwise order. Let C_Y be the portion of the boundary of C which starts at x, includes $y_1, \ldots, y_{m-1}$, and ends at y. For $0 \le i < j \le n$ let $x_i \overset{X}{\rightarrow} x_j$ denote the portion of C_X from x_i to x_j. For $0 \le i < j \le m$ let $y_i \overset{Y}{\rightarrow} y_j$ denote the portion of C_Y from y_i to y_j.

Let P be an optimal Hamiltonian path from $x_0 = y_0$ to $x_n = y_m$ in G_Z. Depending on whether the first and the last edges of P are in C_X or in C_Y, there are four possibilities. We assume that both the first and the last edges of P are in C_Y. Then P must be of the following form:

$$\begin{aligned} x_{i_0} = x_0 = y_0 \overset{Y}{\rightarrow} y_{j_1} \rightarrow x_1 \overset{X}{\rightarrow} x_{i_1} \rightarrow y_{j_1+1} \overset{Y}{\rightarrow} y_{j_2} \rightarrow x_{i_1+1} \overset{X}{\rightarrow} \cdots \overset{X}{\rightarrow} x_{i_t} \\ = x_{n-1} \rightarrow y_{j_t+1} \overset{Y}{\rightarrow} y_m = x_n \end{aligned}$$

for some $0 < j_1 < j_2 < \cdots < j_t < m-1$ and $0 = i_0 < i_1 < i_2 < \cdots < i_t = n-1$. (See Figure 2.1. In Figure 2.1(b) the points in C_X and C_Y are drawn on two vertical lines for the sake of clarity.) We use the following *dummy path* P' to represent P (the edges of P' are shown as dashed lines in Figure 2.1(b)):

$$P' : \ x_{i_0}(= x_0) \rightarrow y_{j_1} \rightarrow x_{i_1} \rightarrow y_{j_2} \rightarrow x_{i_2} \rightarrow \cdots \rightarrow y_{j_{t-1}} \rightarrow x_{i_{t-1}} \rightarrow y_{j_t} \rightarrow x_{i_t}(= x_{n-1}).$$

The edges $x_{i_{l-1}} \rightarrow y_{j_l}$ and $y_{j_l} \rightarrow x_{i_l}$ in P' are called *dummy edges*. P is completely specified by P'.

For each dummy edge $x_{i_{l-1}} \rightarrow y_{j_l}$ in P', the edge $x_{i_{l-1}} \rightarrow x_{i_{l-1}+1}$ is not in P, while the edge $y_{j_l} \rightarrow x_{i_{l-1}+1}$ is in P. For each dummy edge $y_{j_l} \rightarrow x_{i_l}$ in P', the edge $y_{j_l} \rightarrow y_{j_l+1}$ is not in P, while the edge $x_{i_l} \rightarrow y_{j_l+1}$ is in P. This motivates the following definition of the weights of dummy edges $x_i \rightarrow y_j$ and $y_j \rightarrow x_i$ given in the matrices $A[0..n-1, 0..m-1]$ and $B[0..m-1, 0..n-1]$:

$$A[i, j] = w(x_i \rightarrow y_j) = d(x_{i+1}, y_j) - d(x_i, x_{i+1}), \tag{2.1}$$

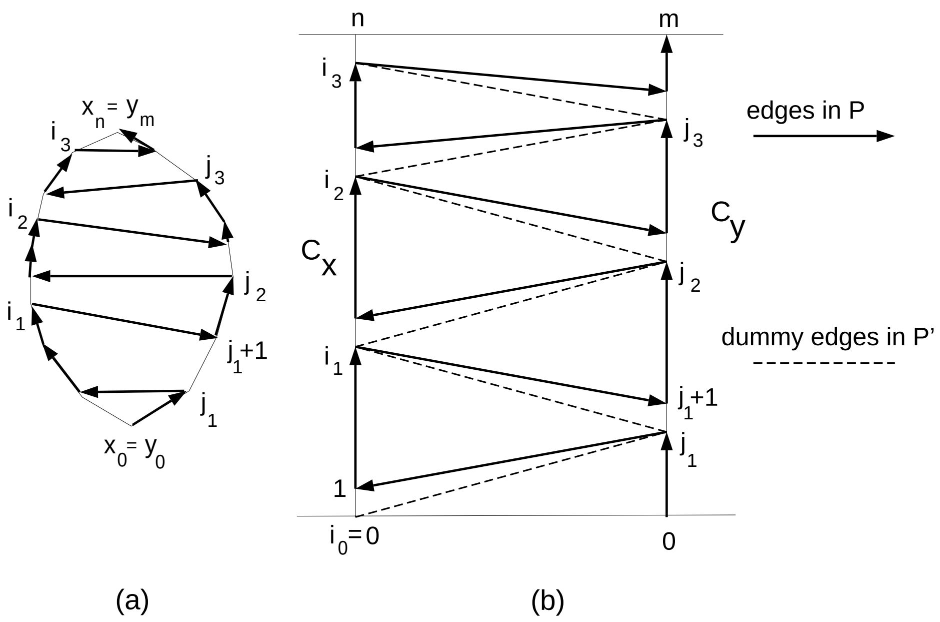


FIG. 2.1. (a) *An optimal path* P *in a convex polygon;* (b) *a simplified representation of* P.

$$B[j,i] = w(y_j \to x_i) = d(y_{j+1}, x_i) - d(y_j, y_{j+1}). \tag{2.2}$$

Note that $A[0,0] = B[0,0] = 0$. Let $S_X = \sum_{i=1}^{n-1} d(x_{i-1}, x_i)$ and $S_Y = \sum_{j=1}^{m} d(y_{j-1}, y_j)$. It is easy to verify that the total weight of P is

$$w(P) = S_X + S_Y + \sum_{l=1}^{t} A[i_{l-1}, j_l] + \sum_{l=1}^{t} B[j_l, i_l]. \tag{2.3}$$

Although the above discussion is carried out by assuming that the first and the last edges of P are in C_Y, it also applies to other cases. For example, if the first edge of P is in C_X, we let $j_1 = 0$. If the last edge of P is in C_X, we let $j_t = m-1$. It is easy to verify that (2.3) is valid for these cases, too.

Since the term $S_X + S_Y$ in (2.3) is fixed for any P, to minimize $w(P)$ we need only to minimize the *reduced weight* $w(P')$ defined as follows:

$$w(P') = \sum_{l=1}^{t} A[i_{l-1}, j_l] + \sum_{l=1}^{t} B[j_l, i_l]. \tag{2.4}$$

Let $G = (X, Y, E)$ be the complete bipartite digraph with $X = \{x_0, x_1, \ldots, x_{n-1}\}$ and $Y = \{y_0, y_1, \ldots, y_{m-1}\}$ and the weight matrices A and B. Then a dummy path P' with minimum reduced weight $w(P')$ is exactly a shortest path in G from x_0 to x_{n-1}.

For $0 \le i < i' \le n-1$ and $0 \le j < j' \le m-1$, by the definition of A and the fact that C is a convex polygon, we have $A[i,j] + A[i',j'] - A[i,j'] - A[i',j] = d(x_{i+1}, y_j) + d(x_{i'+1}, y_{j'}) - d(x_{i+1}, y_{j'}) - d(x_{i'+1}, y_j) < 0$.

Thus A is concave. Similarly, we can show that B is also concave. Let $W = A \times B$. Then

$$W[i,i] = \min_{0 \le j \le m-1} [d(x_{i+1}, y_j) - d(x_i, x_{i+1}) + d(y_{j+1}, x_i) - d(y_j, y_{j+1})].$$

By the quadrangle inequality the expression within the min sign is > 0 for each j. Thus $W[i,i] > 0$ for all $0 \le i \le n-1$. By Theorem 1.1 we have the following theorem.

THEOREM 2.1. *The traveling salesman problem for points on an N-point convex polygon can be solved in $O(N \log N)$ time.*

3. The MLP for points on a straight line. Consider a set S of $n+1$ points, a symmetric distance matrix $d[0..n, 0..n]$, and a tour T which visits the points of S in some order. The *latency* of a point $p \in S$ with respect to T is the length of the tour from the starting point to the first occurrence of p. More precisely, suppose that T visits the points in S in the order $p_0, p_1, \ldots, p_n$ starting at p_0. Let $d(p_{i-1}, p_i)$ be the distance traveled along T between p_{i-1} and p_i. Then the latency of p_i is $w(p_i) = \sum_{j=1}^{i} d(p_{j-1}, p_j)$. The *total latency* $w(T)$ of T is the sum of the latencies of all the points in S: $w(T) = \sum_{i=1}^{n} w(p_i)$. Or, equivalently,

$$w(T) = \sum_{k=1}^{n} d(p_{k-1}, p_k)(n-k+1). \tag{3.1}$$

We wish to find a tour T with minimum $w(T)$. In this section we show that the MLP for points on a straight line can be reduced to the SPBD problem and solved in $O(n \log n)$ time.

Let $S = \{x_n, x_{n-1}, \ldots, x_1, x_0 = y_0 = 0, y_1, y_2, \ldots, y_m\}$ be a set of $N = n+m+1$ distinct points on the real line from left to right. We overload x_i (and y_j) to denote both a point and the distance from it to the origin. The tour starts at the point 0. Define

$$\begin{aligned} w(T_X) &= \sum_{k=1}^{n} (x_k - x_{k-1})(n-k+1), \\ w(T_Y) &= \sum_{k=1}^{m} (y_k - y_{k-1})(m-k+1). \end{aligned}$$

$w(T_X)$ is the total latency of the tour T_X that starts at $x_0 = 0$ and travels the points $x_1, x_2, \ldots, x_n$ in this order. $w(T_Y)$ is the total latency of the tour T_Y that starts at $y_0 = 0$ and travels the points $y_1, y_2, \ldots, y_m$ in this order.

Consider an optimal tour T for S. Depending on whether the first and the last edges of T are to the left or to the right, there are four possibilities. If, for example, the first edge is to the right and the last edge is to the left, then T must be of the following form (see Figure 3.1):

$$x_{i_0} = y_{j_0} = x_0 = y_0 \overset{\Delta}{\rightarrow} y_{j_1} \overset{\Delta}{\rightarrow} x_{i_1} \overset{\Delta}{\rightarrow} y_{j_2} \overset{\Delta}{\rightarrow} x_{i_2} \overset{\Delta}{\rightarrow} \cdots \overset{\Delta}{\rightarrow} x_{i_{t-1}} \overset{\Delta}{\rightarrow} y_{j_t} = y_m \overset{\Delta}{\rightarrow} x_{i_t} = x_n$$

for some $0 = j_0 < j_1 < j_2 < \cdots < j_{t-1} < j_t = m$ and $0 = i_0 < i_1 < \cdots < i_{t-1} < i_t = n$. (The notation $x_i \overset{\Delta}{\rightarrow} y_j$ denotes the straight line segment whose end points are x_i and y_j consisting of several edges.) We use the following *dummy tour* T' to represent T:

$$T' : \ x_{i_0} (= x_0) \rightarrow y_{j_1} \rightarrow x_{i_1} \rightarrow \cdots \rightarrow x_{i_{t-1}} \rightarrow y_{j_t} (= y_m) \rightarrow x_{i_t} (= x_n).$$

T is completely specified by T'. Define the *reduced weight* of the dummy tour T' to be

$$w(T') = \sum_{l=1}^{t} y_{j_l} [n+m-j_l-i_{l-1}] + \sum_{l=1}^{t} x_{i_l} [n+m-j_l-i_l]. \tag{3.2}$$

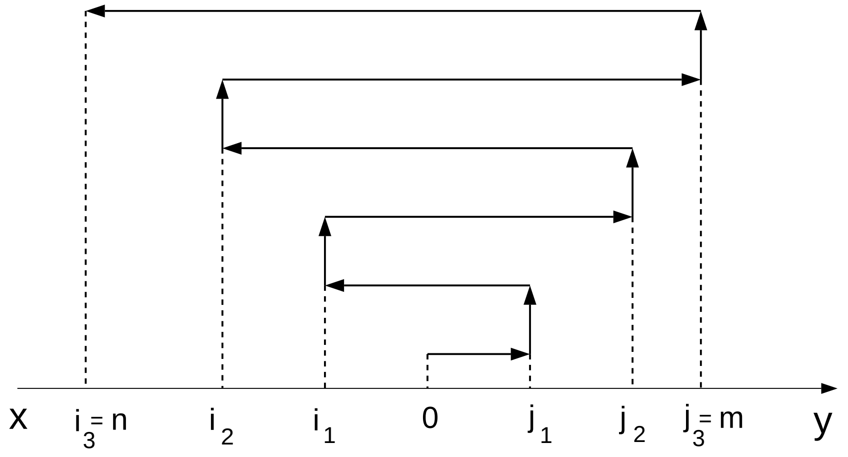


FIG. 3.1. *Optimal tour for points on a straight line.*

LEMMA 3.1. *The weight of T and the reduced weight of T' satisfies*

$$w(T) = w(T_X) + w(T_Y) + 2w(T'). \tag{3.3}$$

Proof. First we note that $w(T_X) + w(T_y)$ is the sum of the shortest-path distances from $x_0 = y_0$ to the points x_i $(1 \le i \le n)$ and y_j $(1 \le j \le m)$. In the tour T, the latency of each point will be the shortest-path distance plus some additional delay (caused by the zigzag-shaped detour). We need to compute this additional delay. Each "loop" of the form "from y_0 to y_{j_l} and back" contributes a delay of $2y_{j_l}$ to each point that is as yet unvisited when this loop is traversed; there are $(n + m - j_l - i_{l-1})$ such points. So the additional delay contributed by this loop is

$$2y_{j_l}(n + m - j_l - i_{l-1}).$$

Similarly, the additional delay contributed by a loop of the form "from x_0 to x_{i_l} and back" is

$$2x_{i_l}(n + m - j_l - i_l).$$

Summing up all these additional delay terms, we have $w(T) = w(T_X) + w(T_Y) + \sum_{l=1}^{t} 2y_{j_l}[n + m - j_l - i_{l-1}] + \sum_{l=1}^{t} 2x_{i_l}[n + m - j_l - i_l] = w(T_X) + w(T_Y) + 2w(T')$. $\square$

Although Lemma 3.1 is proved by assuming that the first edge of T is to the right and the last edge of T is to the left, it also applies to other cases. For example, if the first edge of T is to the left, we can let $j_1 = 0$. If the last edge of T is to the right, we can let $i_{t-1} = n$ and delete from T the subpath from y_m to x_n. It can be verified that (3.3) is valid for those cases, too. Since the term $w(T_X) + w(T_Y)$ is fixed for all T, in order to minimize $w(T)$ we need only to minimize $w(T')$.

Let $G = (X, Y, E)$ be the complete bipartite digraph with $X = \{x_0, x_1, \ldots, x_n\}$, $Y = \{y_0, y_1, \ldots, y_m\}$, and the weight matrices $A[0..n, 0..m]$ and $B[0..m, 0..n]$ defined as follows:

$$A[i,j] = w(x_i \to y_j) = y_j(n+m-i-j),$$
$$B[j,i] = w(y_j \to x_i) = x_i(n+m-i-j).$$

Note that $A[0,0] = B[0,0] = 0$. It is easy to check that a dummy tour T' with minimum reduced weight $w(T')$ is exactly a shortest path in G from x_0 to x_n. By the definition of A, for $0 \leq i < i' \leq n$ and $0 \leq j < j' \leq m$, we have $(A[i,j] + A[i',j']) - (A[i,j'] + A[i',j]) = (i'-i)(y_j - y_{j'}) < 0$. Thus A is concave. Similarly, we can show that B is also concave. Since all the entries of A and B are nonnegative, all the entries of $W = A \times B$ are nonnegative. Thus by Theorem 1.1 we have the following.

THEOREM 3.2. *The minimum latency problem for a set of N points on straight line can be solved in $O(N \log N)$ time.*

4. Solving the SPBD problem.

Our algorithm for solving the SPBD problem is a reduction to an enhanced version of the *least weight subsequence* (LWS) problem. In section 4.1 we describe the LWS problem and its enhanced version. The reduction from the SPBD problem to the enhanced LWS problem is discussed in section 4.2. An algorithm for solving the enhanced LWS problem is given in section 4.3. In section 4.4 we give a complete description of our algorithm for the SPBD problem and analyze its time complexity.

4.1. The LWS problem and the enhanced LWS problem.

The following LWS problem was introduced in [8]. Given a sequence $\{x_0, x_1, \ldots, x_n\}$ and a real-valued weight function $w(x_i, x_j)$ defined for indices $0 \leq i < j \leq n$, find an integer $k \geq 1$ and a sequence $S = \{0 = i_0 < i_1 < \cdots < i_{k-1} < i_k = n\}$ such that the total weight $w(S) = \sum_{l=1}^{k} w(x_{i_{l-1}}, x_{i_l})$ is minimized. The LWS problem can also be formulated as a graph problem: Given an acyclic digraph G with vertex set $V = \{x_0, \ldots, x_n\}$, the edge set $E = \{x_i \to x_j \mid 0 \leq i < j \leq n\}$, and the weight function w, we wish to find a shortest path in G from x_0 to x_n. For an arbitrary weight function w, the LWS problem requires $\Omega(n^2)$ time to solve. The weight function w is *concave* if the following hold:

$$w(x_{i_1}, x_{j_1}) + w(x_{i_2}, x_{j_2}) \leq w(x_{i_2}, x_{j_1}) + w(x_{i_1}, x_{j_2}) \quad \text{for } 0 \leq i_1 \leq i_2 \leq j_1 \leq j_2 \leq n. \tag{4.1}$$

If the weight function is concave, then we have an instance of the concave LWS problem. Hirschberg and Larmore showed that the concave LWS problem can be solved in $O(n \log n)$ time [8]. Similar algorithms were developed in [6, 7]. Wilber discovered an elegant linear-time algorithm for solving this problem [11]. All these algorithms assume that each entry $w(i,j)$ can be computed in constant time. In this paper we consider only the concave LWS problem. From now on the phrase "LWS problem" always means the concave LWS problem.

The *enhanced* version of the LWS problem is defined as follows. An instance of the enhanced LWS problem is a sequence $\{x_0, x_1, \ldots, x_n\}$ and a real-valued concave weight function $w(x_i, x_j)$ defined on *all* $0 \leq i, j \leq n$ such that $w(x_i, x_i) \geq 0$ for all $0 \leq i \leq n$. We want to find a sequence $S = \{0 = i_0, i_1, \ldots, i_k = n\}$ ($i_0, i_1, \ldots, i_k$ are not necessarily in increasing order, as in the ordinary LWS problem), such that the total weight $w(S) = \sum_{l=1}^{k} w(x_{i_{l-1}}, x_{i_l})$ is minimized. In terms of the graph formulation, given a complete digraph G with vertex set $V = \{x_0, x_1, \ldots, x_n\}$ and a weight function w, we wish to find a shortest x_0 to x_n path in G. Let $e = x_i \to x_j$

be an edge of G. If $i < j$, e is called a *forward* edge. If $i = j$, e is called a *selfloop*. If $i > j$, e is called a *backward* edge. We require that the weight of the selfloops of G be nonnegative since otherwise the weight of the shortest path in G would be $-\infty$.

LEMMA 4.1. *For any instance of the enhanced LWS problem there exists a shortest x_0 to x_n path consisting of only forward edges.*

Proof. Let P be a shortest path from x_0 to x_n in G such that the number of edges in P is minimum. Since $w(x_i, x_i) \geq 0$ for all i, we may assume that P contains no selfloops. Toward a contradiction, suppose that P contains a backward edge. Let $x_{i_l} \to x_{i_{l+1}}$ be the first backward edge of P. Then $i_l > i_{l+1}$ and $i_l > i_{l-1}$. By the concavity of w and the assumption that $w(x_i, x_i) \geq 0$ for all x_i, we have $w(x_{i_{l-1}}, x_{i_{l+1}}) \leq w(x_{i_{l-1}}, x_{i_{l+1}}) + w(x_{i_l}, x_{i_l}) \leq w(x_{i_{l-1}}, x_{i_l}) + w(x_{i_l}, x_{i_{l+1}})$. Thus replacing the two edges $x_{i_{l-1}} \to x_{i_l} \to x_{i_{l+1}}$ in P by a single edge $x_{i_{l-1}} \to x_{i_{l+1}}$, we get a path P' such that $w(P') \leq w(P)$ and the number of edges in P' is one less than that in P. This contradicts the choice of P. $\square$

Lemma 4.1, together with the concavity of w, implies that there are no negative cycles in any instance of the enhanced LWS problem. It also implies that we can ignore all the backward edges and selfloops when solving the enhanced LWS problem.

4.2. Reduction. In this section we show that the SPBD problem can be reduced to the enhanced LWS problem. First we need several technical lemmas. The following lemma was proved in [12].

LEMMA 4.2. *If both A and B are concave, then the product matrix $W = A \times B$ is also concave.*

For $0 \leq i \leq n$ and $0 \leq j \leq n$, let $I(i, j)$ denote the smallest index k that realizes the minimum value in definition (1.2). Namely, $I(i, j)$ is the smallest index such that $W[i, j] = A[i, I(i, j)] + B[I(i, j), j]$. The following lemma was proved in [12].

LEMMA 4.3. *For any i, j ($0 \leq i < n$, $0 \leq j < n$), we have $I(i, j) \leq I(i, j + 1) \leq I(i + 1, j + 1)$.*

Remark. The definitions of concavity and the matrix product in [12] are slightly different from the definitions used here. In [12] a concave matrix is an upper triangular matrix such that the condition (1.1) is true for $i_1 \leq i_2 \leq j_1 \leq j_2$. In the matrix product definition (1.2) the minimum is taken over $i \leq k \leq j$. Under these definitions, Yao proved Lemmas 4.2 and 4.3. Under our definitions, Lemmas 4.2 and 4.3 can be proved via similar methods.

Let (i, j) and (i', j') be two pairs of indices. If $i \leq i'$ and $j \leq j'$, we write $(i, j) \prec (i', j')$. By Lemma 4.3, $(i, j) \prec (i', j')$ implies $I(i, j) \leq I(i', j')$.

LEMMA 4.4. *Let $(i_1, j_1), (i_2, j_2), \ldots, (i_p, j_p)$ be p pairs of indices such that $(i_l, j_l) \prec (i_{l+1}, j_{l+1})$ for all $1 \leq l < p$. Then $I(i_1, j_1), I(i_2, j_2), \ldots, I(i_p, j_p)$ can be computed in $O(m \log p)$ time.*

Proof. This can be done in a binary search fashion. More specific, we find $I_{p/2} = I(i_{p/2}, j_{p/2})$ in the first stage, find $I(i_{p/4}, j_{p/4})$ (by searching the range $0..I_{p/2}$) and $I(i_{3p/4}, j_{3p/4})$ (by searching the range $I_{p/2}..m$) in the second stage, and so on. In total, there are $\log p$ stages. Since each stage can be done in $O(m)$ time, the lemma holds. $\square$

Consider an instance of the SPBD problem defined by a complete bipartite digraph $G = (X, Y, E)$ and two concave weight matrices A and B. Let $G' = (X, E')$ be the complete digraph on X with the concave weight matrix $w = A \times B$. If $w(x_i, x_i) \geq 0$ for all $0 \leq i \leq n$, then G' and w define an instance of the enhanced LWS problem.

Let $P' :\ x_0 = x_{i_0} \to x_{i_1} \to \cdots \to x_{i_k} = x_n$ be a shortest path in G' from x_0 to x_n. For each l ($0 \leq l \leq k$), let $j_l = I(i_{l-1}, i_l)$. Then $P :\ x_0 = x_{i_0} \to y_{j_1} \to x_{i_1} \to$

$y_{j_2} \to \cdots \to y_{j_k} \to x_{i_k} = x_n$ is a path in G from x_0 to x_n. Let $w(P')$ denote the weight of P' in G' and $w(P)$ the weight of P in G. Clearly, $w(P) = w(P')$. We will show that $w(P)$ is minimum among all paths from x_0 to x_n in G.

Let Q be a shortest path in G from x_0 to x_n. Since G is bipartite, Q is a concatenation of subpaths $Q_1, Q_2, \ldots, Q_p$ for some $p \geq 1$, where each Q_l $(1 \leq l \leq p)$ consists of two edges $x_{i'_{l-1}} \to y_{j'_l} \to x_{i'_l}$ ($i'_0 = 0$ and $i'_p = n$). For each $1 \leq l \leq p$, if $j'_l \neq I(i'_{l-1}, i'_l)$, we can replace Q_l by the subpath $x_{i'_{l-1}} \to y_{I(i'_{l-1}, i'_l)} \to x_{i'_l}$ without increasing the total weight $w(Q)$. Therefore, without loss of generality, we may assume that $j'_l = I(i'_{l-1}, i'_l)$ for all $1 \leq l \leq p$. Hence the weight of Q_l is $w[i'_{l-1}, i'_l]$. Thus Q corresponds to a path $Q' = \{x_0 = x_{i'_0} \to x_{i'_1} \to \cdots \to x_{i'_p} = x_n\}$ from x_0 to x_n in G' with $w(Q') = w(Q)$. Since the weight of P' is minimum among all such paths in G', we have $w(P) = w(P') \leq w(Q') = w(Q)$. Thus P is a shortest path in G from x_0 to x_n.

LEMMA 4.5. *Let A and B be two concave matrices such that all the entries on the main diagonal of the product matrix $w = A \times B$ are nonnegative. If the enhanced LWS problem defined by the matrix w can be solved in $T(n,m)$ time, then the SPBD problem defined by A and B can be solved in $O(T(n,m) + m \log n)$ time.*

Proof. In order to solve the SPBD problem defined by matrices A and B, we first solve the enhanced LWS problem defined by the matrix $w = A \times B$. Let P' : $x_0 = x_{i_0} \to x_{i_1} \to \cdots \to x_{i_k} = x_n$ be the solution path found. We compute $j_1, j_2, \ldots, j_k$, where $j_l = I(i_{l-1}, i_l)$. Since $(i_0, i_1) \prec (i_1, i_2) \prec \cdots \prec (i_{k-1}, i_k)$, this can be done in $O(m \log n)$ time by Lemma 4.4. The path $x_0 = x_{i_0} \to y_{j_1} \to x_{i_1} \to \cdots \to y_{j_k} \to x_{i_k} = x_n$ is the solution for the SPBD problem. ∎

We want to use Wilber's algorithm in [11] to solve our enhanced LWS problem. In order to do this, however, we have to overcome two difficulties. First, Wilber's algorithm is for solving the (ordinary) LWS problem which is defined by an upper triangle matrix while our problem is defined by a full matrix. Second, Wilber's algorithm assumes that each entry $w(i,j)$ can be evaluated in $O(1)$ time. In our case, an entry of the matrix $w = A \times B$ may need $\Theta(m)$ time to evaluate. We will address these two issues in the next section.

4.3. An algorithm for the enhanced LWS problem. Our algorithm for the enhanced LWS problem is a modification of Wilber's algorithm for the LWS problem. First, we briefly review Wilber's algorithm. (We assume that the reader is familiar with [11].) Then we show how to modify Wilber's algorithm to solve our problem.

Consider an instance of the LWS problem with the sequence $\{x_0, x_1, \ldots, x_n\}$ and the weight matrix $w(x_i, x_j)$. Recall that w is an $(n+1) \times (n+1)$ upper triangular matrix. Let $f(0) = 0$ and, for $1 \leq j \leq n$, let $f(j)$ be the weight of the lowest weight subsequence between x_0 and x_j. For $0 \leq i < j \leq n$ let $g(i,j)$ be the weight of the lowest-weight subsequence between x_0 and x_j whose next-to-last element is x_i. Then we have

$$(4.2) \qquad \begin{cases} f(j) = \min_{0 \leq i < j} g(i,j) & \text{for } 1 \leq j \leq n, \\ g(i,j) = f(i) + w(x_i, x_j) & \text{for } 0 \leq i < j \leq n. \end{cases}$$

To solve the LWS problem it is enough to compute $f(1), f(2), \ldots, f(n)$. Adding $f(i_1) + f(i_2)$ to both sides of inequality (4.1) and applying definition (4.2), we get

$$g(i_1, j_1) + g(i_2, j_2) \leq g(i_1, j_2) + g(i_2, j_1) \quad \text{for } 0 \leq i_1 \leq i_2 \leq j_1 \leq j_2 \leq n.$$

Consider a matrix $M[0..n, 0..m]$. For each column index $0 \leq j \leq m$ let $i(j)$ be the smallest row index such that $M(i(j), j)$ equals the minimum value in the jth column of M. The *column minima searching* problem for M is to find the $i(j)$'s for all $0 \leq j \leq m$. M is called *monotone* if $i(j_1) \leq i(j_2)$ for all $0 \leq j_1 < j_2 \leq m$. M is *totally monotone* if every 2×2 submatrix of M is monotone [3]. If M is concave, then it is easy to check that M is totally monotone. (The reverse is not necessarily true.) For a totally monotone matrix M, the column minima searching problem for M can be solved in $O(n+m)$ time, assuming that each entry of M can be evaluated in $O(1)$ time [3]. Following [8], we will refer to the algorithm in [3] as the SMAWK algorithm.

We extend the definition of g by setting $g(i,j) = +\infty$ for $0 \leq j \leq i \leq n$. Then g becomes a full $(n+1) \times (n+1)$ matrix. It is easy to verify that the extended matrix g is totally monotone. (The only role of the $+\infty$ entries is to make g a full matrix for convenience. These entries otherwise have no effect on the computation.) Our goal is to determine the row index of the minimum value in each column of g (which gives $f(1), \ldots, f(n)$). One might simply want to apply the SMAWK algorithm to g. But we cannot, because for $i < j$, the value of $g(i,j)$ depends on $f(i)$ and $f(i)$ depends on $g(0,i), g(1,i), \ldots, g(i-1,i)$. Thus we cannot compute the value of g in $O(1)$ time as required by the SMAWK algorithm.

Wilber's algorithm starts in the upper left corner of g and works rightward and downward, at each iteration learning enough new values of f to be able to compute enough new values of g to continue with the next iteration. Actually, during one step of each iteration, the algorithm might "pretend" to know values of f that it really does not have. At the end of the iteration, the assumed value of f is checked for validity.

We use $f(j)$ and $g(i,j)$ to refer to the correct values of f and g, respectively. The currently computed value of $f(j)$ is denoted by $F(j)$ and sometimes will be incorrect. The currently computed value of $g(i,j)$ is denoted by $G[i,j]$ and is always computed as $F[i] + w(i,j)$. Therefore $G[i,j] = g(i,j)$ iff $F(i) = f(i)$. The algorithm does not explicitly store the matrices w, g, G. Rather, their entries are calculated when needed. Let $G[i_1, i_2; j_1, j_2]$ denote the submatrix of G consisting of the intersection of rows i_1 through i_2 and columns j_1 through j_2. $G[i_1, i_2; j]$ denotes the intersection of rows i_1 through i_2 with column j. The rows of G are indexed from 0 and the columns are indexed from 1. Wilber's algorithm is as follows.

```
WILBER'S ALGORITHM.
$F[0] \leftarrow c \leftarrow r \leftarrow 0$.
**while** $(c < n)$ do
    1. $p \leftarrow \min\{2c - r + 1, n\}$.
    2. Apply the SMAWK algorithm to find the minimum in each column of sub-
       matrix $S = G[r, c; c+1, p]$. For $j \in [c+1, p]$, let $F[j]$ = the minimum value
       found in $G[r, c; j]$.
    3. Apply the SMAWK algorithm to find the minimum in each column of the
       submatrix $T = G[c+1, p-1; c+2, p]$. For $j \in [c+2, p]$, let $H[j]$ = the
       minimum value found in $G[c+1, p-1; j]$.
    4. If there is an integer $j \in [c+2, p]$ such that $H[j] < F[j]$, then set $j_0$ to the
       smallest such integer. Otherwise set $j_0 \leftarrow p+1$.
    5. **if** $(j_0 = p+1)$ **then** $c \leftarrow p$;
       **else** $F[j_0] \leftarrow H[j_0]$; $r \leftarrow c+1$; $c \leftarrow j_0$.
**end**.
```

Figure 4.1 shows the submatrices S and T during a typical iteration of the algorithm. (This figure is taken from [11].) Each time we are at the beginning of the

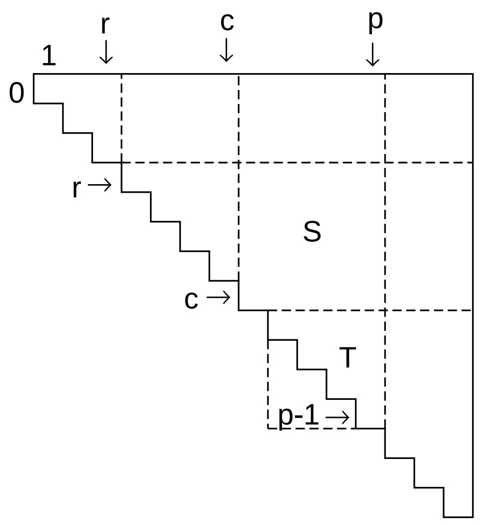


FIG. 4.1. *A typical iteration of Wilber's algorithm.*

loop, the following invariants hold:

(a) $r \geq 0$ and $c \geq r$.

(b) For each $j \in [0, c]$, $F[j] = f(j)$.

(c) All the minima in columns $c + 1$ through n of g are in rows $\geq r$.

These invariants are clearly satisfied at the beginning when $r = c = 0$.

Invariant (b) implies that $G[i, j] = g(i, j)$ for all j and all $i \in [0, c]$. So the entries of the submatrix S are the same as the corresponding entries of g. Therefore S is totally monotone, and for each $j \in [c + 1, p]$ step 2 sets $F[j]$ to the minimum value of the subcolumn $g(r, c; j)$. Also, since S contains all the finite-valued cells in column $c + 1$ of g that are in rows $\geq r$, we have $F[c + 1] = f(c + 1)$ at the end of step 2. On the other hand, we do not necessarily have $F[j] = f(j)$ for any $j \in [c + 2, p]$, since g has finite-valued cells in those columns that are in rows $\geq r$ and not in S.

In step 3 we proceed as if $F[j] = f(j)$ for all $j \in [c + 1, p - 1]$. Since this may be false, some of the values in T may be bogus. However, T is always totally monotone because if we add $F[i_1] + F[i_2]$ to both sides of (4.1) we get $G[i_1, j_1] + G[i_2, j_2] \leq G[i_1, j_2] + G[i_2, j_1]$. Thus the SMAWK algorithm works correctly and $H[j]$ is set to the minimum value of the subcolumn $G[c + 1, p - 1; j]$ (which is not necessarily the same as the minimum value of the subcolumn $g(c + 1, p - 1; j)$). Note that since all the entries on and below the main diagonal of g are $+\infty$, they cannot be $H[j]$ for any j and hence have no effect on the computation.

In step 4 we either verify that $F[j] = f(j)$ for all $j \in [c + 2, p]$ (this is the case if $H[j] \geq F[j]$ for all $j \in [c + 2, p]$) or we find the smallest j where this condition fails (this is the case when there exists $j \in [c+2, p]$ such that $H[j] < F[j]$). In either case, the values of c and r are set accordingly at step 5 so that the loop invariants hold. This completes the description of Wilber's algorithm.

Next we discuss how to use Wilber's algorithm to solve the enhanced LWS problem. Let $w[0..n, 0..n]$ be the weight matrix of an instance of the enhanced LWS problem. Let L denote the portion of w consisting of the entries on and below the main diagonal of w. Let w' be the matrix obtained from w by replacing all the entries in L by $+\infty$. Then w' defines an instance of the (ordinary) LWS problem. By

Lemma 4.1, the solution for the problem defined by w' is identical to the solution for the problem defined by w. If each entry of w can be computed in $O(1)$ time, we can use Wilber's algorithm on w' to solve the problem. However, if the enhanced LWS problem is derived from an instance of the SPBD problem, the entries of the matrix $w = A \times B$ cannot be computed in $O(1)$ time. In this case we cannot afford to change w to w' since doing so will destroy some properties of w that are crucial for obtaining a fast algorithm. Fortunately, we will show that Wilber's algorithm can be applied directly to w to solve the enhanced LWS problem.

It is enough to show that the entries in L have no effect on the computation of Wilber's algorithm. The only place where Wilber's algorithm needs the entries in L is step 3, where the SMAWK algorithm is applied to the submatrix T. For each $j \in [c+2, p]$ let $F[j]$ and $H[j]$ be the minimum value of column j in S and T, respectively. There are three cases:

(a) $F[j] \leq H[j]$.

(b) $F[j] > H[j]$ and $H[j]$ is not in L. Namely, $H[j] = G[i,j]$ for some $i < j$.

(c) $F[j] > H[j]$ and $H[j]$ is in L. Namely, $H[j] = G[i,j]$ for some $i \geq j$.

In cases (a) and (b) the values in L do not affect the computation. In the following we show that case(c) cannot occur. Toward a contradiction, assume that there exist indices $j \in [c+2, p]$ and i such that $i \geq j$ and $H[j] = G[i,j] < F[j]$.

Case 1: If $i = j$, then $H[j] = G[j,j] = F[j] + w(j,j) \geq F[j]$. This is impossible.

Case 2: If $i > j$, then $H[j] = G[i,j] = F[i] + w(i,j)$. Recall that $F[i]$ is the minimum value of the subcolumn $G[r,c;i]$. Suppose that $F[i] = G[t,i] = F[t] + w(t,i)$ for some $r \leq t \leq c$. Note that $t \leq c < i$ and $j < i$. By the concavity of w we have $w(t,j) + w(i,i) \leq w(t,i) + w(i,j)$. Since $w(i,i) \geq 0$ for all i, we have $H[j] = F[i] + w(i,j) = F[t] + w(t,i) + w(i,j) \geq F[t] + w(t,j) + w(i,i) \geq F[t] + w(t,j) = G[t,j] \geq F[j]$. This contradicts the assumption that $H[j] < F[j]$.

Since case (c) cannot occur, the entries in L do not affect the computation of Wilber's algorithm, regardless of whether they are changed to $+\infty$ or not. Hence we have proved the following lemma.

LEMMA 4.6. *An instance of the enhanced LWS problem defined by a (full) concave matrix w can be correctly solved by applying Wilber's algorithm to the matrix w.*

Next we address the second difficulty mentioned at the end of section 4.2. If the instance of the enhanced LWS problem is derived from an instance of the SPBD problem (defined by matrices A and B), the weight matrix w of the enhanced LWS problem is the product matrix $w = A \times B$. Therefore the key assumption of Wilber's algorithm that each entry $w[i,j]$ can be evaluated in $O(1)$ time is not valid.

During each stage of Wilber's algorithm (steps 2 and 3) we need to find column minima of the submatrices S and T. Both S and T have the form $C'[r,c;q,p]$ where $C'[i,j] = F[i] + w[i,j]$ for some known value $F[i]$. Since the values $C'[i,j]$ cannot be computed in $O(1)$ time, we cannot use the SMAWK algorithm directly. Instead, we use the algorithm given in the following lemma.

LEMMA 4.7. *The column minima searching problem for the submatrix $C'[r,c;q,p]$ with $r \leq q$ and $c \leq p$ can be solved in $O((c-r) + (p-q) + (k_2 - k_1))$ time, where $k_1 = I(r,r)$ and $k_2 = I(p,p)$.*

Proof. By Lemma 4.3, for each $i \in [r,c]$ and $j \in [q,p]$, $w[i,j] = \min_{0 \leq k \leq m}(A[i,k] + B[k,j])$ can be computed by searching k in the range $k \in [k_1, k_2]$. For $j \in [q,p]$ let $d(j)$ denote the column minimum of $C'[r,c;j]$. Then $d(j) = \min_{r \leq i \leq c}\{F[i] + w[i,j]\} = \min_{r \leq i \leq c}\{F[i] + \min_{k_1 \leq k \leq k_2}(A[i,k] + B[k,j])\} = \min_{k_1 \leq k \leq k_2}\{B[k,j] + \min_{r \leq i \leq c}(F[i] + A[i,k])\}$.

For $i \in [r, c]$ and $k \in [k_1, k_2]$ let $A'[i, k] = F[i] + A[i, k]$. Then A' is totally monotone. For each $k \in [k_1, k_2]$ let $J[k]$ be the minimum of the subcolumn $A'[r, c; k]$.

For $k \in [k_1, k_2]$ and $j \in [q, p]$ let $B'[k, j] = B[k, j] + J[k]$. Then B' is totally monotone. Clearly, $d(j)$ is the minimum of the subcolumn $B'[k_1, k_2; j]$. Thus the column minima $d(j)$'s of $C'[r, c; q, p]$ can be found by two applications of the SMAWK algorithm, once on A' and once on B'. Each entry of A' and B' can be evaluated in $O(1)$ time. Thus the total time needed is $O((c-r)+(k_2-k_1))+O((k_2-k_1)+(p-q)) = O((c-r)+(p-q)+(k_2-k_1))$. $\square$

4.4. The SPBD algorithm and its complexity. The following is the complete description of our SPBD algorithm.

SPBD ALGORITHM.

```
Input: An instance of the SPBD problem defined by two concave matrices $A$ and $B$.
  1. Compute $I(0,0), I(1,1), \ldots, I(n,n)$ (cf. Lemma 4.4).
  2. Solve the enhanced LWS problem defined by the matrix $w = A \times B$ by ap-
     plying Wilber's algorithm on $w$. But instead of using the SMAWK algorithm
     we use the algorithm given in Lemma 4.7 to search the column minima of the
     submatrix $S$ and $T$ during the execution.
  3. Using the method described in Lemma 4.5, convert the solution of the en-
     hanced LWS problem defined by $w$ to the solution of the original SPBD
     problem.
end.
```

The correctness of the algorithm follows from the discussion of the last subsection. Next we analyze the running time of the SPBD algorithm. We concentrate on step 2 since this is the nontrivial part of the algorithm. Each iteration of Wilber's algorithm is completely specified by three parameters: r, c, p. Let r_i, c_i, p_i be the values of these parameters in the ith iteration. The parameters for the next iteration are calculated in step 5 as follows:

Case 1: "then" part of step 5 is executed. In this case, $r_{i+1} = r_i$, $c_{i+1} = p_i$, and

Case 1a: $p_{i+1} = 2c_{i+1} - r_{i+1} + 1$, if it is $\leq n$, or

Case 1b: $p_{i+1} = n$, otherwise.

Case 2: "else" part is executed. In this case, $r_{i+1} = c_i + 1$, $c_{i+1} = j_0$ (where $c_i + 2 \leq j_0 \leq p_i$), and

Case 2a: $p_{i+1} = 2c_{i+1} - r_{i+1} + 1$, if it is $\leq n$, or

Case 2b: $p_{i+1} = n$, otherwise.

If Case 1a (or 1b, 2a, 2b, respectively) applies to the ith iteration, we call it a type 1a (or 1b, 2a, 2b, respectively) iteration. We call $[r_i, p_i]$ the ith *span*; r_i and p_i are the left and the right ends of the ith span, respectively. Note that after a type 1a or 1b iteration, the left end is not changed and the right end increases. After a type 2a or 2b iteration, the left end increases and the right end may increase, decrease, or remain unchanged. For an interval $[t, t+1]$ $(0 \leq t < n)$ we say that a span $[r_i, p_i]$ *covers* $[t, t+1]$, written as $[t, t+1] \in [r_i, p_i]$, if $r_i \leq t$ and $t+1 \leq p_i$. Since the left end of spans never decreases, the spans "move" from left to right during the execution of the algorithm. Once the left end of a span is $\geq t+1$, $[t, t+1]$ will never be covered by any subsequent spans. First we make the following obvious observations:

(a) If a type 1a or 1b iteration follows a type 1b or 2b iteration, the algorithm terminates immediately.

(b) If the ith iteration is of type 1a, then $p_{i+1} - r_{i+1} = (2c_{i+1} - r_{i+1} + 1) - r_{i+1} = 2(p_i - r_i) + 1$. Namely, the length of the $(i+1)$th span is $1 + 2 \times$ (the length of the ith span).

(c) Suppose that the ith iteration is of type 2a or 2b. Since $p_i \leq 2c_i - r_i + 1$, we have $c_i \geq (p_i + r_i - 1)/2$. Hence $r_{i+1} = c_i + 1 \geq (p_i + r_i - 1)/2 + 1$.

(d) Suppose that an interval $[t, t+1]$ is covered by the ith span $[r_i, p_i]$. If the ith iteration is of type 1a or 1b, and the $(i+1)$st iteration is of type 2a or 2b, then $r_{i+2} = c_{i+1} + 1 = p_i + 1 > t + 1$. Hence $[t, t+1]$ is not covered by $[r_{i+2}, p_{i+2}]$ nor by any subsequent spans.

The following lemma gives an upper bound on the number of times an interval $[t, t+1]$ can be covered by spans. This bound is needed in the analysis of our algorithm.

LEMMA 4.8. *Any interval* $[t, t+1]$ $(0 \leq t < n)$ *is covered by at most* $2 \log n + 2$ *spans.*

Proof. Let $[r_{i_1}, p_{i_1}], [r_{i_2}, p_{i_2}], \ldots, [r_{i_k}, p_{i_k}]$ be all the spans covering $[t, t+1]$, where $i_1 < i_2 < \cdots < i_k$. Then $r_{i_l} \leq t$ and $t + 1 \leq p_{i_l}$ for all $1 \leq l \leq k$.

Let l be the first index such that the i_lth iteration is of type 1a or type 1b. (If no such l exists, let $l = k$.) We first show that $k - l \leq \log n + 2$.

Case 1: The i_lth iteration is of type 1b. If the $(i_l + 1)$st iteration is of type 1a or 1b, then the algorithm terminates by observation (a). If the $(i_l + 1)$st iteration is of type 2a or 2b, then by observation (d) $[t, t+1]$ is not covered by $[r_{i_l+2}, p_{i_l+2}]$ nor by any subsequent spans.

Case 2: The i_lth iteration is of type 1a. Let s (possibly $s = 0$) be the largest integer such that the iterations $i_l, i_l + 1, \ldots, i_l + s$ are all of type 1a. Clearly, $[t, t+1]$ is covered by the spans $[r_{i_l+1}, p_{i_l+1}], \ldots, [r_{i_l+s}, p_{i_l+s}]$. By observation (b) each type 1a iteration doubles the length of the span. Since the length of a span is at most n, we have $s \leq \log n$. The $(i_l + s + 1)$st iteration is of type 1b, 2a, or 2b. If it is of type 2a or 2b, then by observation (d) $[t, t+1]$ is not covered by the $(i_l + s + 2)$nd span nor by any subsequent spans. If the $(i_l + s + 1)$st iteration is of type 1b, then, similar to Case 1, either the algorithm terminates at the $(i_l + s + 2)$nd iteration, or $[t, t+1]$ is not covered by the $(i_l + s + 2)$nd span nor by any subsequent spans.

In either case, the number of spans following the i_lth iteration that cover $[t, t+1]$ is at most $\log n + 2$. So $k - l \leq \log n + 2$. Next we show $l \leq \log n$ and this will complete the proof of the lemma.

For each $1 \leq h < l$ the i_hth iteration is of type 2a or 2b. Fix an index h. For each $j \geq i_h$ let $L_j = (t+1) - r_j$. Note that if $L_j \leq 0$, then the span $[r_j, p_j]$ cannot cover the interval $[t, t+1]$. By the fact that $t + 1 \leq p_{i_h}$ and observation (c), we have

$$\begin{aligned} L_{i_h+1} &= (t+1) - r_{i_h+1} \leq (t+1) - ((p_{i_h} + r_{i_h} - 1)/2 + 1) \\ &= (2t - p_{i_h} - r_{i_h} + 1)/2 \leq (t - r_{i_h})/2 < L_{i_h}/2. \end{aligned}$$

Since the left end of the spans never decreases, we now have that $L_{i_{h+1}} \leq L_{i_h+1} < L_{i_h}/2$. This is true for all $1 \leq h < l$. Hence $L_{i_l} < L_{i_1}/2^l$. If $l > \log n$, then L_{i_l} becomes 0 and the interval $[t, t+1]$ is not covered by $[r_{i_l}, p_{i_l}]$ nor by any subsequent spans. So we must have $l \leq \log n$. This establishes the lemma. ☐

We are now ready to prove Theorem 1.1.

Proof of Theorem 1.1. Step 1 of the SPBD algorithm takes $O(m \log n)$ time by Lemma 4.4. In step 2 we use Wilber's algorithm to solve the enhanced LWS problem defined by the matrix $w = A \times B$. But instead of using the SMAWK algorithm we use the subroutine in Lemma 4.7 for finding column minima in S and T. Note that all the other steps of Wilber's algorithm take $O(n + m)$ time. Thus if we can show that the time needed by these subroutine calls is bounded by $O(n + m \log n)$, the theorem will follow from Lemma 4.5.

Consider the ith iteration. We need to find the column minima of the submatrices $S_i = G[r_i, c_i; c_i + 1, p_i]$ and $T_i = G[c_i + 1, p_i - 1; c_i + 2, p_i]$. Let $k_1 = I(r_i, r_i)$, $k_2 = I(p_i, p_i)$, and $k_3 = I(c_i + 1, c_i + 1)$. Since $r_i < c_i + 1 \leq p_i$ we have $k_1 \leq k_3 \leq k_2$ by Lemma 4.3.

By Lemma 4.7 the searching of S_i needs $O((c_i - r_i) + (p_i - c_i - 1) + (k_2 - k_1)) = O((p_i - r_i) + (k_2 - k_1))$ time. The searching of T_i needs $O((p_i - 1 - c_i - 1) + (p_i - c_i - 2) + (k_2 - k_3))$ time. Since $p_i \leq 2c_i - r_i + 1$ and $k_3 \geq k_1$, this is bounded by $O((p_i - r_i) + (k_2 - k_1))$. Thus the total time needed to search S_i and T_i in all the iterations is $\sum_{i=1}^{K} O((p_i - r_i) + (I(p_i, p_i) - I(r_i, r_i)))$, where K is the total number of iterations of the algorithm. Since Wilber's original algorithm takes $O(n)$ time, the term $\sum_{i=1}^{K} O(p_i - r_i)$ is bounded by $O(n)$. On the other hand,

$$\begin{aligned}\sum_{i=1}^{K}(I(p_i, p_i) - I(r_i, r_i)) &= \sum_{i=1}^{K}\sum_{t=r_i}^{p_i-1}(I(t+1, t+1) - I(t, t)) \\ &= \sum_{t,i \text{ where } [t,t+1]\in[r_i,p_i]} (I(t+1, t+1) - I(t, t)). \qquad (4.3)\end{aligned}$$

By Lemma 4.8 each interval $[t, t + 1]$ is covered by at most $2 \log n + 2$ spans. Thus the above sum is bounded by $O(\log n \sum_{t=0}^{n-1}(I(t + 1, t + 1) - I(t, t))) = O(m \log n)$ as to be shown. $\square$

5. Conclusion.

We introduced the SPBD problem and showed that if the weight matrices are concave, then the SPBD problem can be reduced to the enhanced LWS problem and solved in $O(n+m \log n)$ time. As applications, we showed that the MLP for points on a straight line and the TSP for points on a convex polygon can be reduced to the SPBD problem and solved in $O(n \log n)$ time, which substantially improves the previously known $O(n^2)$-time algorithms. The setting of the SPBD problem is quite general. It is interesting to find other applications of the SPBD problem.

We tried (but failed) to use this technique to solve the MLP for points on a convex polygon. (To our knowledge, the MLP for this special case is not known to be in P.) In the two applications discussed in this paper, the optimal paths are simple (i.e., no two edges of the path cross). Unfortunately, the optimal tour in the MLP for points on a convex polygon does not have this crucial property. It would be interesting to find a polynomial-time algorithm for solving the MLP for this case.

Another open problem is to solve the MLP for an *r-arm star graph* G, which has a center vertex c and r "arms" connected to c, and each arm is a straight line with several vertices on it. (Thus the straight line discussed in section 3 is a two-arm star graph.) If each arm of G contains at most k vertices, then the MLP problem for G can be solved in $O(r \times k^r)$ time by using dynamic programming, which is not a polynomial in terms the number of vertices of G. Is it possible to solve this problem in polynomial time using ideas similar to the SPBD algorithm? This would seem to require, at the very least, solving r-partite generalization of the SPBD problem.

Acknowledgments. The authors would like to thank the referees for pointing out the much simplified proof of Lemma 3.1, for mentioning the last open problem in section 5, and for suggestions that considerably improved the exposition.

REFERENCES

[1] F. Afrati, S. Cosmadakis, C. Papadimitriou, G. Papageorgiou, and N. Papkostantinou, *The complexity of the traveling repairman problem*, Informatique Theorique Appl. (Theoret. Informatics Appl.) 20 (1986), pp. 79–87.

[2] A. Aggarwal and J. Park, *Notes on searching in multidimensional monotone arrays*, in Proc. 29th IEEE Foundations of Computer Science, White Plains, NY, 1988, pp. 497–512.

[3] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, *Geometric applications of a matrix searching algorithm*, Algorithmica, 2 (1987), pp. 195–208.

[4] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng, *Constructing trees in parallel*, in Proc. ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, 1989, pp. 421–431.

[5] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan, *The minimum latency problem*, in Proc. 26th ACM Symposium on the Theory of Computing, Montreal, Quebec, 1994, pp. 163–171.

[6] D. Eppstein, *Sequence comparison with mixed convex and concave costs*, J. Algorithms, 11 (1990), pp. 85–101.

[7] Z. Galil and R. Giancarlo, *Speeding-up dynamic programming with applications to molecular biology*, Theoret. Comput. Sci., 64 (1989), pp. 107–118.

[8] D. S. Hirschberg and L. L. Larmore, *The least weight subsequence problem*, SIAM J. Comput., 16 (1987), pp. 628–638.

[9] M. M. Klawe and D. J. Kleitman, An almost linear time algorithm for generalized matrix searching, SIAM J. Discrete Math., 3 (1990), pp. 81–97.

[10] O. Marcotte and S. Suri, *Fast matching algorithms for points on a polygon*, SIAM J. Comput., 20 (1991), pp. 405–422.

[11] R. Wilber, *The concave least-weight subsequence problem revisited*, J. Algorithms, 9 (1988), pp. 418–425.

[12] F. F. Yao, *Efficient dynamic programming using quadrangle inequalities*, in Proc. 12th ACM Symposium on the Theory of Computing, 1980, pp. 429–435.

[13] F. F. Yao, *Speed-up in dynamic programming*, SIAM J. Alg. Discrete Methods, 3 (1982), pp. 532–540.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 81–102

REGULAR CLOSURE OF DETERMINISTIC LANGUAGES[*]

EBERHARD BERTSCH[†] AND MARK-JAN NEDERHOF[‡]

Abstract. We recall the notion of regular closure of classes of languages. We present two important results. The first result is that all languages which are in the regular closure of the class of deterministic (context-free) languages can be recognized in linear time. This is a nontrivial result, since this closure contains many inherently ambiguous languages. The second result is that the class of deterministic languages is contained in the closure of the class of deterministic languages with the prefix property or, stated in an equivalent way, all LR(k) languages are in the regular closure of the class of LR(0) languages.

Key words. context-free languages, regular languages

AMS subject classifications. 68Q45, 68Q15

PII. S009753979528682X

1. Introduction.

Despite important insights in applying sophisticated matrix operations to recognition techniques [16], the known upper bound on the time complexity of context-free language recognition still exceeds $\mathcal{O}(n^2)$, measured in the length of the input string. However, there are many languages whose time complexity has been shown to be linear by means of *specialized* algorithms but which are not recognized in linear time by *general* recognition algorithms. The frontier of knowledge in this area is advanced by the results presented in this article.

We introduce a special class of two-level automata. Their upper level is constituted by a (classical) finite automaton whose transitions are, however, not enabled by a single terminal symbol but by any element of a given (lower-level) language. All languages at the lower level are assumed to have the restrictive LR(k)-property. Due to the well-known correspondence between regular expressions and finite automata, the new class of languages may thus be stated to result from the set of deterministic (i.e., LR(k)) languages by recursively applying concatenation, union, and Kleene star to given languages in that class.

The new class contains some notorious specimens such as $\{a^m b^m c^n\} \cup \{a^m b^n c^n\}$, an inherently ambiguous language [8]. As will be developed in the body of this article, a linear upper time bound on recognizing such languages can be established.

The results are related to our previous work [13] on efficient recognition of language suffixes. It had in turn been motivated by practical syntax error detection (more precisely, by noncorrecting error recovery [15]).

The article may be outlined as follows. After introducing the definitional framework in section 2, we show that all deterministic context-free languages can be constructed from prefix-free deterministic languages by regular operations (section 3).

[*]Received by the editors May 26, 1995; accepted for publication (in revised form) March 28, 1997; published electronically September 14, 1999. Portions of this paper appear in a modified form as *An innovative finite state concept for recognition and parsing of context-free languages*, in Extended Finite State Models of Language, András Kornai, ed., Cambridge University Press, Cambridge, UK, 1997, an abstract of which appeared in *Natural Language Engineering*, 2 (1997), pp. 383–384.

http://www.siam.org/journals/sicomp/29-1/28682.html

[†]Ruhr University Bochum, Bochum, Germany (bertsch@lpi.ruhr-uni-bochum.de).

[‡]DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany (nederhof@dfki.de). The research of this author was supported by Dutch Organization for Scientific Research (NWO) grant 305-00-802 while he was employed at the University of Groningen, Groningen, the Netherlands.

This follows from a detailed decomposition of pushdown computations into sequences of moves that do not discard more than one element of the stack they started with.

The most important proposition of section 4 is that each deterministic pushdown automaton can be transformed into an equivalent one which is "loop-free." This requires a fairly deep discussion of individual pushing and popping moves. In essence, automata are changed in such a way that the stack contents after certain moves must reflect the *amount* of processed input. Using tabulation, the transformed pushdown automata can be simulated at all input positions in linear time. This result is part of a proof of the fact that languages in the regular closure of the deterministic languages can be recognized in linear time by means of a two-level device, to be defined in section 5.

Section 6 deals with various syntheses and applications of the obtained results. In particular, "on-line" and "off-line" variants of the two-level device are presented and compared. The difference between these notions is rather similar to the one between Earley's recognition algorithm [7] and the Cocke–Kasami–Younger technique [8]. With the latter approach, partial recognition results are collected without reference to their left context.

Although the parse tree concept is less immediate for the new kind of language description than for ordinary grammars, we are able to sketch an efficient transduction procedure yielding representations of the syntactic structure of given inputs (section 7).

Two applications are presented in section 8. First we prove that suffix recognition is possible in linear time. This new proof is much shorter than recently published proofs of this fact. We then describe an application in pattern matching.

2. Notation. A finite automaton $\mathcal{F}$ is a 5-tuple (S, Q, q_s, F, T), where S and Q are finite sets of input symbols and states, respectively; $q_s \in Q$ is the *initial* state and $F \subseteq Q$ is the set of *final* states; and the transition relation T is a subset of $Q \times S \times Q$.

An input string $b_1 \cdots b_m \in S^*$ is *recognized* by the finite automaton if there is a sequence of states $q_0, q_1, \ldots, q_m$ such that $q_0 = q_s$, $(q_{k-1}, b_k, q_k) \in T$ for $1 \leq k \leq m$, and $q_m \in F$. For a certain finite automaton $\mathcal{F}$, the set of all such strings w is called the language *accepted by* $\mathcal{F}$, denoted $L(\mathcal{F})$. The languages accepted by finite automata are called the *regular* languages.

In the following, we describe a type of pushdown automaton without internal states and with very simple kinds of transition. This is a departure from the standard literature (e.g., [9]), but simplifies considerably our definitions and proofs in the remainder of the paper. That the generative capacity of this type of pushdown automaton is not affected with respect to any of the more traditional types can be argued by the fact that every context-free language is accepted by a nondeterministic LR recognizer of a form very similar to our type of pushdown automaton [14]. See also [13].

Thus, we define a pushdown automaton (PDA) $\mathcal{A}$ to be a 5-tuple $(\Sigma, \Delta, X_{initial}, F, T)$, where Σ, Δ, and T are finite sets of input symbols, stack symbols and transitions, respectively; $X_{initial} \in \Delta$ is the *initial* stack symbol, and $F \subseteq \Delta$ is the set of *final* stack symbols.

We consider a fixed input string $a_1 \cdots a_n \in \Sigma^*$. A *configuration* of the automaton is a pair (δ, v) consisting of a stack $\delta \in \Delta^*$ and the remaining input v, which is a suffix of the original input string $a_1 \cdots a_n$.

The *initial* configuration is of the form $(X_{initial}, a_1 \cdots a_n)$, where the stack is formed by the initial stack symbol $X_{initial}$. A *final* configuration is of the form $(\delta X, \epsilon)$, where the element on top of the stack is some final stack symbol $X \in F$.

The transitions in T are of the form $X \overset{z}{\mapsto} XY$, where $z = \epsilon$ or $z = a$, or of the form $XY \overset{\epsilon}{\mapsto} Z$.

The application of such a transition $\delta_1 \overset{z}{\mapsto} \delta_2$ is described as follows. If the topmost symbols on the stack are δ_1, then these may be replaced by δ_2, provided either $z = \epsilon$ or $z = a$ and a is the first symbol of the remaining input. If $z = a$ then furthermore a is removed from the remaining input.

Formally, for a fixed PDA we define the binary relation $\vdash$ on configurations as the least relation satisfying $(\delta\delta_1, v) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \overset{\epsilon}{\mapsto} \delta_2$ and $(\delta\delta_1, av) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \overset{a}{\mapsto} \delta_2$.

In the case that we consider more than one PDA at the same time we use symbols $\overset{z}{\mapsto}_{\mathcal{A}}$ and $\vdash_{\mathcal{A}}$ instead of $\overset{z}{\mapsto}$ and $\vdash$ if these refer to one particular PDA $\mathcal{A}$.

The recognition of a certain input v is obtained if, starting from the initial configuration for that input, we can reach a final configuration by repeated application of transitions, or, formally, if $(X_{initial}, v) \vdash^* (\delta X, \epsilon)$, with some $\delta \in \Delta^*$ and $X \in F$, where $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$ (and $\vdash^+$ denotes the transitive closure of $\vdash$). For a certain PDA $\mathcal{A}$, the set of all such strings v which are recognized is called the language *accepted by* $\mathcal{A}$, denoted $L(\mathcal{A})$. A PDA is called *deterministic* if, for all possible configurations, at most one transition is applicable. The languages accepted by deterministic PDAs (DPDAs) are called *deterministic languages*.

We may restrict DPDAs such that no transitions apply to final configurations, by imposing $X \notin F$ if there is a transition $X \overset{z}{\mapsto} XY$ and $Y \notin F$ if there is a transition $XY \overset{\epsilon}{\mapsto} Z$. We call such a DPDA *prefix-free*. The languages accepted by such DPDAs are obviously *prefix-free*, which means that no string in the language is a prefix of any other string in the language. Conversely, any prefix-free deterministic language is accepted by some prefix-free DPDA, the proof being that in a DPDA, all transitions of the form $X \overset{z}{\mapsto} XY$, $X \in F$, and $XY \overset{\epsilon}{\mapsto} Z$, $Y \in F$, can be removed without consequence to the accepted language if this language is prefix-free.

In compiler design, the deterministic languages are better known as LR(k) languages, and the prefix-free deterministic languages as LR(0) languages [9].

A prefix-free DPDA is in normal form if, for all input v, $(X_{initial}, v) \vdash^* (\delta X, \epsilon)$, with $X \in F$, implies $\delta = \epsilon$, and furthermore F is a singleton $\{X_{final}\}$. Any prefix-free DPDA can be put into normal form. (See [9, Theorem 5.1] for a proof of a related result.) We define a *normal PDA* (NPDA) to be a prefix-free deterministic PDA in normal form.

We define a subrelation $\models^+$ of $\vdash^+$ as $(\delta, vw) \models^+ (\delta\delta', w)$ if and only if $(\delta, vw) = (\delta, z_1 z_2 \cdots z_m w) \vdash (\delta\delta_1, z_2 \cdots z_m w) \vdash \cdots \vdash (\delta\delta_m, w) = (\delta\delta', w)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all k, $1 \leq k \leq m$. Informally, we have $(\delta, vw) \models^+ (\delta\delta', w)$ if configuration $(\delta\delta', w)$ can be reached from (δ, vw) without the bottommost part δ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of δ. Note that $(\delta_1 X, vw) \models^+ (\delta_1 X\delta', w)$ implies $(\delta_2 X, vw') \models^+ (\delta_2 X\delta', w')$ for any δ_2 and any w', since the transitions neither address the part of the stack below X nor read the input following v.

3. Metadeterministic languages.

In this section we define a new subclass of the context-free languages, which results from combining deterministic languages by the operations used to specify regular languages.

We first define the concept of *regular closure* of a class of languages.[1] Let $\mathcal{L}$ be a

[1]This notion was called *rational closure* in [3].

class of languages. The regular closure of $\mathcal{L}$, denoted $C(\mathcal{L})$, is defined as the smallest class of languages such that

(i) $\emptyset \in C(\mathcal{L})$,
(ii) if $l \in \mathcal{L}$ then $l \in C(\mathcal{L})$,
(iii) if $l_1, l_2 \in C(\mathcal{L})$ then $l_1 l_2 \in C(\mathcal{L})$,
(iv) if $l_1, l_2 \in C(\mathcal{L})$ then $l_1 \cup l_2 \in C(\mathcal{L})$, and
(v) if $l \in C(\mathcal{L})$ then $l^* \in C(\mathcal{L})$.

Note that a language in $C(\mathcal{L})$ may be described by a regular expression over symbols representing languages in $\mathcal{L}$.

Let $\mathcal{D}$ denote the class of deterministic languages. Then the class of *metadeterministic* languages is defined to be its regular closure, $C(\mathcal{D})$. This class is obviously a subset of the class of context-free languages, since the class of context-free languages is closed under concatenation, union, and Kleene star, and it is a *proper* subset, since, for example, the context-free language $\{ww^R \mid w \in \{a,b\}^*\}$ is not in $C(\mathcal{D})$. (w^R denotes the mirror image of w.)

Finite automata constitute a computational representation for regular languages; DPDAs constitute a computational representation for deterministic languages. By combining these two mechanisms we obtain the metadeterministic automata, which constitute a computational representation for the metadeterministic languages.

Formally, a metadeterministic automaton $\mathcal{M}$ is a triple $(\mathcal{F}, A, \mu)$, where $\mathcal{F} = (S, Q, q_s, F, T)$ is a finite automaton, A is a finite set of DPDAs with identical alphabets Σ, and μ is a mapping from S to A.

The language accepted by such a device is composed of languages accepted by the DPDAs in A according to the transitions of the finite automaton $\mathcal{F}$. Formally, a string v is *recognized by* automaton $\mathcal{M}$ if there is some string $b_1 \cdots b_m \in S^*$, a sequence of PDAs $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m \in A$, and a sequence of strings $v_1, \ldots, v_m \in \Sigma^*$ such that

(i) $b_1 \cdots b_m \in L(\mathcal{F})$,
(ii) $\mathcal{A}_k = \mu(b_k)$ for $1 \leq k \leq m$,
(iii) $v_k \in L(\mathcal{A}_k)$ for $1 \leq k \leq m$, and
(iv) $v = v_1 \cdots v_m$.

The set of all strings recognized by automaton $\mathcal{M}$ is called the language *accepted by* $\mathcal{M}$, denoted $L(\mathcal{M})$.

EXAMPLE 3.1. *As a simple example of a language accepted by a metadeterministic automaton, consider $L = L_1 \cup L_2$, where $L_1 = \{a^m b^n c^n \mid n, m \in \{0, 1, \ldots\}\}$ and $L_2 = \{a^m b^m c^n \mid n, m \in \{0, 1, \ldots\}\}$. It is well established that L is not a deterministic language* [9, *Example* 10.1]. *However, it is the union of two languages L_1 and L_2, which are by themselves deterministic. Therefore, L is accepted by a metadeterministic automaton $\mathcal{M}$ which uses two DPDAs $\mathcal{A}_1$ and $\mathcal{A}_2$, accepting L_1 and L_2, respectively.*

We may, for example, define $\mathcal{M}$ as $(\mathcal{F}, \{\mathcal{A}_1, \mathcal{A}_2\}, \mu)$ with $\mathcal{F} = (S, Q, q_s, F, T)$, where

(i) $S = \{b_1, b_2\}$,
(ii) $Q = \{q_s, q_f\}$,
(iii) $F = \{q_f\}$,
(iv) $T = \{(q_s, b_1, q_f),\ (q_s, b_2, q_f)\}$, *and*
(v) $\mu(b_1) = \mathcal{A}_1$ *and* $\mu(b_2) = \mathcal{A}_2$.

A graphic representation for $\mathcal{M}$ is given in Figure 3.1. *States $q \in Q$ are represented by vertices labeled q and triples $(q, b, p) \in T$ are represented by arrows from q to p labeled $\mu(b)$.* □

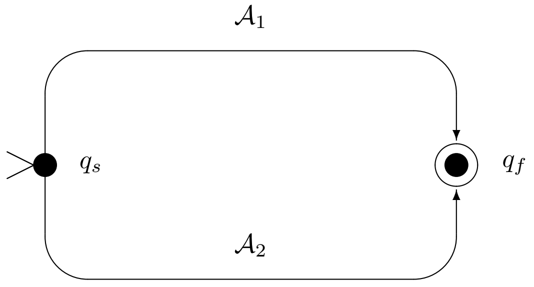


FIG. 3.1. *A metadeterministic automaton.*

That the metadeterministic automata precisely accept the metadeterministic languages is reflected by the following equation

$$C(\mathcal{D}) = \{L(\mathcal{M}) \mid \mathcal{M} \text{ is a metadeterministic automaton}\}.$$

This equation straightforwardly follows from the equivalence of finite automata and regular expressions and the equivalence of DPDAs and deterministic languages.

Let $\mathcal{N}$ denote the class of prefix-free deterministic languages. In the same vein, we have

$$C(\mathcal{N}) = \{L(\mathcal{M}) \mid \mathcal{M} = (\mathcal{F}, A, \mu) \text{ is a metadeterministic automaton where } A \text{ is a set of NPDAs}\}.$$

In the following discussion, we set out to prove a number of properties of languages in $C(\mathcal{D})$, represented by their metadeterministic automata (i.e., their corresponding recognition devices). The DPDAs in an arbitrary such device cause some technical difficulties which may be avoided if we restrict ourselves to metadeterministic automata which use only normal PDAs, as opposed to arbitrary DPDAs. Fortunately, this restriction does not reduce the class of languages that can be described, or in other words, $C(\mathcal{N}) = C(\mathcal{D})$. We prove this equality below.

Since $C(\mathcal{N}) \subseteq C(\mathcal{D})$ is vacuously true, it is sufficient to argue that $\mathcal{D} \subseteq C(\mathcal{N})$, from which $C(\mathcal{D}) \subseteq C(C(\mathcal{N})) = C(\mathcal{N})$ follows, using the closure properties of C—in particular, monotonicity and idempotence.

We prove that $\mathcal{D} \subseteq C(\mathcal{N})$ by showing how for each DPDA $\mathcal{A}$ a metadeterministic automaton $\rho(\mathcal{A}) = (\mathcal{F}, A, \mu)$ may be constructed such that A consists only of prefix-free DPDAs and $L(\rho(\mathcal{A})) = L(\mathcal{A})$. This construction follows.

CONSTRUCTION 1 (DPDA to metadeterministic automaton). *Let* $\mathcal{A} = (\Sigma, \Delta, X_{initial}, F_{\mathcal{A}}, T_{\mathcal{A}})$ *be a deterministic PDA. Construct the metadeterministic automaton* $\rho(\mathcal{A}) = (\mathcal{F}, A, \mu)$, *with* $\mathcal{F} = (S, Q, q_s, F_{\mathcal{F}}, T_{\mathcal{F}})$, *where*

(i) $S = \{b_{X,Y} \mid X, Y \in \Delta\} \cup \{c_{X,Y} \mid X, Y \in \Delta\}$,
(ii) $Q = \Delta$,
(iii) $q_s = X_{initial}$,
(iv) $F_{\mathcal{F}} = F_{\mathcal{A}}$,
(v) $T_{\mathcal{F}} = \{(X, b_{X,Y}, Y) \mid X, Y \in \Delta\} \cup \{(X, c_{X,Y}, Y) \mid X, Y \in \Delta\}$.

The set A *consists of (prefix-free deterministic) PDAs* $\mathcal{B}_{X,Y}$ *and* $\mathcal{C}_{X,Y}$ *for all* $X, Y \in \Delta$, *defined as follows.*

Each $\mathcal{B}_{X,Y}$ is defined to be $(\Sigma, \{X^{in}, Y^{out}\}, X^{in}, \{Y^{out}\}, T)$, where X^{in} and Y^{out} are fresh symbols and the transitions in T are

$$X^{in} \overset{z}{\mapsto}_{\mathcal{B}_{X,Y}} X^{in}Y^{out} \quad \textit{for all } X \overset{z}{\mapsto}_{\mathcal{A}} XY, \textit{some } z.$$

Each $\mathcal{C}_{X,Y}$ is defined to be $(\Sigma, \Delta \cup \{X^{in}, Y^{out}\}, X^{in}, \{Y^{out}\}, T)$, where X^{in} and Y^{out} are fresh symbols and the transitions in T are those in $T_{\mathcal{A}}$ plus the extra transitions

$$\begin{array}{lll} X^{in} & \overset{z}{\mapsto}_{\mathcal{C}_{X,Y}} X^{in}Z & \textit{for all } X \overset{z}{\mapsto}_{\mathcal{A}} XZ, \textit{ some } z \textit{ and } Z; \\ X^{in}Z & \overset{\epsilon}{\mapsto}_{\mathcal{C}_{X,Y}} Y^{out} & \textit{for all } XZ \overset{\epsilon}{\mapsto}_{\mathcal{A}} Y, \textit{ some } Z. \end{array}$$

The function μ maps the symbols $b_{X,Y}$ to automata $\mathcal{B}_{X,Y}$ and the symbols $c_{X,Y}$ to automata $\mathcal{C}_{X,Y}$.

Each automaton $\mathcal{B}_{X,Y}$ mimics a single transition of $\mathcal{A}$ of the form $X \overset{z}{\mapsto}_{\mathcal{A}} XY$. Formally, $\mathcal{B}_{X,Y}$ recognizes a string z if and only if $(X, z) \vdash_{\mathcal{A}} (XY, \epsilon)$.

Each automaton $\mathcal{C}_{X,Y}$ mimics a computation of $\mathcal{A}$ that replaces stack element X with stack element Y. Formally, $\mathcal{C}_{X,Y}$ recognizes a string v if and only if $(X, v) \models^{+}_{\mathcal{A}} (XZ, \epsilon) \vdash_{\mathcal{A}} (Y, \epsilon)$ for some $Z \in \Delta$.

For the proof, consider that recognition of v by $\mathcal{C}_{X,Y}$ means that $(X^{in}, v) \models^{+}_{\mathcal{C}_{X,Y}} (X^{in}Z, \epsilon) \vdash_{\mathcal{C}_{X,Y}} (Y^{out}, \epsilon)$ for some Z, due to the nature of its transitions. This is equivalent to $(X^{in}, z) \vdash_{\mathcal{C}_{X,Y}} (X^{in}W, \epsilon)$, $(W, v') \vdash^{*}_{\mathcal{C}_{X,Y}} (Z, \epsilon)$, $(X^{in}Z, \epsilon) \vdash_{\mathcal{C}_{X,Y}} (Y^{out}, \epsilon)$, and $v = zv'$, for some z, v', and W, due to the definition of $\models^{+}$. This is again equivalent to $(X, z) \vdash_{\mathcal{A}} (XW, \epsilon)$, $(W, v') \vdash^{*}_{\mathcal{A}} (Z, \epsilon)$, $(XZ, \epsilon) \vdash_{\mathcal{A}} (Y, \epsilon)$, and $v = zv'$, by virtue of the construction of $\mathcal{C}_{X,Y}$ from $\mathcal{A}$. Finally, this conjunction is equivalent to $(X, v) \models^{+}_{\mathcal{A}} (XZ, \epsilon) \vdash_{\mathcal{A}} (Y, \epsilon)$.

Note that the languages recognized by some of the $\mathcal{B}_{X,Y}$ and $\mathcal{C}_{X,Y}$ may be the empty set.

That all $\mathcal{B}_{X,Y}$ and $\mathcal{C}_{X,Y}$ are deterministic follows from the fact that by assumption $\mathcal{A}$ is deterministic. That all $\mathcal{B}_{X,Y}$ and $\mathcal{C}_{X,Y}$ are prefix-free follows from the fact that no transitions apply when a final stack symbol is on top of the stack.

To prove that $L(\rho(\mathcal{A})) = L(\mathcal{A})$ we have to show the following.

1. Any sequence of transitions of the form $(X_{initial}, a_1 \cdots a_n) \vdash^{*}_{\mathcal{A}} (\delta X, \epsilon)$, with $X \in F_{\mathcal{A}}$, can be decomposed into a list of m sequences of transitions $(X^{in}_{k-1}, v_k) \vdash^{*}_{\mathcal{A}_k} (\delta_k X^{out}_k, \epsilon)$, $1 \leq k \leq m$, $X_0 = X_{initial}$ and $X_m = X$, using a list of m automata $\mathcal{A}_1, \ldots, \mathcal{A}_m \in A$ that recognize strings $v_1, \ldots, v_m$, respectively, where $a_1 \cdots a_n = v_1 \cdots v_m$ and such that the string $b_1 \cdots b_m$ with $\mu(b_k) = \mathcal{A}_k$ for $1 \leq k \leq m$ is recognized by $\mathcal{F}$.

2. Conversely, if we have a string $b_1 \cdots b_m$ recognized by $\mathcal{F}$, then we must show that any list of m sequences of transitions $(X^{in}_{k-1}, v_k) \vdash^{*}_{\mathcal{A}_k} (\delta_k Y^{out}_k, \epsilon)$ for automata $\mathcal{A}_k = \mu(b_k)$, $1 \leq k \leq m$, can be composed into a single sequence $(X_0, v_1 \cdots v_m) \vdash^{*}_{\mathcal{A}} (\delta Y_m, \epsilon)$, with $X_0 = X_{initial}$ and $Y_m \in F_{\mathcal{A}}$, using $X_k = Y_k$, $1 \leq k \leq m$.

The intuition behind decomposing a sequence of transitions for DPDA $\mathcal{A}$ is conveyed by Figure 3.2. We see the development of the stack, of which the height alternately increases and decreases during performance of the transitions. We assume the input is recognized at t9, where some final stack symbol X_9 is on top of the stack. We now locate the point in time t8 where the stack was of the same height for the last time before t9. Assume that at t8 some stack symbol X_8 is on top of the stack. The sequence of transitions from t8 to t9 is of the form $(X_8, v) \models^{+}_{\mathcal{A}} (X_8 Z, \epsilon) \vdash_{\mathcal{A}} (X_9, \epsilon)$ for some $Z \in \Delta$, which means that the stack development can be mimicked by the PDA

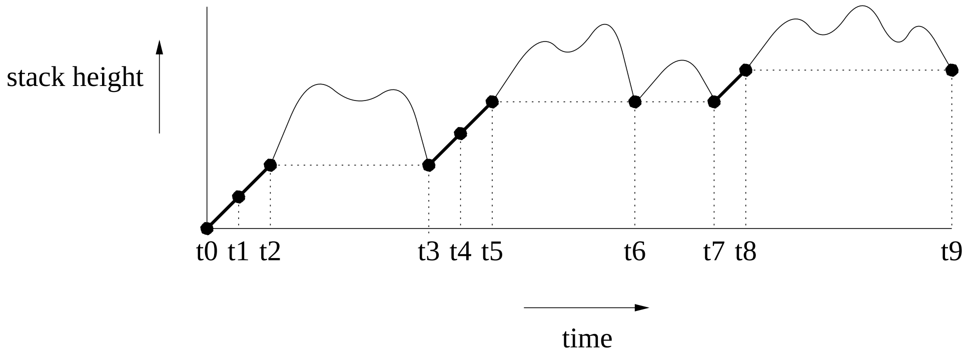


FIG. 3.2. *A stack development of a DPDA $\mathcal{A}$ on input $v_1 \cdots v_9$.*

$\mathcal{C}_{X_8,X_9}$. In Figure 3.2 the step between t7 and t8 is of the form $(X_7, z) \vdash_{\mathcal{A}} (X_7X_8, \epsilon)$, which is mimicked by the PDA $\mathcal{B}_{X_7,X_8}$.

This can be continued until the complete sequence of transitions has been decomposed into a list of nine sequences which are mimicked by PDAs of the form $\mathcal{B}_{X_{k-1},X_k}$ or $\mathcal{C}_{X_{k-1},X_k}$. The corresponding string $b_1 \cdots b_9$, where $b_k = b_{X_{k-1},X_k}$ or $b_k = c_{X_{k-1},X_k}$, $1 \leq k \leq 9$, then allows state $X_9 \in F_{\mathcal{F}} = F_{\mathcal{A}}$ to be reached from state q_s, or, in other words, this string is recognized by finite automaton $\mathcal{F}$.

The proof of the general case uses induction on t. We show that if we have a sequence of transitions

$$(X_0, z_1z_2 \cdots z_t) \vdash_{\mathcal{A}} (\delta_1X_1, z_2 \cdots z_t) \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} (\delta_{t-1}X_{t-1}, z_t) \vdash_{\mathcal{A}} (\delta_tX_t, \epsilon),$$

then for some m there is

(i) a list $Y_0, \ldots, Y_m \in \Delta$, with $Y_0 = X_0$ and $Y_m = X_t$,

(ii) a list $b_1, \ldots, b_m \in S$, such that (Y_0, b_1, Y_1), (Y_1, b_2, Y_2), ..., $(Y_{m-1}, b_m, Y_m) \in T$, and

(iii) a list $v_1, \ldots, v_m \in \Sigma^*$, such that $v_1 \cdots v_m = z_1 \cdots z_t$ and v_k is recognized by PDA $\mu(b_k)$, for $1 \leq k \leq m$.

The case that $t = 0$ can trivially be solved with $m = 0$; for $t > 0$ we distinguish between two cases:

(i) $|\delta_{t-1}X_{t-1}| < |\delta_tX_t|$, or, in other words, the last step used a pushing transition; or

(ii) $|\delta_{t-1}X_{t-1}| > |\delta_tX_t|$, or, in other words, the last step used a popping transition.

In the first case we may assume by definition that the automaton $\mathcal{B}_{X_{t-1},X_t}$ recognizes z_t. The induction hypothesis for $t-1$, applied to $(X_0, z_1z_2 \cdots z_{t-1}) \vdash_{\mathcal{A}}^* (\delta_{t-1}X_{t-1}, \epsilon)$, provides the required three lists with some $m-1$ instead of m. We set $Y_m = X_t$, $b_m = b_{X_{t-1},X_t}$ (so that $\mu(b_m) = \mathcal{B}_{X_{t-1},X_t}$), $v_m = z_t$, which gives us the required three lists for t. Note that $(X_0, z_1 \cdots z_{t-1}) \vdash_{\mathcal{A}}^* (\delta_{t-1}X_{t-1}, \epsilon)$ and $(X_0, z_1 \cdots z_{t-1}z_t) \vdash_{\mathcal{A}}^* (\delta_{t-1}X_{t-1}, z_t)$ are equivalent.

In the second case we may assume that there is a maximal $t' < t$ such that $|\delta_{t'}X_{t'}| = |\delta_tX_t|$. Note that then $|\delta_{t''}X_{t''}| > |\delta_tX_t|$, for $t' < t'' < t$, which means we have $(\delta_{t'}X_{t'}, z_{t'+1} \cdots z_t) \models_{\mathcal{A}}^+ (\delta_{t'}X_{t'}X_{t-1}, \epsilon) \vdash_{\mathcal{A}} (\delta_{t'}X_t, \epsilon)$, which is equivalent to $(X_{t'}, z_{t'+1} \cdots z_t) \models_{\mathcal{A}}^+ (X_{t'}X_{t-1}, \epsilon) \vdash_{\mathcal{A}} (X_t, \epsilon)$. By definition we get that the automaton $\mathcal{C}_{X_{t'},X_t}$ recognizes $z_{t'+1} \cdots z_t$. The induction hypothesis for t', applied to

$(X_0, z_1z_2\cdots z_{t'}) \vdash^*_{\mathcal{A}} (\delta_{t'}X_{t'}, \epsilon)$, provides the required three lists with some $m-1$ instead of m. We set $Y_m = X_t$, $b_m = c_{X_{t'},X_t}$ (so that $\mu(b_m) = \mathcal{C}_{X_{t'},X_t}$), $v_m = z_{t'+1}\cdots z_t$, which gives us the required three lists for t. Note that $(X_0, z_1\cdots z_{t'}) \vdash^*_{\mathcal{A}} (\delta_{t'}X_{t'}, \epsilon)$ and $(X_0, z_1\cdots z_{t'}z_{t'+1}\cdots z_t) \vdash^*_{\mathcal{A}} (\delta_{t'}X_{t'}, z_{t'+1}\cdots z_t)$ are equivalent.

We now give a proof of the converse, viz. that, if we have a string $b_1\cdots b_m \in L(\mathcal{F})$, then a list of m sequences of transitions for automata $\mathcal{A}_k = \mu(b_k)$ recognizing strings v_k, $1 \leq k \leq m$, can be composed into a single sequence for $\mathcal{A}$ recognizing $v_1\cdots v_m$.

Because of the definition of $\rho(\mathcal{A})$, this list of m sequences consists of sequences of the form $(X^{in}_{k-1}, v_k) \vdash^*_{\mathcal{A}_k} (\delta'_k X^{out}_k, \epsilon)$, $k = 1, 2, \ldots, m$, where $\delta'_k = X^{in}_{k-1}$ if $\mathcal{A}_k$ is of the form $\mathcal{B}_{X_{k-1},X_k}$ and $\delta'_k = \epsilon$ if $\mathcal{A}_k$ is of the form $\mathcal{C}_{X_{k-1},X_k}$. The existence of these sequences implies the existence of sequences $(X_{k-1}, v_k) \vdash^*_{\mathcal{A}} (\delta_k X_k, \epsilon)$, where $\delta_k = X_{k-1}$ if $\mathcal{A}_k$ is of the form $\mathcal{B}_{X_{k-1},X_k}$ and $\delta_k = \epsilon$ if $\mathcal{A}_k$ is of the form $\mathcal{C}_{X_{k-1},X_k}$.

We can place these m sequences after one another to obtain $(X_0, v_1\cdots v_m) \vdash^*_{\mathcal{A}} (\delta_1\cdots\delta_m X_m, \epsilon)$, making use of the fact that $(X_{k-1}, v_k) \vdash^*_{\mathcal{A}} (\delta_k X_k, \epsilon)$ implies $(\delta_1\cdots\delta_{k-1}X_{k-1}, v_k v_{k+1}\cdots v_k) \vdash^*_{\mathcal{A}} (\delta_1\cdots\delta_{k-1}\delta_k X_k, v_{k+1}\cdots v_k)$. Since $b_1\cdots b_m \in L(\mathcal{F})$ and therefore $X_0 = q_s = X_{initial}$ and $X_m \in F_{\mathcal{F}} = F_{\mathcal{A}}$, this sequence recognizes $v_1\cdots v_k$.

This discussion yields the following theorem.

THEOREM 3.2. $C(\mathcal{N}) = C(\mathcal{D})$

This theorem can be paraphrased as "The class of LR(k) languages is contained in the regular closure of the class of LR(0) languages."[2]

EXAMPLE 3.3. *We demonstrate Construction* 1 *by means of an example. Consider the language* $L_{Pal} = \{wcw^R \mid w \in \{a,b\}^*\}$, *where* w^R *denotes the mirror image of string* w. *This language consists of palindromes in which a symbol* c *occurs as the center of each palindrome.*

Now consider the language $L_{PrePal} = \{v \mid \exists w[vw \in L_{Pal}]\}$, *consisting of all prefixes of palindromes. This language, which is obviously not prefix-free, is accepted by the PDA* $\mathcal{A}_{PrePal} = (\Sigma, \Delta, I, F, T)$, *with* $\Sigma = \{a,b,c\}$, $\Delta = \{I, A, B, C, \overline{A}, \overline{\overline{A}}, \overline{B}, \overline{\overline{B}}\}$, $F = \{I, A, B, C\}$, *and* T *consists of the following transitions:*

$$\begin{array}{lcll}
X & \overset{a}{\mapsto} & XA & for\ X \in \{I, A, B\}, \\
X & \overset{b}{\mapsto} & XB & for\ X \in \{I, A, B\}, \\
X & \overset{c}{\mapsto} & XC & for\ X \in \{I, A, B\}, \\
C & \overset{a}{\mapsto} & C\overline{A}, & \\
C\overline{A} & \overset{\epsilon}{\mapsto} & \overline{\overline{A}}, & \\
A\overline{\overline{A}} & \overset{\epsilon}{\mapsto} & C, & \\
C & \overset{b}{\mapsto} & C\overline{B}, & \\
C\overline{B} & \overset{\epsilon}{\mapsto} & \overline{\overline{B}}, & \\
B\overline{\overline{B}} & \overset{\epsilon}{\mapsto} & C. &
\end{array}$$

The automaton operates by pushing each a *or* b *it reads onto the stack in the form of* A *or* B *until it reads* c, *and then the symbols read are matched against the occurrences of* A *and* B *on the stack. Note that* F *is* $\{I, A, B, C\}$, *which means that a recognized string may be the prefix of a palindrome instead of being a palindrome itself.*

[2]Not all definitions of LR(0) in the literature are equivalent. For example, [8] allows some LR(0) languages that are not prefix-free. Theorem 13.3.1 in [8] implies that the alternative class of LR(0) languages is contained in the regular closure of the class of what we call LR(0) languages.

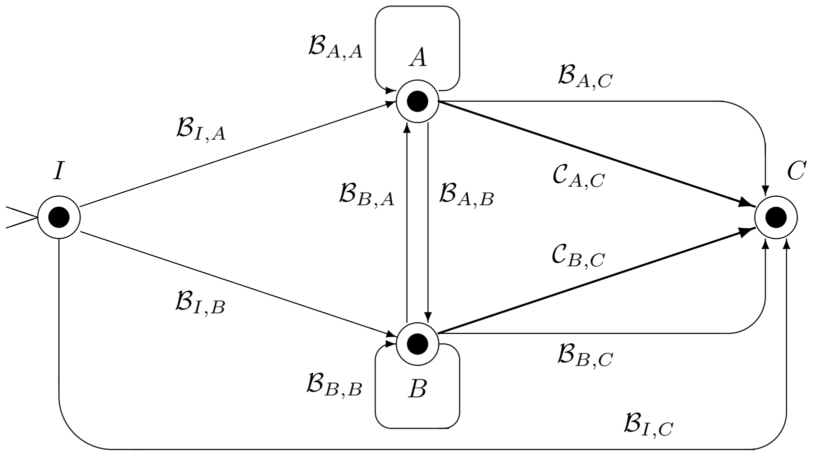


FIG. 3.3. *Metadeterministic automaton* $\rho(\mathcal{A}_{PrePal})$.

The upper level of the metadeterministic automaton $\rho(\mathcal{A}_{PrePal})$ *is shown in Figure* 3.3. *(Automata accepting the empty language have been omitted from this representation, as have vertices which after this omission do not occur on any path from I to any other final state.)*

The automaton $\mathcal{B}_{A,B}$ *accepts the language* $\{b\}$*, since the only pushing transition of* $\mathcal{A}_{PrePal}$ *which places B on top of A reads b. As another example of a lower level automaton, automaton* $\mathcal{C}_{A,C}$ *accepts the language* $\{wa \mid w \in L_{Pal}\}$*, since* $(A, v) \models^{+}_{\mathcal{A}} (AZ, \epsilon) \vdash_{\mathcal{A}} (C, \epsilon)$*, some Z, holds only for v of the form wa, with* $w \in L_{Pal}$*; for example,* $(A, bcba) \vdash_{\mathcal{A}} (AB, cba) \vdash_{\mathcal{A}} (ABC, ba) \vdash_{\mathcal{A}} (ABC\overline{B}, a) \vdash_{\mathcal{A}} (AB\overline{\overline{B}}, a) \vdash_{\mathcal{A}} (AC, a) \vdash_{\mathcal{A}} (AC\overline{A}, \epsilon) \vdash_{\mathcal{A}} (A\overline{\overline{A}}, \epsilon) \vdash_{\mathcal{A}} (C, \epsilon)$.

Note that Construction 1 *together with a mechanical transformation from finite automata to regular expressions (e.g.,* [9, *Theorem* 2.4]) *gives us a method for obtaining a regular expression over LR(0) languages, given an LR(k) language. For example, the equation* $L_{PrePal} = \{a, b\}^*(\epsilon \cup L_{Pal})$ *may be derived from Figure* 3.3. □

4. Recognizing fragments of a string. In this section we investigate the following problem. Given an input string $a_1 \cdots a_n$ and an NPDA $\mathcal{A}$, find all pairs of input positions (j, i) such that substring $a_{j+1} \cdots a_i$ is recognized by $\mathcal{A}$; or in other words, such that $(X_{initial}, a_{j+1} \cdots a_i) \vdash^* (X_{final}, \epsilon)$. It will be proved that this problem can be solved in linear time.

For technical reasons we have to assume that the stack always consists of at least two elements. This is accomplished by assuming that a fresh stack symbol $\perp$ occurs below the bottom of the actual stack and that the actual initial configuration is created by an imaginary extra step $(\perp, v) \vdash (\perp X_{initial}, v)$.

The original problem stated above is now generalized to finding all 4-tuples (X, j, Y, i), with $X, Y \in \Delta$ and $0 \leq j \leq i \leq n$, such that $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. In words, this condition states that if a stack has an element labeled X on top, then the pushdown automaton can, by reading the input between j and i and without ever popping X, obtain a stack with one more element, labeled Y, which is on top of X. Such 4-tuples are henceforth called *items*.

The items are computed by a dynamic programming algorithm based on work from [1, 11, 6, 12].

Algorithm 1 (dynamic programming). Consider an NPDA and an input string $a_1 \cdots a_n$.

```
1. Let the set $\mathcal{U}$ be $\{(\bot, i, X_{initial}, i) \mid 0 \leq i \leq n\}$.
2. Perform one of the following two steps as long as one of them is applicable.
push  1. Choose a pair, not considered before, consisting of a transition $X \overset{z}{\mapsto} XY$
         and an input position $j$, such that $z = \epsilon \vee z = a_{j+1}$.
      2. If $z = \epsilon$ then let $i = j$, else let $i = j + 1$.
      3. Add item $(X, j, Y, i)$ to $\mathcal{U}$.
pop   1. Choose a triple, not considered before, consisting of a transition $XY \overset{\epsilon}{\mapsto} Z$
         and items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}$.
      2. Add item $(W, h, Z, i)$ to $\mathcal{U}$.
3. Finally, define the set $\mathcal{V}$ to be $\{(j, i) \mid (\bot, j, X_{final}, i) \in \mathcal{U}\}$.
```

It can be proved [1, 11] that Algorithm 1 eventually adds an item (X, j, Y, i) to $\mathcal{U}$ if and only if $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. Specifically, $(\bot, j, X_{final}, i) \in \mathcal{U}$ is equivalent to $(\bot, a_{j+1} \cdots a_i) \vdash (\bot X_{initial}, a_{j+1} \cdots a_i) \vdash^* (\bot X_{final}, \epsilon)$. Therefore, the existence of such an item $(\bot, j, X_{final}, i) \in \mathcal{U}$, or, equivalently, the existence of $(j, i) \in \mathcal{V}$, indicates that substring $a_{j+1} \cdots a_i$ is recognized by $\mathcal{A}$, which solves the original problem stated at the beginning of this section.

If no restrictions apply, the number of 4-tuples computed in $\mathcal{U}$ can be quadratic in the length of the input. The central observation is this: It is possible that items $(X, j, Y, i) \in \mathcal{U}$ are added for several (possibly linearly many) i, with fixed X, j, and Y. This may happen if $(\bot, a_h \cdots a_j \cdots a_{i_m}) \vdash^* (\delta X, a_{j+1} \cdots a_{i_m}) \models^+ (\delta XY, a_{i_1+1} \cdots a_{i_m})$ and $(Y, a_{i_1+1} \cdots a_{i_m}) \vdash^+ (Y, a_{i_2+1} \cdots a_{i_m}) \vdash^+ \cdots \vdash^+ (Y, a_{i_{m-1}+1} \cdots a_{i_m}) \vdash^+ (Y, \epsilon)$, which leads to m items $(X, j, Y, i_1), \ldots, (X, j, Y, i_m)$. Such a situation can in the most trivial case be caused by a pair of transitions $X \overset{z}{\mapsto} XY$ and $XY \overset{\epsilon}{\mapsto} X$; the general case is more complex, however.

However, whenever it can be established that for all X, j, and Y there is at most one i with (X, j, Y, i) being constructed, the number of entries computed in $\mathcal{U}$ is linear in the length of the input string, and we get a linear time bound by the reasoning presented at the end of this section.

The following definition identifies the intermediate objective for obtaining a linear complexity. We define a PDA to be *loop-free* if $(X, v) \vdash^+ (X, \epsilon)$ does not hold for any X and v. The intuition is that reading some input must be reflected by a change in the stack.

Our solution to linear-time recognition for automata which are not loop-free is the following: We define a language-preserving transformation from an arbitrary NDPA to a loop-free NDPA. (A similar transformation for the purpose of recognizing suffixes of strings in linear time was described in [13].) Intuitively, this is done by pushing extra elements $\overline{X}$ on the stack so that we have $(X, v) \vdash^+ (\overline{X}X, \epsilon)$ instead of $(X, v) \vdash^+ (X, \epsilon)$, where $\overline{X}$ is a special stack symbol to be defined shortly.

As a first step we remark that for an NPDA we can divide the stack symbols into two sets *PUSH* and *POP*, defined by

$$\begin{aligned} PUSH &= \{X \mid \text{there is a transition } X \overset{z}{\mapsto} XY\}, \\ POP &= \{Y \mid \text{there is a transition } XY \overset{\epsilon}{\mapsto} Z\} \cup \{X_{final}\}. \end{aligned}$$

It is straightforward to show that determinism of the PDA requires that *PUSH* and *POP* be disjoint. We may further assume that each stack symbol belongs to either *PUSH* or *POP*, provided we assume that the PDA is *reduced*, meaning that there are no transitions or stack symbols which are useless for obtaining the final configuration from an initial configuration.[3]

CONSTRUCTION 2 (NPDA transformation). *Consider an NPDA* $\mathcal{A} = (\Sigma, \Delta, X_{initial}, \{X_{final}\}, T)$*, of which the set of stack symbols* Δ *is partitioned into* $PUSH$ *and* POP*, as explained above. From this NPDA a new PDA* $\tau(\mathcal{A}) = (\Sigma, \Delta', X'_{initial}, \{X'_{final}\}, T')$ *is constructed,* $X'_{initial}$ *and* X'_{final} *being fresh symbols, where* $\Delta' = \Delta \cup \{X'_{initial}, X'_{final}\} \cup \{\overline{X} \mid X \in PUSH\}$*,* $\overline{X}$ *being fresh symbols, and the transitions in* T' *are given by*

$$\begin{array}{lll}
XY & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & Z \quad \textit{for } XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} Z \textit{ with } Z \in POP,\\
XY & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & \overline{Z} \quad \textit{for } XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} Z \textit{ with } Z \in PUSH,\\
\overline{X} & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & \overline{X}X \quad \textit{for } X \in PUSH,\\
\overline{X}Y & \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} & Y \quad \textit{for } X \in PUSH, Y \in POP,\\
X & \overset{z}{\mapsto}_{\tau(\mathcal{A})} & XY \quad \textit{for } X \overset{z}{\mapsto}_{\mathcal{A}} XY,
\end{array}$$

and the two transitions $X'_{initial} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X'_{initial}X_{initial}$ *and* $X'_{initial}X_{final} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X'_{final}$*.*

To provide an intuitive explanation of this construction, we observe that the unwanted sequences of transitions have the property of replacing a push symbol X by itself without affecting the part of the stack under it in the course of doing so. The transformation has the effect that instead of X, a padded form of it consisting of two symbols $\overline{X}X$ is produced by the corresponding new transition sequence. Thus, for example, a sequence $(X, v_1v_2) \models^+ (XY_1, v_2) \vdash (X, v_2) \models^+ (XY_2, \epsilon) \vdash (X, \epsilon)$ in the original automaton is turned into a sequence $(X, v_1v_2) \models^+ (XY_1, v_2) \vdash (\overline{X}, v_2) \vdash (\overline{X}X, v_2) \models^+ (\overline{X}XY_2, \epsilon) \vdash (\overline{X}\,\overline{X}, \epsilon) \vdash (\overline{X}\,\overline{X}X, \epsilon)$ in the transformed automaton.

The padding has to be gotten rid of later, viz. when some genuine pop symbol is on top of it. We could, for example, obtain $(\overline{X}\,\overline{X}X, z) \vdash (\overline{X}\,\overline{X}XY, \epsilon) \vdash (\overline{X}\,\overline{X}Z, \epsilon) \vdash (\overline{X}Z, \epsilon) \vdash (Z, \epsilon)$, where the original automaton would do $(X, z) \vdash (XY, \epsilon) \vdash (Z, \epsilon)$, assuming $Z \in POP$.

EXAMPLE 4.1. *We demonstrate this construction further by means of a more elaborate example.*

Consider the NPDA $\mathcal{A} = (\{a, b\}, \{X, Y, Z, P\}, X, \{P\}, T)$*, where* T *contains the transitions given in the left half of Figure* 4.1*. It is clear that* $\mathcal{A}$ *is not loop-free: we have* $(X, a) \vdash (XY, \epsilon) \vdash (X, \epsilon)$*. If the input* $a_1 \cdots a_n$ *to Algorithm* 1 *is* a^n*, then* $(\perp, a_{j+1} \cdots a_i) \models^+ (\perp X, \epsilon)$ *and therefore* $(\perp, j, X, i) \in \mathcal{U}$ *for* $0 \leq j \leq i \leq n$*. This explains why the time complexity is quadratic.*

We divide the stack symbols into $PUSH = \{X\}$ *and* $POP = \{Y, Z, P\}$*. Of the transformed automaton* $\tau(\mathcal{A}) = (\{a, b\}, \{X, Y, Z, P, X', P', \overline{X}\}, X', \{P'\}, T')$*, the transitions are given in the right half of Figure* 4.1*. That the complexity of Algorithm* 1 *is no longer quadratic but linear for the transformed PDA is proved in the remainder of this section.*

The recognition of aab by $\mathcal{A}$ *and* $\tau(\mathcal{A})$ *is compared in Figure* 4.2. □

We now set out to prove that τ has the required properties.

LEMMA 4.2. *If* $\mathcal{A}$ *is an NPDA, then* $\tau(\mathcal{A})$ *is an NPDA.*

Proof. To check the NPDA property, we must establish that $\tau(\mathcal{A})$ is deterministic, prefix-free, and in normal form. We discuss these points in sequence.

Determinism in the case of $X'_{initial}$ is obvious since only one transition applies. This also holds for X_{final}, for which there were no applicable transitions in $\mathcal{A}$ because

[3]Note that each PDA may be turned into a reduced PDA accepting the same language by just omitting the useless transitions.

$\mathcal{A}$	$\tau(\mathcal{A})$
	$X' \overset{\epsilon}{\mapsto} X'X$
$X \overset{a}{\mapsto} XY$	$X \overset{a}{\mapsto} XY$
$XY \overset{\epsilon}{\mapsto} X$	$XY \overset{\epsilon}{\mapsto} \overline{X}$
	$\overline{X} \overset{\epsilon}{\mapsto} \overline{X}X$
$X \overset{b}{\mapsto} XZ$	$X \overset{b}{\mapsto} XZ$
$XZ \overset{\epsilon}{\mapsto} P$	$XZ \overset{\epsilon}{\mapsto} P$
	$\overline{X}P \overset{\epsilon}{\mapsto} P$ (Some other transitions of this form have been omitted, because they are useless.)
	$X'P \overset{\epsilon}{\mapsto} P'$

FIG. 4.1. *The transformation τ applied to an NPDA $\mathcal{A}$.*

$\mathcal{A}$		$\tau(\mathcal{A})$	
Stack	Input	Stack	Input
X	aab	X'	aab
		$X'X$	aab
XY	ab	$X'XY$	ab
X	ab	$X'\overline{X}$	ab
		$X'\overline{X}X$	ab
XY	b	$X'\overline{X}XY$	b
X	b	$X'\overline{X}\,\overline{X}$	b
		$X'\overline{X}\,\overline{X}X$	b
XZ		$X'\overline{X}\,\overline{X}XZ$	
P		$X'\overline{X}\,\overline{X}P$	
		$X'\overline{X}P$	
		$X'P$	
		P'	

FIG. 4.2. *The sequences of configurations recognizing aab, using $\mathcal{A}$ and $\tau(\mathcal{A})$.*

it is prefix-free by assumption. No transitions apply when X'_{final} is on top of the stack.

For each symbol $\overline{X}$ on top of the stack, exactly one pushing transition may be applied. For each pair of symbols $\overline{X}\,Y$ on top of the stack, with $Y \in \mathit{POP}$, exactly one popping transition may be applied.

The other cases of XY on top with pop symbol Y produce either Z or $\overline{Z}$ deterministically, depending on whether Z is a push or pop symbol.

For push symbols $X \in \Delta$ on top of the stack, the unique push move also available in $\mathcal{A}$ is the only possibility.

Prefix-freeness follows from the fact that no transitions with a final stack symbol on top of the stack are possible. On the same line, the property of being in normal form means that the unique final stack symbol can only be at the bottom. This is guaranteed by producing it only from the initial stack symbol which is itself not produced by any transition. □

LEMMA 4.3. *If $\mathcal{A}$ is an NPDA, then $\mathcal{A}$ and $\tau(\mathcal{A})$ accept the same language.*

Proof. We first prove that for all stack symbols $X_1, \ldots, X_m$ from Δ we have $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_m, \epsilon)$ if and only if $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$, where, for all i, $1 \leq i \leq m$, we have $\alpha_i = \overline{Y_{i,1}}\,\overline{Y_{i,2}} \cdots \overline{Y_{i,r_i}}$ for some $r_i \geq 0$.

only if. The proof is given by induction on the number of steps used in $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_m, \epsilon)$.

1. If zero steps are involved then we have $(X_{initial}, \epsilon) \vdash^*_{\mathcal{A}} (X_{initial}, \epsilon)$. By definition we have also $(X_{initial}, \epsilon) \vdash^*_{\tau(\mathcal{A})} (X_{initial}, \epsilon)$.

2. Suppose that the last step is a push. Then we have $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_{m-1}, z) \vdash_{\mathcal{A}} (X_1 \cdots X_m, \epsilon)$, where the last transition used is $X_{m-1} \overset{z}{\mapsto}_{\mathcal{A}} X_{m-1} X_m$. The induction hypothesis informs us that $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_{m-1} X_{m-1}, z)$. Also, because $X_{m-1} \overset{z}{\mapsto}_{\tau(\mathcal{A})} X_{m-1} X_m$, we have $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_{m-1} X_{m-1} X_m, \epsilon)$.

3. Suppose that the last step is a pop. Then we have $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_{m-1} X'_m X_{m+1}, \epsilon) \vdash_{\mathcal{A}} (X_1 \cdots X_{m-1} X_m, \epsilon)$, where the last transition used is $X'_m X_{m+1} \overset{\epsilon}{\mapsto}_{\mathcal{A}} X_m$. The induction hypothesis informs us that $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \alpha_{m+1} X_{m+1}, \epsilon)$, with $\alpha_{m+1} = \overline{Y_1} \cdots \overline{Y_r}$, for some $r > 0$. We first have $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \overline{Y_1} \cdots \overline{Y_r} X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \overline{Y_1} \cdots \overline{Y_{r-1}} X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} \cdots \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon)$, using the transitions $\overline{Y_j} X_{m+1} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X_{m+1}$, $1 \leq j \leq r$, which exist since $X_{m+1} \in POP$. Subsequently, there are two possibilities:

(i) If $X_m \in PUSH$ then $X'_m X_{m+1} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} \overline{X_m}$ and $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X_m}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X_m} X_m, \epsilon)$. In the last configuration, $\alpha_m \overline{X_m}$ is a sequence of "barred" symbols, as desired.

(ii) If $X_m \in POP$ then $X'_m X_{m+1} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X_m$ and $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$.

if. Analogously to the "only if" part, the proof is given by induction on the number of steps used in $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$. The following cases are possible.

1. Suppose that the last transition used was $\overline{X_m} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} \overline{X_m} X_m$. Then we have $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m XY, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m \overline{X_m}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m \overline{X_m} X_m, \epsilon)$, where $\alpha_m = \alpha'_m \overline{X_m}$, and the second-to-last transition used was $XY \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} \overline{X_m}$ for some X and Y. The induction hypothesis informs us that $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_{m-1} XY, \epsilon)$. From $XY \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} \overline{X_m}$ we conclude the existence of $XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} X_m$. Therefore, $(X_1 \cdots X_{m-1} XY, \epsilon) \vdash_{\mathcal{A}} (X_1 \cdots X_{m-1} X_m, \epsilon)$.

2. Suppose that the last transition used was $\overline{X} X_m \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X_m$. Then we have $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X_m} X_m, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$. Since $\alpha_m \overline{X_m}$ is a sequence of barred symbols, the induction hypothesis informs us that $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_m, \epsilon)$.

3. Suppose that the last transition used was $XY \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X_m$ for X and Y from Δ. Then we have $(X_{initial}, v) \vdash^*_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha_m XY, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha_m X_m, \epsilon)$. From the induction hypothesis, $(X_{initial}, v) \vdash^*_{\mathcal{A}} (X_1 \cdots X_{m-1} XY, \epsilon)$. The transition $XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} X_m$ is identically available in $\mathcal{A}$, thus providing the desired sequence of transitions.

4. The argument for a transition $X \overset{z}{\mapsto}_{\tau(\mathcal{A})} XY$, $X \in \Delta$, is analogous to case 3 because again the transition is available also in the old automaton.

From the definition of τ it is clear that $(X'_{\mathit{initial}}, v) \vdash^*_{\tau(\mathcal{A})} (X'_{\mathit{final}}, \epsilon)$ is possible only if $(X_{\mathit{initial}}, v) \vdash^*_{\tau(\mathcal{A})} (X_{\mathit{final}}, \epsilon)$. From the "if" part we conclude that $(X_{\mathit{initial}}, v) \vdash^*_{\tau(\mathcal{A})} (X_{\mathit{final}}, \epsilon)$ implies $(X_{\mathit{initial}}, v) \vdash^*_{\mathcal{A}} (X_{\mathit{final}}, \epsilon)$.

From the "only if" part we conclude that $(X_{\mathit{initial}}, v) \vdash^*_{\mathcal{A}} (X_{\mathit{final}}, \epsilon)$ implies $(X'_{\mathit{initial}}, v) \vdash_{\tau(\mathcal{A})} (X'_{\mathit{initial}} X_{\mathit{initial}}, v) \vdash^*_{\tau(\mathcal{A})} (X'_{\mathit{initial}} \alpha X_{\mathit{final}}, \epsilon) \vdash^*_{\tau(\mathcal{A})} (X'_{\mathit{initial}} X_{\mathit{final}}, \epsilon) \vdash_{\tau(\mathcal{A})} (X'_{\mathit{final}}, \epsilon)$, for some $\alpha = \overline{Y_1} \cdots \overline{Y_r}$, using the transitions $\overline{Y_j} X_{\mathit{final}} \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} X_{\mathit{final}}$, $1 \leq j \leq r$, which exist since $X_{\mathit{final}} \in \mathit{POP}$.

This proves the equivalence of the two accepted languages. □

LEMMA 4.4. *If $\mathcal{A}$ is an NPDA, then $\tau(\mathcal{A})$ is loop-free.*

Proof. Consider the set of stack symbols of $\tau(\mathcal{A})$. We define a partial ordering $<$ on these symbols as the least ordering satisfying

$$\begin{aligned}
X'_{\mathit{final}} &< X'_{\mathit{initial}}, \\
X &< \overline{Y} \quad \text{for all } X \in \mathit{POP}, Y \in \mathit{PUSH}, \\
\overline{Y} &< Z \quad \text{for all } Y, Z \in \mathit{PUSH}, \\
X &< Z \quad \text{for all } X \in \mathit{POP}, Z \in \mathit{PUSH}.
\end{aligned}$$

Note that this relation is transitive and irreflexive. Below we prove that if $(X, v) \vdash^+_{\tau(\mathcal{A})} (Z, \epsilon)$ then $Z < X$. This is sufficient to prove that $\tau(\mathcal{A})$ is loop-free, since $<$ is irreflexive.

Consider $(X, v) \models^+_{\tau(\mathcal{A})} (XY, \epsilon) \vdash_{\tau(\mathcal{A})} (Z, \epsilon)$, using some transition $XY \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} Z$ for the last step. It is obvious that $X \notin \mathit{POP}$ since otherwise Y could not have been on top of X. There are three remaining cases.

(i) If $X = X'_{\mathit{initial}}$ then Z must be X'_{final}. Therefore, $Z < X$.

(ii) If $X \in \mathit{PUSH}$ then either $Z \in \mathit{POP}$ or Z is of the form $\overline{Z'}$, with $Z' \in \mathit{PUSH}$, according to the definition of τ. Therefore, in either case $Z < X$.

(iii) If X is of the form $\overline{X'}$, with $X' \in \mathit{PUSH}$, then the transition $XY \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} Z$ must be of the form $\overline{X'} Y \overset{\epsilon}{\mapsto}_{\tau(\mathcal{A})} Y$, with $Y = Z \in \mathit{POP}$. Therefore, $Z < X$.

Since each sequence $(X, v) \vdash^+_{\tau(\mathcal{A})} (Z, \epsilon)$ can be split up into smaller sequences (in this case at most two, leading from a symbol in PUSH to a barred symbol and then to a symbol in POP) of the form $(X, v) \models^+_{\tau(\mathcal{A})} (XY, \epsilon) \vdash_{\tau(\mathcal{A})} (Z, \epsilon)$ and since $<$ is transitive, the required result follows. □

We now return to the issue of the time complexity of Algorithm 1. We start with a minor result.

LEMMA 4.5. *Let $\mathcal{U}$ be computed, using Algorithm 1, for a loop-free NPDA and certain input. There can be at most one item of the form $(X, j, Y, i) \in \mathcal{U}$ for each X, Y, and j.*

Proof. The existence of an item $(X, j, Y, i) \in \mathcal{U}$ requires that $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. Because the NPDA is (by definition) deterministic, the existence of two items $(X, j, Y, i_1), (X, j, Y, i_2) \in \mathcal{U}$ (say, $i_1 < i_2$) requires that $(X, a_{j+1} \cdots a_{i_1}) \models^+ (XY, \epsilon)$ and $(Y, a_{i_1+1} \cdots a_{i_2}) \vdash^+ (Y, \epsilon)$, because of the definition of $\models^+$. However, $(Y, a_{i_1+1} \cdots a_{i_2}) \vdash^+ (Y, \epsilon)$ is not possible if the NPDA is loop-free. □

THEOREM 4.6. *For a loop-free NPDA, Algorithm 1 has linear time demand, measured in the length of the input.*

Proof. Let the input be $a_1 \cdots a_n$. Let $|\Delta|$ denote the number of stack symbols. We investigate how many steps are applied in the process of computing $\mathcal{U}$.

push. Since the PDA is deterministic, there are $\mathcal{O}(|\Delta| \cdot n)$ combinations of a stack symbol X and an input position i such that there is a transition $X \overset{z}{\mapsto} XY$ with $z = \epsilon \vee z = a_i$. Therefore, the pushing step is applied $\mathcal{O}(|\Delta| \cdot n)$ times.

pop. There are $\mathcal{O}(|\Delta|^2 \cdot n)$ items of the form $(W, h, X, j) \in \mathcal{U}$ because of Lemma 4.5. For each of these, there are $\mathcal{O}(|\Delta|)$ items of the form $(X, j, Y, i) \in \mathcal{U}$, again because of Lemma 4.5. The popping step is therefore applied $\mathcal{O}(|\Delta|^3 \cdot n)$ times.

Together, they yield $\mathcal{O}(|\Delta|^3 \cdot n)$ steps. Computing $\mathcal{V}$ from $\mathcal{U}$ can be done straightforwardly within $|\Delta|^2 \cdot n$ steps, since there are at worst that many elements in $\mathcal{U}$ according to Lemma 4.5. $\square$

COROLLARY 4.7. *For any NPDA $\mathcal{A}$ and input $a_1 \cdots a_n$, the set $\{(j, i) \mid a_{j+1} \cdots a_i \in L(\mathcal{A})\}$ can be computed in linear time.*

Proof. From Lemmas 4.2 and 4.4 and Theorem 4.6 we conclude that $\{(j, i) \mid a_{j+1} \cdots a_i \in L(\tau(\mathcal{A}))\}$ can be computed in linear time. According to Lemma 4.3, this is the same set as $\{(j, i) \mid a_{j+1} \cdots a_i \in L(\mathcal{A})\}$. $\square$

5. Metadeterministic recognition. With the results from the preceding section, we can prove that the recognition problem for metadeterministic languages can be solved in linear time by giving a tabular algorithm simulating metadeterministic automata.

Consider a metadeterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$. Because of Theorem 3.2, we may assume without loss of generality that the DPDAs in A are all NPDAs. Because of the existence of transformation τ, we may furthermore assume that those NPDAs are all loop-free.

For deciding whether some input string $a_1 \cdots a_n$ is recognized by $\mathcal{M}$, we first determine which substrings of the input are recognized by which NPDAs in A. Then, we traverse the finite automaton, identifying the input symbols of $\mathcal{F}$ with automata which recognize consecutive substrings of the input string. In order to obtain linear time complexity, we again use tabulation, this time by means of pairs (q, i), which indicate that state q has been reached at input position i.

The following is the complete algorithm.

ALGORITHM 2 (metadynamic programming). Consider a metadeterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, where $\mathcal{F} = (S, Q, q_s, F, T)$ and A is a finite set of loop-free NPDAs, and consider an input string $a_1 \cdots a_n$.

```
1. Construct the tables $\mathcal{V}_{\mathcal{A}}$ as the sets $\mathcal{V}$ in Algorithm 1 for the respective $\mathcal{A} \in A$ and
input $a_1 \cdots a_n$.
2. Let the set $\mathcal{W}$ be $\{(q_s, 0)\}$. Perform the following as long as it is applicable.
    1. Choose a quadruple, not considered before, consisting of
           (i) a pair $(q, j) \in \mathcal{W}$,
          (ii) a PDA $\mathcal{A} \in A$,
         (iii) a pair $(j, i) \in \mathcal{V}_{\mathcal{A}}$, and
          (iv) a state $p \in Q$,
       such that $(q, b, p) \in T$ for some $b$ with $\mu(b) = \mathcal{A}$.
    2. Add $(p, i)$ to $\mathcal{W}$.
3. Recognize the input when $(q, n) \in \mathcal{W}$, for some $q \in F$.
```

THEOREM 5.1. *Algorithm 2 recognizes $a_1 \cdots a_n$ if and only if $a_1 \cdots a_n \in L(\mathcal{M})$.*

Proof. First we prove that Algorithm 2 eventually adds an item (q, i) to $\mathcal{W}$ if and only if there is some string $b_1 b_2 \cdots b_m \in S^*$, a sequence of states $q_0, \ldots, q_m \in Q$, a sequence of PDAs $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m \in A$, and a sequence of strings $w_1, \ldots, w_m \in \Sigma^*$ such that

(i) $q_0 = q_s$, $(q_{k-1}, b_k, q_k) \in T$ for $1 \le k \le m$, and $q_m = q$,
(ii) $\mathcal{A}_k = \mu(b_k)$ for $1 \le k \le m$,
(iii) $w_k \in L(\mathcal{A}_k)$ for $1 \le k \le m$, and
(iv) $a_1 \cdots a_i = w_1 \cdots w_m$.

The "if" part of the proof is by induction on m: Suppose that the above four conditions hold. Then in particular $(q_{m-1}, b_m, q_m) \in T$, $\mathcal{A}_m = \mu(b_m)$, and $w_m = a_{j+1} \cdots a_i \in L(\mathcal{A}_m)$, some j. This last condition is equivalent to $(j, i) \in \mathcal{V}_{\mathcal{A}_m}$. The above four conditions for m imply the same conditions for $m-1$ with j instead of i, and therefore we may use the induction hypothesis to derive that (q_{m-1}, j) is added to $\mathcal{W}$. We conclude that the conditions are fulfilled under which the algorithm adds (q_m, i) to $\mathcal{W}$.

The "only if" part of the proof is very similar. Proof by induction can be applied here if we assume that each item is given a "time stamp," identifying the point in time when that item is (first) added to $\mathcal{W}$.

From the above characterization, the correctness of $(q, n) \in \mathcal{W}$ as criterion for recognition of the input immediately follows. A similar property is proved in full detail in [4]. $\square$

We now get the main theorem of this article.

THEOREM 5.2. *Recognition can be performed in linear time for all metadeterministic languages.*

Proof. It is sufficient to prove that Algorithm 2 operates in linear time. Because of Theorem 4.6 and since there is a finite number of NPDAs in A, the tables $\mathcal{V}_{\mathcal{A}}$, $\mathcal{A} \in A$, can be constructed in linear time.

Furthermore, the table $\mathcal{W}$ is constructed in a linear number of steps, since each step corresponds with one quadruple $((q, j),\ \mathcal{A},\ (j, i),\ p)$, with $(j, i) \in \mathcal{V}_{\mathcal{A}}$, of which there are at most $|Q|^2 \cdot |A| \cdot n$. Note that prefix-freeness of each $\mathcal{A}$ implies that for any j there is at most one i such that $(j, i) \in \mathcal{V}_{\mathcal{A}}$. $\square$

6. On-line simulation. The nature of Algorithm 2 as simulation of metadeterministic automata is such that it could be called an *off-line* algorithm. A case in point is that it simulates steps of PDAs at certain input positions, which can never be useful for recognition of the input if the preceding input were taken into account. By processing the input strictly from left to right and by computing the table elements in a demand-driven way, we obtain an *on-line* algorithm, which leads to fewer table elements, although the *order* of the time complexity is not reduced.

The realization of this on-line algorithm consists of two steps: first, we adapt the pushing step so that the PDAs by themselves are simulated on-line; second, we merge Algorithm 1 and Algorithm 2 such that they cooperate by passing control back and forth concerning (1) where a PDA should start to try to recognize a subsequent substring according to the finite automaton and (2) at what input position a PDA has succeeded in recognizing a substring. Conceptually, the finite automaton and the PDAs operate in a routine–subroutine relation.

ALGORITHM 3 (on-line metadynamic programming). Consider a metadeterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, with $\mathcal{F} = (S, Q, q_s, F, T)$ and A is a finite set of loop-free NPDAs, and consider an input string $a_1 \cdots a_n$.

1. Let the set $\mathcal{W}$ be $\{(q_s, 0)\}$.
2. Let the sets $\mathcal{U}_{\mathcal{A}}$ be $\emptyset$ for all $\mathcal{A} \in A$.
3. Perform one of the following four steps as long as one of them is applicable.

```
down   1. Choose a pair, not considered before, consisting of
                (i)  a pair $(q, i) \in \mathcal{W}$ and
                (ii) a PDA $\mathcal{A} \in A$,
          such that $(q, b, p) \in T$ for some $b$ with $\mu(b) = \mathcal{A}$ and some $p$.
       2. Add $(\perp, i, X_{initial}, i)$ to $\mathcal{U}_{\mathcal{A}}$.
```

push 1. For some PDA $\mathcal{A} \in A$, choose a pair, not considered before, consisting of a transition $X \overset{z}{\mapsto}_{\mathcal{A}} XY$ and an input position j, such that there is an item $(W, h, X, j) \in \mathcal{U}_{\mathcal{A}}$, for some W and h, and such that $z = \epsilon \vee z = a_{j+1}$.
2. If $z = \epsilon$ then let $i = j$, else let $i = j + 1$.
3. Add item (X, j, Y, i) to $\mathcal{U}_{\mathcal{A}}$.

pop 1. For some PDA $\mathcal{A} \in A$, choose a triple, not considered before, consisting of a transition $XY \overset{\epsilon}{\mapsto}_{\mathcal{A}} Z$ and items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}_{\mathcal{A}}$.
2. Add item (W, h, Z, i) to $\mathcal{U}_{\mathcal{A}}$.

up 1. Choose a quadruple, not considered before, consisting of
(i) a pair $(q, j) \in \mathcal{W}$,
(ii) a PDA $\mathcal{A} \in A$,
(iii) an item $(\perp, j, X_{\mathit{final}}, i) \in \mathcal{U}_{\mathcal{A}}$, and
(iv) a state $p \in Q$,
such that $(q, b, p) \in T$ for some b with $\mu(b) = \mathcal{A}$.
2. Add (p, i) to $\mathcal{W}$.

4. The input is recognized when $(q, n) \in \mathcal{W}$ for some $q \in F$.

The popping and pushing steps, simulating the PDA steps, operate much as in Algorithm 1. An important difference is that the pushing step no longer operates irrespective of preceding input: It simulates only a push on some stack element X if it has been established with regard to previously processed input that such an element may indeed appear on top of the stack.

A second difference is that the PDA steps are simulated by starting at input positions computed by the "down" step, which adds $(\perp, i, X_{\mathit{initial}}, i)$ to $\mathcal{U}_{\mathcal{A}}$ only if recognition of a substring recognized by $\mathcal{A}$ is needed from position i in order to enable a transition to a next state in the finite automaton.

The "up" step constitutes a shift of control back to the finite automaton after some PDA has succeeded in recognizing a substring.[4]

The characterization of the elements in $\mathcal{W}$ we gave after Algorithm 2 is still valid for Algorithm 3. The characterization of the elements in the sets $\mathcal{U}_{\mathcal{A}}$ is more restricted than before, however. Relying on a standard result for on-line tabular simulation of PDAs [11], one can prove that Algorithm 3 eventually adds an item (X, j, Y, i) to $\mathcal{U}_{\mathcal{A}}$ for some $\mathcal{A} \in A$ if and only if there is some $h \leq j$ and some state q such that
1. $(q, h) \in \mathcal{W}$ for some q with $(q, b, p) \in T$, some b with $\mu(b) = \mathcal{A}$ and some p,
2. $(\perp, a_{h+1} \cdots a_j) \vdash^*_{\mathcal{A}} (\delta X, \epsilon)$, for some δ, and
3. $(X, a_{j+1} \cdots a_i) \models^+_{\mathcal{A}} (XY, \epsilon)$.

The first condition states that the finite automaton is in need of a substring recognized by PDA $\mathcal{A}$ starting from position h. The second condition states that some configuration can be reached from an initial configuration by reading the input from position h up to position j, and in this configuration an element labeled X is on top of the stack. The third condition is as before.

A device which recognizes some language by reading input strings from left to right is said to satisfy the *correct-prefix property* if it cannot read past the first incorrect symbol in an incorrect input string. A different way of expressing this is that if it has

[4]If μ is bijective and $(q, b, p) \in T$ is unique for each b, then the condition $(\perp, j, X_{\mathit{final}}, i) \in \mathcal{U}_{\mathcal{A}}$ may be replaced with $(Y, j, X, i) \in \mathcal{U}_{\mathcal{A}}$, with X final in $\mathcal{A}$, and mention of $(q, j) \in \mathcal{W}$ may be omitted. This generalization allows arbitrary PDAs as opposed to NPDAs. In particular, nondeterministic PDAs may be used. For deterministic, loop-free (but not necessarily normal or prefix-free) PDAs the on-line algorithm then still has a linear time complexity. We do not pursue this option because the resulting recognition algorithm cannot be turned into a parsing algorithm; see also section 7.

succeeded in processing a prefix w of some input string wv, then w is a prefix of some input string wv' which can be recognized.

A consequence of the on-line property of Algorithm 3 is that it satisfies the correct-prefix property, provided that both the finite automaton $\mathcal{F}$ and the PDAs in A satisfy the correct-prefix property. A straightforward proof can be obtained from the characterizations of the elements in $\mathcal{W}$ and $\mathcal{U}_{\mathcal{A}}$, $\mathcal{A} \in A$, given above.

7. Producing parse trees. We have shown that metadeterministic recognition can be done efficiently. The next step is to investigate how the recognition algorithms can be extended to become parsing algorithms.

The approach to tabular context-free parsing in [11, 6] is to start with pushdown transducers. A pushdown transducer can be seen as a PDA of which the transitions produce certain *output symbols* when they are applied. The *output string*, which is a list of all output symbols which are produced while successfully recognizing an input, is then seen as a representation of the parse.

If the pushdown transducers are to be realized using a tabular algorithm such as Algorithm 1, then we may apply the following to compute all output strings without impairing the time complexity of the recognition algorithm. The idea is that a context-free grammar, the *output grammar*, is constructed as a side effect of recognition. For each item (X, j, Y, i) added to the table, the grammar contains a nonterminal $A_{(X,j,Y,i)}$. This nonterminal is to generate all lists of output symbols which the pushdown transducer produces while computing $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. The rules of the output grammar are created when items are computed from others. For example, if we compute an item (W, h, Z, i) from two items $(W, h, X, j), (X, j, Y, i) \in \mathcal{U}$, using a popping transition $XY \overset{\epsilon}{\mapsto} Z$ which produces output symbol a, then the output grammar is extended with rule $A_{(W,h,Z,i)} \to A_{(W,h,X,j)}\ A_{(X,j,Y,i)}\ a$.

The start symbol of the output grammar is $A_{(\perp,0,X_{\mathit{final}},n)}$, for recognition of the complete input. For Algorithm 1, however, which recognizes fragments of the input, we have several output grammars, of which the start symbols are of the form $A_{(\perp,j,X_{\mathit{final}},i)}$. The sets of rules of these grammars may overlap.

The languages generated by output grammars consist of all output strings which may be produced by the pushdown transducer while successfully recognizing the corresponding substrings. In the case of DPDAs, these are of course singleton languages.

In a straightforward way this method may be extended to off-line simulation of a metadeterministic automaton $\mathcal{M} = (\mathcal{F}, A, \mu)$, where A is now a set of pushdown transducers, as follows.

1. We create subgrammars for v and the respective automata in A separately, following the ideas above.
2. We merge all grammar rules constructed for the different automata $\mathcal{A} \in A$. We assume that the sets of stack symbols from the respective automata are pairwise disjoint in order to avoid name clashes.
3. For each automaton $\mathcal{A} \in A$ we add rules $A_{(\mathcal{A},j,i)} \to A_{(\perp,j,X_{\mathit{final}},i)}$ if $A_{(\perp,j,X_{\mathit{final}},i)}$ is a nonterminal found while constructing $\mathcal{U}_{\mathcal{A}}$.
4. While constructing table $\mathcal{W}$, we may extend the output grammar with a rule $A_{(p,i)} \to A_{(q,j)}\ A_{(\mathcal{A},j,i)}$ when a pair (p, i) is derived from a pair $(q, j) \in \mathcal{W}$ and a pair $(j, i) \in \mathcal{V}_{\mathcal{A}}$.
5. We extend the output grammar with all rules of the form $S \to (q, n)$, where $q \in F$. S is the start symbol of the grammar.

(For on-line processing similar considerations apply.)

In this way, we may produce a context-free grammar reflecting the structure of the input string without impairing the time complexity of the recognition algorithm.

8. Applications.

8.1. Suffix recognition. For a language L we define the language $\mathit{suffix}(L) = \{v \mid \exists w[wv \in L]\}$. A member of $\mathit{suffix}(L)$ will be called a *suffix*. In this section we will assume that L is a deterministic language.

Only recently [2] has it been shown that suffixes can be recognized in linear time. In [13] it was shown furthermore that *parsing* of suffixes is possible in linear time. Here we give an alternative proof of this result as a corollary to the previous sections. This we do by providing a transformation from a deterministic language L, specified in the form of a deterministic PDA $\mathcal{A}$, to a metadeterministic automaton $\sigma(\mathcal{A})$ recognizing suffixes.

For technical reasons, we assume that the PDA $\mathcal{A}$ satisfies a property called *pop-realistic*, which means that if it can pop a number of elements off a stack, then those elements may indeed occur on top of a stack in a configuration reachable from an initial configuration. Formally, we say that a PDA is *pop-realistic* if $(\delta, v) \vdash^* (X, \epsilon)$, some δ, v, X, implies $(X_{\mathit{initial}}, w) \vdash^* (\delta'\delta, \epsilon)$ for some w and δ'.

The assumption that $\mathcal{A}$ is pop-realistic is not a theoretical restriction, since any PDA can be mechanically transformed into one that is pop-realistic and that accepts the same language [5]; nor is it a practical restriction, since many naturally occurring PDAs realizing top-down or LR recognition, for example, already satisfy this property.

CONSTRUCTION 3 (suffix recognition). *Let $\mathcal{A} = (\Sigma, \Delta, X_{\mathit{initial}}, F_{\mathcal{A}}, T_{\mathcal{A}})$ be a DPDA which is pop-realistic. Construct the metadeterministic automaton $\sigma(\mathcal{A}) = (\mathcal{F}, A, \mu)$, with $\mathcal{F} = (S, Q, q_s, \{q_f\}, T_{\mathcal{F}})$, where*

(i) $S = \{e\} \cup \{b_{X,Y} \mid X, Y \in \Delta\} \cup \{c_X \mid X \in \Delta\}$,

(ii) $Q = \Delta \cup \{q_s, q_f\}$,

(iii) q_s *and* q_f *are fresh symbols,*

(iv) $T_{\mathcal{F}} = \{(q_s, e, X) \mid X \in \Delta\} \cup \{(X, b_{X,Y}, Y) \mid X, Y \in \Delta\} \cup \{(X, c_X, q_f) \mid X \in \Delta\}$.

As the set of PDAs we take $A = \{\mathcal{E}\} \cup \{\mathcal{B}_{X,Y} \mid X, Y \in \Delta\} \cup \{\mathcal{C}_X \mid X \in \Delta\}$. These PDAs are defined as follows.[5]

$\mathcal{E}$ is defined to be $(\Sigma, \{\diamond\}, \diamond, \{\diamond\}, \emptyset)$, where $\diamond$ is a fresh symbol.

Each $\mathcal{B}_{X,Y}$ is defined to be $(\Sigma, \Delta \cup \{X^{in}, X^{out}\}, X^{in}, \{X^{out}\}, T)$, where X^{in} and X^{out} are fresh symbols and the transitions in T are those in $T_{\mathcal{A}}$ plus the extra transitions

$$\begin{array}{lll} X^{in} & \overset{\epsilon}{\mapsto}_{\mathcal{B}_{X,Y}} & X^{in}X, \\ X^{in}X' & \overset{\epsilon}{\mapsto}_{\mathcal{B}_{X,Y}} & X^{out} \qquad \textit{for all } ZX' \overset{\epsilon}{\mapsto}_{\mathcal{A}} Y, \textit{ some } X' \textit{ and } Z. \end{array}$$

Each $\mathcal{C}_X$ is defined to be $(\Sigma, \Delta, X, F_{\mathcal{A}}, T_{\mathcal{A}})$.

The function μ maps the symbol e to automaton $\mathcal{E}$, the symbols $b_{X,Y}$ to automata $\mathcal{B}_{X,Y}$, and the symbols c_X to automata $\mathcal{C}_X$.

The automaton $\mathcal{E}$ accepts the singleton language containing the empty string. Its use at a transition $(q_s, e, X) \in T_{\mathcal{F}}$ is to mimic an arbitrary computation of $\mathcal{A}$, leading to a configuration where X is on top of the stack. During this computation an unknown input string is read, and the rest of the stack is composed of an unknown combination of stack symbols.

[5]Note that the languages recognized by some of these automata may be the empty set.

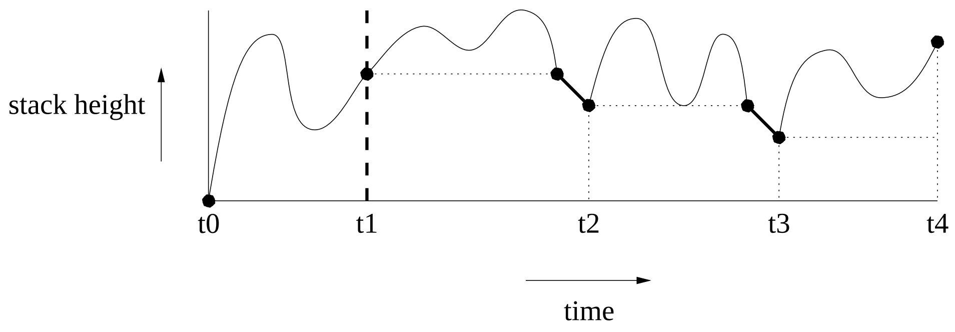


FIG. 8.1. *A stack development of a DPDA $\mathcal{A}$, divided into recognition of a prefix and recognition of the remaining suffix.*

Each automaton $\mathcal{B}_{X,Y}$ mimics all computations beginning with X on top of the stack and ending with the first configuration where the stack shrinks to below the original height; at that point Y is on top. Formally, $\mathcal{B}_{X,Y}$ recognizes a string v if and only if there are X' and Z such that $(X, v) \vdash^* (X', \epsilon)$ and $(ZX', \epsilon) \vdash (Y, \epsilon)$; first, the stack may grow and shrink while reading v, replacing X by some element X', and then X' and an element Z beneath it in the stack are replaced by Y.

Each automaton $\mathcal{C}_X$ mimics all computations of $\mathcal{A}$ that start with X on top of the stack and eventually reach a final configuration without ever having a stack of which the height is 1 less than that of the original stack. Formally, $\mathcal{C}_X$ recognizes a string v if and only if $(X, v) \vdash^*_{\mathcal{A}} (\delta Y, \epsilon)$ for some $Y \in F_{\mathcal{A}}$.

For a complete proof that $L(\sigma(\mathcal{A})) = \mathit{suffix}(L(\mathcal{A}))$, which is similar to the proof in section 3, see [5]. In this article we merely convey the intuition.

Figure 8.1 suggests how the stack may alternately grow and shrink while $\mathcal{A}$ recognizes some input. From t0 to t1 some prefix of the input is read. Acceptance of the remainder of the input, the suffix, is achieved between t1 and t4. Suppose that the stack shrinks to maximally two elements below the height it had at t1: at t2 the stack shrinks to one element below the original height, and at t3 the stack shrinks one element farther. The stack development between t1 and t2 is mimicked by automaton $\mathcal{B}_{X_1,X_2}$, where we assume that X_1 and X_2 are on top of the stack at t1 and t2, respectively. Similarly, the development between t2 and t3 is mimicked by $\mathcal{B}_{X_2,X_3}$. The final part, between t3 and t4, is mimicked by $\mathcal{C}_{X_3}$.

Conversely, if we have consecutive segments of the input recognized by a sequence of automata $\mathcal{B}_{X_1,X_2}$, $\mathcal{B}_{X_2,X_3}$ and $\mathcal{C}_{X_3}$, then composition of the three sequences of transitions leads to a development of the stack as suggested between t1 and t4 in Figure 8.1. The existence of the required sequence of transitions between t0 and t1 follows from the assumption that $\mathcal{A}$ is pop-realistic.

The preceding discussion and Theorem 5.2 together imply the following corollary.

COROLLARY 8.1. *Recognition of suffixes can be performed in linear time for all deterministic languages.*

8.2. Generalized pattern matching. In [10] the following problem is treated. Given are a finite set of input symbols Σ, an input string $a_1 \cdots a_n \in \Sigma^*$ and a pattern $b_1 \cdots b_m \in \Sigma^*$. To be decided is whether $a_1 \cdots a_n = vb_1 \cdots b_m w$ for some $v, w \in \Sigma^*$, or in words, whether $b_1 \cdots b_m$ is a substring of $a_1 \cdots a_n$.

This problem can also be stated as follows. To be decided is whether $a_1 \cdots a_n$ is a member of the language $\Sigma^*\{b_1 \cdots b_m\}\Sigma^*$. This language is described as a regular expression over deterministic languages, i.e., Σ and $\{b_1 \cdots b_m\}$, and therefore this language is metadeterministic. Consequently, the algorithms in this article apply.

The time demand can then be shown to be $\mathcal{O}(n \cdot m)$, which is, of course, $\mathcal{O}(n)$ if n is taken as sole parameter. This is in contrast to the algorithm in [10], which provides a complexity of $\mathcal{O}(n + m)$. This seems a stronger result if time complexity is the only matter under consideration. From a broader perspective, however, our approach allows a larger class of problems to be solved.

For example, the substring problem can be generalized as follows. Given are a finite set of input symbols Σ, an input string $a_1 \cdots a_n \in \Sigma^*$, and a deterministic language $L \subseteq \Sigma^*$. To be decided is whether $a_1 \cdots a_n = uvw$, some $u, w \in \Sigma^*$ and $v \in L$, or in words, whether some substring of $a_1 \cdots a_n$ is in L. As before, the problem can be translated into a membership problem of some string in a metadeterministic language, and therefore our approach allows this problem to be solved in $\mathcal{O}(n)$ time.

9. Conclusions. We have introduced a new subclass of the context-free languages, the metadeterministic languages, which include the deterministic languages properly. We have given recognition algorithms for this class and shown that they have a linear time complexity. Our results are nontrivial since this class contains inherently ambiguous languages. It is still an open problem whether a constructive definition exists for *all* context-free languages which can be recognized in linear time.

Acknowledgments. The second author has had fruitful discussions with Joop Leo about linear-time recognizability of subclasses of context-free languages.

The authors acknowledge help from two referees. The presentation of several proofs was improved by their suggestions.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, *Time and tape complexity of pushdown automaton languages*, Inform. and Control, 13 (1968), pp. 186–206.

[2] J. Bates and A. Lavie, *Recognizing substrings of LR(k) languages in linear time*, ACM Transactions on Programming Languages and Systems, 16 (1994), pp. 1051–1077.

[3] J. Berstel, *Transductions and Context-Free Languages*, B.G. Teubner, Stuttgart, 1979.

[4] E. Bertsch, *An Asymptotically Optimal Algorithm for Non-correcting LL(1) Error Recovery*, Bericht 176, Fakultät für Mathematik, Ruhr-Universität Bochum, Bochum, Germany, 1994.

[5] E. Bertsch and M. Nederhof, *Regular Closure of Deterministic Languages*, Bericht 186, Fakultät für Mathematik, Ruhr-Universität Bochum, Bochum, Germany, 1995.

[6] S. Billot and B. Lang, *The structure of shared forests in ambiguous parsing*, in 27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Vancouver, BC, Canada, 1989, pp. 143–151.

[7] J. Earley, *An efficient context-free parsing algorithm*, Communications of the ACM, 13 (1970), pp. 94–102.

[8] M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.

[9] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[10] D. Knuth, J. Morris, Jr., and V. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[11] B. Lang, *Deterministic techniques for efficient non-deterministic parsers*, in Automata, Languages and Programming, 2nd Colloquium, Saarbrücken, Lecture Notes in Comput. Sci. 14, Springer-Verlag, New York, 1974, pp. 255–269.

[12] M. Nederhof, *Linguistic Parsing and Program Transformations*, Ph.D. thesis, University of Nijmegen, Nijmegen, the Netherlands, 1994.

[13] M. Nederhof and E. Bertsch, *Linear-time suffix parsing for deterministic languages*, J. ACM, 43 (1996), pp. 524–554.

[14] M. Nederhof and G. Satta, *Efficient tabular LR parsing*, in 34th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Santa Cruz, CA, 1996, pp. 239–246.

[15] H. Richter, *Noncorrecting syntax error recovery*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 478–489.

[16] L. Valiant, *General context-free recognition in less than cubic time*, J. Comput. System Sci., 10 (1975), pp. 308–315.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 103–131

CONSTRUCTING EVOLUTIONARY TREES IN THE PRESENCE OF POLYMORPHIC CHARACTERS*

MARIA BONET†, CYNTHIA PHILLIPS‡, TANDY WARNOW§, AND SHIBU YOOSEPH¶

Abstract. Most phylogenetics literature and construction methods based upon characters presume monomorphism (one state per character per species), yet polymorphism (multiple states per character per species) is well documented in both biology and historical linguistics. In this paper we consider the problem of inferring evolutionary trees for polymorphic characters. We show efficient algorithms for the construction of perfect phylogenies from polymorphic data. These methods have been used to help construct the evolutionary tree proposed by Warnow, Ringe, and Taylor for the Indo-European family of languages and presented by invitation at the National Academy of Sciences in November 1995.

Key words. algorithms, graphs, evolutionary trees

AMS subject classifications. 05C05, 68Q25, 92-08, 92B05

PII. S0097539796324636

1. Introduction.

Determining the evolutionary history of a set S of objects (taxa or species) is a problem with applications in a number of domains such as biology, comparative linguistics, and literature. Primary data used to compare different taxa (whether biological species, populations, or languages) can be described using *characters*, where a character is a function $\alpha : S \rightarrow Z$, where Z denotes the integers and thus represents the set of possible *states* of α. In this paper we consider tree construction when characters are permitted to have more than one state on a given object. We call this the *polymorphism problem.* A character that is permitted to have more than one state on a given object will be called a *polymorphic character*, and one that can have only one state for every object is referred to as a *monomorphic* character.

Polymorphism is well documented in both the molecular genetics and comparative linguistics domains [8, 33]. For example, the population geneticist Masatoshi Nei writes: "The study of protein polymorphism has indicated that the extent of genetic variation in natural populations is enormous. However, the total amount of genetic variation cannot be known unless it is studied at the DNA level. The study of DNA polymorphism is still in its infancy, but the results so far obtained indicate that the extent of DNA polymorphism is far greater than that of protein polymorphism" [28, p. 254]. Polymorphism also arises in the comparison of languages. The Indo-Europeanist Donald Ringe writes: "In choosing lexical characters we try to work with

*Received by the editors August 21, 1996; accepted for publication (in revised form) February 20, 1998; published electronically September 14, 1999.
http://www.siam.org/journals/sicomp/29-1/32463.html

†Department of Computer Science, Universidad Politecnica de Cataluña, Barcelona, Spain (bonet@goliat.upc.es). The research of this author was partly supported by NSF grant CCR-9403447.

‡Sandia National Laboratories, Albuquerque, NM (caphill@cs.sandia.gov). The work of this author was performed under U.S. Department of Energy contract DE-AC04-76AL85000.

§Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (tandy@central.cis.upenn.edu). The research of this author was partly supported by an NSF National Young Investigator Award under contract ccr-9457800 and an NSF grant in linguistics.

¶Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (yooseph@gradient.cis.upenn.edu). The research of this author was partly supported by a fellowship from the University of Pennsylvania and by a fellowship from the University of California at Berkeley supported by NSF grant DMS-9406348.

basic meanings (semantic slots), choosing from each language the word that most usually expresses each basic meaning. Languages typically have one word for each basic semantic slot, but instances of two (or even more) words apparently filling the same basic slot are not rare" [30].

Thus polymorphic data is a reality when working with evolutionary tree construction for both linguistic analysis and biological taxa, and methods appropriate for such construction must be devised. In the phylogenetics literature and programs (such as Phylip, PAUP, and MacClade), algorithms and software to *evaluate* fixed leaf-labeled tree topologies for polymorphic data have explicitly required that the number of states be kept quite small because the evaluation requires time exponential in the number of states. This is the first algorithmic study of this problem to go beyond fixed topology problems for bounded number of states.

The concept of an idealized evolutionary tree was introduced by LeQuesne in a series of articles (see, for example, [24, 25]), and later termed "Perfect Phylogenies" by Gusfield in [18], and studied in several papers (see, for example, [13, 14]). The major contribution of this paper is a methodology for inferring perfect phylogenies from monomorphic and polymorphic characters. Recent work in historical linguistics [39] has shown that perfect phylogenies should be obtainable from properly selected and encoded linguistic characters. Algorithms for constructing perfect phylogenies from monomorphic characters were used in [39] to analyze the Indo-European family of languages, whose first-order subgrouping had been argued for decades without resolution. The methodology we propose here significantly extends the range of the data that can be analyzed in historical linguistics. We have applied this methodology to the data set studied by Warnow, Ringe, and Taylor. Detection and resolution of polymorphism led to a modification of their initially proposed phylogeny, which was based only on monomorphic characters. Our methodology and its results were presented at the Symposium on the Frontiers of Science at the National Academy of Sciences in November 1995 [38].

The structure of the rest of the paper is as follows. In section 2 we discuss the causes of polymorphism in linguistics and biology and define the problem of inferring trees from polymorphic characters in these two domains. We show that a perfect phylogeny is an appropriate objective when working with linguistic data as well as some biological data. In section 3 we present two algorithms, one graph theoretic and one combinatorial, for the problem of inferring perfect phylogenies from polymorphic data. In section 4.3 we present a methodology for inferring perfect phylogenies from data which combine monomorphic and polymorphic data. In section 5 we present our analysis of the Indo-European data studied by Warnow, Ringe, and Taylor [39]. In section 6 we consider the problem of inferring evolutionary trees from polymorphic data when a perfect phylogeny is an unlikely outcome. We conclude in section 7.

2. Foundations.

The causes of polymorphism in biology and linguistics differ, and within biology, polymorphism has more than one cause as well. In linguistics, convergence of meanings over time, borrowing of synonyms from other languages, and the inability of modern-day linguists to detect subtle differences of meaning in words from ancient languages can all produce polymorphic characters. Some such cases, like English *little* and *small,* arise by the convergence of meanings over time; others, like American English *stone* and *rock* (to describe a small chunk of the substance that can be thrown), are instances of replacement in progress (*rock* is replacing *stone* in that basic meaning in America). It can be shown that the different manifestations of polymorphism in linguistics each can be described by the conflation of two or more distinct linguistic characters. Often we are able to determine the precise number of monomorphic characters that have merged into the polymorphic character. In linguistics it has been observed that monomorphic characters are *convex*, by which we mean that the nodes sharing any state of any character form a connected set in the tree.

DEFINITION 2.1. *Given a set S of taxa defined by a set C of characters ($|C| = k$), where each $\alpha_j \in C$ is a function $\alpha_j : S \to (2^Z - \{\emptyset\})$, let T be a tree that is leaf-labeled by the taxa in S and with each internal node v labeled with a vector from $(2^Z - \{\emptyset\})^k$ such that the value of $\alpha_j(v)$ is given by the jth component of this vector. A character (polymorphic or monomorphic) $\alpha_j \in C$ is* convex *on T if for all $i \in Z$ the set $X_{i,\alpha_j} = \{v \in V(T) : i \in \alpha_j(v)\}$ is connected. T is a* perfect phylogeny *if every character is convex.*

For polymorphism caused by convergence of convex monomorphic characters, polymorphism can be considered a *separation* problem.

DEFINITION 2.2. *Let α be a character defined on a species set S and let i be a character state of α. Then we define $\alpha^{-1}(i) = \{s \in S : i \in \alpha(s)\}$.*

DEFINITION 2.3. *Let β be a polymorphic character with character states $0, \dots, r-1$, and for each $s \in S$ suppose $|\beta(s)| \le l'$. Then β is* separated *into l monomorphic characters $\alpha_1, \dots, \alpha_l$, where $l \ge l'$, if there is a function $f : \{0, \dots, r-1\} \to \{1, \dots, l\}$, such that if $f(i) = j$ then i is a character state of α_j and in addition, for every $s \in \alpha_j^{-1}(i)$, we have $i \in \beta(s)$.*

Example. Let $S = \{A, B, C, D\}$ and let $\beta(A) = \{0, 1\}, \beta(B) = \{2, 1\}, \beta(C) = \{0\}$, and $\beta(D) = \{2\}$. Then β is separated into two monomorphic characters α_1 and α_2 by setting $f(0) = 1, f(1) = 2, f(2) = 1$. Thus $\alpha_1(A) = \{0\}, \alpha_1(B) = \{2\}, \alpha_1(C) = \{0\}, \alpha_1(D) = \{2\}, \alpha_2(A) = \{1\}$, and $\alpha_2(B) = \{1\}$. In addition, we set $\alpha_2(C) = \{4\}$ and $\alpha_2(D) = \{5\}$.

In the example note that $\alpha_2(C)$ and $\alpha_2(D)$ are set to previously unused values. Thus *if for some $j' \in \{1, 2, \dots, l\}$ $\alpha_{j'}(s)$ is undetermined, then $\alpha_{j'}(s)$ can be set to a previously unused state.*

Problem 1 (separation into l convex characters).

Input. Set S of taxa (defined by set C of characters) and an integer l.

Question. Can we separate each character into at most l monomorphic characters, so that a perfect phylogeny exists for the derived set of monomorphic characters?

Due to inadequate historical evidence, input data may not reflect the actual degree of polymorphism. Separation may be necessary to obtain convexity even if all input characters appear monomorphic. For example, consider four languages with three characters: $A = (1, 2, 1), B = (1, 2, 2), C = (1, 2, 1), D = (1, 2, 2)$. Suppose the first two characters convolve (meanings merge) and linguists detect only one of these characters for each language. This polymorphic character appears monomorphic: $A = (1, 1), B = (1, 2), C = (2, 1), D = (2, 2)$. There is no perfect phylogeny for this set, but we can separate the first character into two such that there is a perfect phylogeny: $A = (1, a, 1), B = (1, b, 2), C = (c, 2, 1)$, and $D = (d, 2, 2)$. Because of lost information, we cannot completely determine the *inferred* characters α_i (hence the use of singletons or previously unused states).

We note that an r-state character can always be separated into r monomorphic *single state* characters which are each convex on all phylogenies. Thus $l = r$ is an easy instance of Problem 1.

In biology, polymorphic characters can arise when dealing with allozyme data [26] and morphological data [40]. In coding allozyme data, each locus is assumed to be a character (as opposed to a character being defined as the presence or absence of individual alleles) and the set of character states can then be defined by the combination of alleles present at the locus. When dealing with sequence data, alternative encodings of the same amino acid sequence can also lead to the presence of polymorphic characters. In each of these cases, the number of different forms that the character can take on a given taxon may be bounded, in which case we may reasonably seek a tree in which every node has no more than some prespecified bound of states for each character. This bound may be character dependent.

Definition 2.4. *Let T be a phylogeny. A character α is said to have* load l *if, for every $v \in V(T)$, $|\alpha(v)| \leq l$. The load of T is defined to be the maximum load on any character.*

Problem 2 (l-load perfect phylogeny).

Input. Set S of taxa (defined by set C of polymorphic characters) and an integer l.

Question. Does an l-load perfect phylogeny exist?

For many morphological characters in biology, convexity is a reasonable assumption (e.g., consider *vertebrate-invertebrate*). Although the causes of polymorphism in biology and linguistics differ, when convexity can be assumed the different problem formulations are equivalent.

Theorem 2.5. *Given a set of taxa defined by a set C of polymorphic characters, T is an l-load perfect phylogeny for C iff we can separate each polymorphic character into at most l monomorphic characters such that T is also a perfect phylogeny for the derived set C'.*

Proof. One direction is easy. For the converse, let T be a perfect phylogeny with load l, let $\alpha \in C$ be given, and assume α has r states present on S. Let T_i be the subgraph of T induced by the vertices labeled i by α. Since T is a perfect phylogeny, each T_i is a subtree. Define G_α to be the graph whose vertices are in one-to-one correspondence with the subtrees $T_i, i = 1, 2, \ldots, r$, and where $(T_i, T_j) \in E$ iff $T_i \cap T_j \neq \emptyset$. Note that since T has load l, G_α has max clique size at most l. G_α is triangulated since it is the intersection graph of subtrees of a tree [7], and hence G_α is perfect [17]. Since G_α is perfect, the chromatic number of G_α equals the max clique size and hence is bounded by l. Hence we can partition the nodes of G_α into at most l independent sets, $V_1, V_2, \ldots, V_l$. Each V_i thus defines a monomorphic character (filled in with singletons), and hence T is a perfect phylogeny for each of these monomorphic characters. ◻

Polymorphism in characters that are based upon columns of molecular sequences behaves differently than polymorphism in morphological characters; for these characters, variations on the parsimony criterion are more appropriate optimization criteria. We discuss the computational complexity of these problems in section 6.

3. Inferring perfect phylogenies from polymorphic characters.

When the maximum permissible load for each character is not given, the problem of inferring perfect phylogenies is best stated as a *minimum load* problem. This is addressed in section 3.1. When the maximum permissible load for each character is given, we have two algorithms which can construct perfect phylogenies; both are efficient when the number of characters is small. These algorithms are presented in section 4. When the character set includes a sufficient number of monomorphic characters, we have a third algorithm which combines techniques for monomorphic and polymorphic characters. This algorithm is presented in section 4.3.

The various parameters to the problem are n (the number of species), k (the number of characters), r (the maximum number of states per character), and l (maximum load for each character).

3.1. Minimum load problems. When convexity of the monomorphic constituents of the polymorphic characters is a reasonable request, we may seek a tree with a prespecified load bound, or else we may seek a tree with a minimum possible load bound. We call the latter problem the *minimum* (or *min*) *load problem.*

We note that the min load problem is NP-hard since the question of whether a 1-load perfect phylogeny exists is NP-complete [4, 36]. However, although the 1-load perfect phylogony problem is NP-complete, the natural fixed parameter versions of the problem are solvable in polynomial time; see [1, 2, 5, 12, 19, 21, 22, 23, 37]. The 2-load perfect phylogeny problem is the next question to consider.

THEOREM 3.1.

(i) *The min load problem can be solved in polynomial time for all fixed n.*
(ii) *The min load problem can be solved in polynomial time when $r = 2$.*
(iii) *The min load problem is NP-hard for all fixed k.*
(iv) *The min load problem is NP-hard for all fixed $r \geq 3$.*
(v) *Determining whether a 2-load perfect phylogeny exists is solvable in polynomial time for all fixed n.*
(vi) *Determining whether a 2-load perfect phylogeny exists is solvable in polynomial time for $r = 2$.*
(vii) *Determining whether a 2-load perfect phylogeny exists is solvable in polynomial time for all fixed k.*
(viii) *Determining whether a 2-load perfect phylogeny exists is NP-complete for all fixed $r \geq 3$.*

Proof. Parts (i) and (v): When n is fixed, the number of possible leaf-labeled topologies is bounded, so we need only consider the min load problem on a fixed topology. Determining the minimum load on a fixed leaf-labeled topology is trivial, since for each internal node $v \in V(T)$ and each character $\alpha \in C$ we simply set $\alpha(v) = \{i : \exists x, y \text{ leaves of } T \text{ with } v \text{ on the path from } x \text{ to } y, \text{ and } i \in \alpha(x) \cap \alpha(y)\}$. This determines the minimum load for the topology. The same argument can be used to show that 2-load perfect phylogeny is solvable in polynomial time when n is fixed.

Parts (ii) and (vi): If $r = 2$, then clearly the min load problem and thus the 2-load perfect phylogeny problem can be solved in polynomial time by observing that 1-load perfect phylogeny on binary characters is solvable in polynomial time [18] and that there is always an r-load perfect phylogeny on any input set containing characters with at most r states.

Part (iii): We now show that the min load problem is NP-hard for all fixed k by showing that the l-load perfect phylogeny problem with fixed number of characters $k \geq 1$, where each character has input load 2 (i.e., two states for every species), is NP-complete. The reduction is from the following problem involving partial t-tree recognition. See section 4.2.3 for definitions of t-trees and partition intersection graphs.

Problem 3 (partial t-tree recognition).

Input. A graph $G = (V, E)$ and an integer $t \leq (n-1)$, where $|V| = n$.

Question. Is G a partial t-tree, i.e., does there exist $G' = (V, E')$ such that $E \subseteq E'$ and G' is a t-tree?

The above problem was shown to be NP-complete by Arnborg, Corneil, and Proskurowski [3].

The reduction is as follows. Let $(G = (V, E), t)$ be an instance of the partial t-tree problem. The corresponding instance of the load problem consists of the species set

$S = \{s_e | e \in E\}$ and one character α, with $\alpha(s_e) = \{i, j\}$, where $e = (i, j)$. Also, set $l = t + 1$. We claim that the instance to the partial t-tree problem has a solution iff the corresponding instance to the load problem has a solution. This can be seen by observing that G is the partition intersection graph of the instance of the load problem and thus we can use Theorems 4.8 and 4.9.

Parts (iv) and (viii): Next we show that the 2-load perfect phylogeny problem, where each of the input characters is monomorphic, is NP-complete for fixed $r \geq 3$. This will also imply that the min load problem is NP-hard for fixed $r \geq 3$. The reduction is from the partial binary characters problem (PBCP), which is defined as follows.

Problem 4 (partial binary character perfect phylogeny).

Input. An $n \times k$ matrix M, of n species and k characters, in which each entry of M is an element of the set $\{0, 1, *\}$.

Question. Can each $*$ entry be set to 0 or 1 so that there exists a 1-load perfect phylogeny with the new matrix ?

The above problem is just a reformulation of the quartet consistency problem, which was shown to be NP-complete by [36].

Given an instance I of PBCP, the instance of the 2-load problem is constructed as follows. Replace each $*$ entry in the matrix defined by I with a 2. Let C be the set of k characters and let S be the set of n species defined by this new matrix. We will add $2k$ new characters and $9k$ new species as follows ($s^i_\alpha = (x, y, z)$ indicates that $\alpha(s^i_\alpha) = x, \alpha^1(s^i_\alpha) = y, \alpha^2(s^i_\alpha) = z$):

1. Initialize $S' = S$ and $C' = C$.
2. For each $\alpha \in C$, do the following:
    a. Define two new characters α^1 and α^2 and nine new species $s^1_\alpha, \ldots, s^9_\alpha$ as follows:
        i. For each $\beta \in C'$ (where $\beta \neq \alpha$) set $\beta(s^i_\alpha) = 2$, where $1 \leq i \leq 9$.
        ii. For each $s \in S'$ set $\alpha^1(s) = 2$ and $\alpha^2(s) = 2$.
        iii. Set $s^1_\alpha = (0, 0, 2)$, $s^2_\alpha = (0, 1, 2)$, $s^3_\alpha = (0, 2, 2)$, $s^4_\alpha = (2, 0, 0)$, $s^5_\alpha = (2, 1, 1)$, $s^6_\alpha = (2, 2, 2)$, $s^7_\alpha = (1, 2, 0)$, $s^8_\alpha = (1, 2, 1)$, $s^9_\alpha = (1, 2, 2)$.
    b. Update $S' = S' \cup \{s^1_\alpha, \ldots, s^9_\alpha\}$ and $C' = C' \cup \{\alpha^1, \alpha^2\}$.

$I' = (S', C')$ is the instance of the load problem. We claim that I has a solution iff I' has a perfect phylogeny with load 2. The proof follows. Let T be a perfect phylogeny which is a solution to instance I of PBCP. Each vertex in T is a k-tuple binary vector. We will first construct the solution for the load problem when restricted to the initial species set S. Identify the species which initially had a $*$ entry for that character and replace the state for that character by a 2. It can be verified that by doing this for every character in C and then relabeling the internal vertices so that the convexity property still holds, we get a solution to the load problem for the initial species S and character set C. Extend the character set C to C' by adding the new characters, which consist entirely of character state 2. This is still a solution to the load problem for S with C'. Let T' be the tree obtained as a result of the above modifications. We will now show how to add the additional $9k$ species. We define $\alpha^{-1}(i) = \{s : \alpha(s) = i\}$. For each $\alpha \in C$ note that there is some edge in T' which separates $\alpha^{-1}(0)$ from $\alpha^{-1}(1)$. For each character $\alpha \in C$, identify an edge e which separates $\alpha^{-1}(0)$ from $\alpha^{-1}(1)$. Attach the 9 new species, associated with α, as shown in Figure 3.1. We note that $\alpha^{-1}(0) \subseteq A$ and $\alpha^{-1}(1) \subseteq B$.

Let T'' be the tree finally obtained after the addition of the $9k$ new species as described above. It can be easily verified that T'' is a solution to instance I' of the

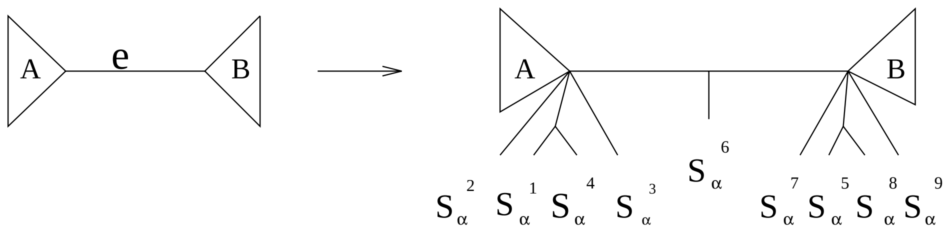


FIG. 3.1. *Adding the* 9 *new species associated with* α.

2-load problem.

For the other direction of the proof, let T be a solution to instance I' of the load problem. We first observe that in any solution to an instance of the 2-load problem involving 3-state monomorphic characters in the input, every character α has associated with it an edge, which splits $\alpha^{-1}(0)$ from $\alpha^{-1}(1)$, or $\alpha^{-1}(0)$ from $\alpha^{-1}(2)$, or $\alpha^{-1}(1)$ from $\alpha^{-1}(2)$. Observe that in I', for each $\alpha \in C$, the only partition possible is $\alpha^{-1}(0)$ from $\alpha^{-1}(1)$. This follows as a result of the constraints imposed by α^1 and α^2. Thus, to get a solution to the PBCP for instance I, we restrict T to the original set of species S and characters C, and then for each character in C we replace the 2's that appear on the 0's side of the partition by 0's and the 2's that appear on the 1's side of the partition by 1's.

This completes the proof.

Part (vii): If k is fixed, then the 2-load perfect phylogeny problem can be solved in polynomial time using the algorithms from section 4. $\square$

This theorem shows that *any* polynomial time algorithm *requires* both k and l bounded (under $P \neq NP$ assumption).

4. Algorithms for perfect phylogenies from polymorphic characters.

In this section we present the two algorithms for inferring perfect phylogenies from polymorphic data when we know the load bound. Although the algorithms we will present assume a universal load bound, these algorithms can be easily modified to allow individual load bounds for each character and will achieve comparable running times. For the sake of clarity, we will present these algorithms as though the load bound is the same for each character; the run times of these algorithms when implemented to handle variable constraints are given within their respective sections.

4.1. A combinatorial algorithm for fixed k and l.

The algorithm we present is an extension and simplification of the algorithm of Agarwala and Fernández-Baca [2]. For the remainder of this section the term *perfect phylogeny* refers to an l-load perfect phylogeny.

Because each character has only r states and each node can choose at most l of these in an l-load perfect phylogeny, the number of possible labels for nodes in the tree is $O(r^{lk})$. Let us call this set S^* and note that $S \subseteq S^*$ (since otherwise some node in S has load greater than l). In contrast to the algorithm in [2], we do not require that the internal nodes be labeled distinctly from the species in S, and instead we will permit species in S to be internal nodes because we can transform any perfect phylogeny in which some species in S label internal nodes into a perfect phylogeny in which all species label leaves by attaching a leaf for s to the internal node labeled by s.

We need some preliminary definitions and facts.

DEFINITION 4.1. $\sum_{\alpha \in C} |\alpha(x) \triangle \alpha(y)|$ *is the* extended Hamming distance *of* $e = (x, y)$*, where* $\triangle$ *denotes the symmetric difference. However, we will call this the* Hamming distance*, understanding this to refer to the extended Hamming distance.*

We note that if a perfect phylogeny exists for S, then one exists where the Hamming distance on any edge is exactly one. We will seek a perfect phylogeny with this property. Working with such perfect phylogenies allows us to quickly solve subproblems because it limits the number of ways a (maximally refined) perfect phylogeny can be constructed.

DEFINITION 4.2 (see [23]). *Given* $x \in S^*$*, the equivalence relation* E_x *is the transitive closure of the following relation* E'_x *on* $S - \{x\}$*:* aE'_xb *if there exists character* α *such that* $(\alpha(a) \cap \alpha(b)) - \alpha(x) \neq \emptyset$*. We denote this set of equivalence classes by* $(S - \{x\})/x$*.*

An observation that follows immediately from this definition is that if T is a perfect phylogeny on S and x an internal node in T, then two species in S which are in the same equivalence class of $(S - \{x\})/x$ must be in the same component of $T - \{x\}$. We also make the following observation.

LEMMA 4.3. *Let* T *be a perfect phylogeny on* S *and* x *be an internal node in* T*. Consider* T *as rooted at* x*. Let* G *be an equivalence class of* $(S - \{x\})/x$ *and let* $y = lca_T(G)$*. Let* v *be a node of* T *on the path from* x *and* y *(thus* $v = x$ *or* $v = y$ *is also possible). Then there exist* $H_1, \ldots, H_t$ *in* $(S - \{v\})/v$ *such that* $H_1 \cup \cdots \cup H_t = G$*.*

Proof. Let T be a perfect phylogeny for S, and x, G, y, and v are as stated. Let $H_1, \ldots, H_t$ be equivalence classes of $(S - \{v\})/v$ containing species from G. Clearly, to prove the lemma it will suffice to prove that all H_i are either disjoint from G or contained in G.

Suppose, by way of contradiction, that for some $i, 1 \leq i \leq t$, H_i contains species from G and from $S - G$. We will show that this implies the existence of a character $\alpha \in C$ and a state a of α such that $a \notin \alpha(v)$, yet $a \in \alpha(x) \cap \alpha(z)$ for some leaf z below v; such a character is not convex on T, contradicting our assumption that T is a perfect phylogeny. This will show that all equivalence classes H_i are either disjoint from or contained in G and will establish our claim.

Since H_i contains species in G and in $S - G$, and is an equivalence class of $(S - \{v\})/v$, there are species $z_1 \in H_i \cap G$ and $z_2 \in H_i - G$ and character α such that $(\alpha(z_1) \cap \alpha(z_2)) - \alpha(v) \neq \emptyset$ (this follows from $(S - \{v\})/v$ being the transitive closure of E_v). Let $a \in \alpha(z_1) \cap \alpha(z_2) - \alpha(v)$. Since z_1 and z_2 are in different equivalence classes of $(S - \{x\})/x$, $a \in \alpha(x)$. Now let $z = lca_T(z_1, z_2)$. This node z is in the subtree rooted at v and satisfies $a \in \alpha(z)$ because T is a perfect phylogeny and z is on the path between z_1 and z_2. This is the character α and state a we stated we would demonstrate, proving our claim. ∎

We now present a dynamic programming algorithm for constructing perfect phylogenies from polymorphic data. We define the *search graph* $SG = (V, E)$ as follows. Each vertex in V is associated with a pair $[G, x]$, where $G = S$ or $G \in (S - \{x\})/x$, and represents the question, Does $G \cup \{x\}$ have a perfect phylogeny? The edges of the search graph are of the form $([G, x][S, x])$, and all pairs of the form $([G_1, x_1], [G_2, x_2])$ where $G_1 \subseteq G_2$ and x_1 and x_2 satisfy $\sum_{\alpha \in C} |\alpha(x_1) \triangle \alpha(x_2)| = 1$. There are $O(r^{lk})$ nodes of type $[S, x]$, and $O(nr^{lk})$ of type $[G, x]$ (because there are at most n equivalence classes in $(S - \{x\})/x$). Also, there are $O(nr^{lk})$ edges of type $([G, x][S, x])$ and $O(nlkr^{lk+1})$ of type $([G_1, x_1], [G_2, x_2])$, since the outdegree of every node is at most lkr.

DEFINITION 4.4. *Given a node* $[G, x]$*, a set of nodes* $[H_1, y], [H_2, y], \ldots, [H_p, y]$

such that (a) Hamming(x,y)= 1 *and (b)* $\cup_i H_i = G$ *is called a* bundle.

There can be multiple bundles going into $[G, x]$, corresponding to the maximally refined perfect phylogenies of $G \cup \{x\}$. If $[H_1, y], [H_2, y], \ldots, [H_p, y]$ is a bundle for $[G, x]$, and all the subproblems have perfect phylogenies, then there is a perfect phylogeny for $G \cup \{x\}$ with subtrees T_i labeled by H_i. We can also have a bundle of just one edge (i.e., ([G,y],[G,x])); such a bundle indicates the existence of a perfect phylogeny T for $G \cup \{y\}$ in which the node corresponding to y has only one child. This is necessary if we require all edges to have Hamming distance 1.

ALGORITHM PHYLOGENY(S). *First create the search graph* G_S*. For each node* $[G, x]$*, determine its bundles. Note that some incoming edges* $([G_1, x_1], [G, x])$ *may not correspond to any bundle because* $(S - \{x_1\})/x_1$ *does not have the proper form (i.e.,* G *may not be the union of a subset of the components of* $(S - \{x_1\})/x_1$*). Remove such edges. Now for each bundle compute the size of the bundle (number of edges)* b_i *and set a counter* $\mathbf{count}_i$ *equal to* b_i*. Each node* $[G_1, x_1]$ *that is a predecessor of node* $[G_2, x_2]$ *is given a pointer to the counter for its bundle. We initialize a queue of "true" nodes as empty.*

We locate each node $[G, x]$ *with* $|G| = 1$*, mark it as true, and place it in the queue. We then pull a node* $[G_1, x_1]$ *out of the queue and process it as follows. For each edge in the search graph* $([G_1, x_1], [G_2, x_2])$ *we decrement the counter for the appropriate bundle into* $[G_2, x_2]$*. If the counter is decremented to* 0*, then all edges of the bundle have been set to true and node* $[G_2, x_2]$ *is added to the queue. When we have processed all edges out of node* $[G_1, x_1]$ *we choose another node from the queue and continue. If we ever try to enqueue a node of the form* $[S, x]$*, then the instance has a perfect phylogeny. If the queue is emptied without ever labeling a node of this form as true, then there is no perfect phylogeny.*

As we enqueue true nodes, we build a topology for a perfect phylogeny for the subproblem represented by that node, ultimately building one for the whole problem if it exists. We denote the topology of the perfect phylogeny for $[G, x]$ *by* $T[G, x]$*. We enqueue* $[G, x]$ *when a bundle* $[H_1, y], [H_2, y], \ldots, [H_p, y]$ *is found such that each* $[H_i, y]$ *has been determined to be true, and hence a topology* $T[H_i, y]$ *for each subproblem has already been determined. We create a new node* v*. If* $x \in S$*, then we label the node* x*. Otherwise it remains unlabeled for now. A method for labeling these nodes is given in the proof of Theorem* 4.6*. We take each of the trees* $T[H_1, y], T[H_2, y], \ldots, T[H_p, y]$*, merge the roots into a single node, and make this node a child of node* v*. Once* $[G, x]$ *has been enqueued, we construct the tree* $T[G, x]$ *and we do not consider any more edges entering* $[G, x]$*. Thus we compute only one topology per true subproblem.*

LEMMA 4.5. *If there exists a perfect phylogeny for* $S \cup \{x\}$*, then the algorithm PHYLOGENY(S) assigns true to* $[G, x]$*, for each* $G \in (S - \{x\})/x$*.*

Proof. The proof is by induction on $|G|$. The base case is trivial. Suppose that the claim holds for all nodes $[G', x']$ where $|G'| < k$. Consider now the node $[G, x]$ where $|G| = k$. Let T be a perfect phylogeny for $S \cup \{x\}$. Assume that T is a perfect phylogeny where the Hamming distance between every pair of adjacent nodes in the tree is one. Consider T as rooted at x, and let $y = lca_T(G)$. From Lemma 4.3, for each node v in the path between x and y, there is a set of equivalence classes of $(S - \{v\})/v$ whose union equals G. Because $y = lca_T(G)$, y is the first node below x for which there are classes $H_1, H_2, \ldots, H_p$, $p > 1$, of $(S - \{y\})/y$ such that $\cup H_i = G$. For all i, $H_i \cup \{y\}$ has a perfect phylogeny. Since $|H_i| < |G|$ it follows that the algorithm has already determined (correctly) that $[H_i, y]$ is true for each $i = 1, 2, \ldots, p$.

We now need to show that for every node z on the path from y to x, $[G, z]$ is set to true. This will prove that $[G, x]$ is true.

Consider the node $z = parent(y)$. We have two cases to consider, depending upon whether z and x are distinct. We consider the first case, where $z = x$. The edges $([H_j, y], [G, x]), j = 1, 2, \ldots, p$ constitute a bundle for $[G, x] = [G, z]$ so that $[G, x] = [G, z]$ is also set to true. We now consider the second case, where $z \neq x$. In this case, G is an equivalence class of $(S - \{z\})/z$, so that $[G, z]$ is also a subproblem, and $([G, y], [G, z])$ is an edge in the search graph. Since $[G, y]$ is set to true (by the above analysis) the algorithm also sets $[G, w]$ to true for all w such that $[G, w]$ is a vertex in the search graph. Thus, for each node w on the path from y to x, $[G, w]$ is set to true; setting $w = x$ yields the result. □

THEOREM 4.6. *The algorithm PHYLOGENY(S) runs in time $O(r^{lk+1}lkn)$ and returns "yes" iff S has a perfect phylogeny.*

Proof. If S has a perfect phylogeny, then there is some species x that can be an internal node of the tree. By Lemma 4.5 the algorithm will return "yes." Suppose now that the algorithm returns the answer yes, and suppose the leaf-labeled tree produced is D. We now show that the internal nodes of this tree can be labeled so as to create a perfect phylogeny T with load l. Given a character α and an unlabeled node v, we assign $\alpha(v)$ to be the states i such that for some pair of leaves x and y in different subtrees of $D - \{v\}$, $i \in \alpha(x) \cap \alpha(y)$. This clearly creates a perfect phylogeny, and we now need to show that the load is bounded by l. Suppose for some node v the load exceeds l, so that (without loss of generality) for each of the first $l+1$ states, $1, 2, \ldots, l+1$, of α, there are at least two subtrees of v with that state. The node v represents a node $[G, y]$ in the search graph, and since it appears in D there is a bundle $[G_1, z], [G_2, z], \ldots, [G_t, z]$ such that all nodes in the bundle are set to true and z and y have distance one. By the construction of D, the subtrees of v have leaf sets $G_1, G_2, \ldots, G_t, S - G$, where $G_i \in (S - \{z\})/z$ for $i = 1, 2, \ldots, t$ (from which it also follows that $S - G$ is the union of the remaining equivalence classes of $(S - \{z\})/z)$. Also by construction, z had load at most l, so that z is labeled with at most l α-states. Thus at least one of the states $1, 2, 3, \ldots, l+1$ is missing from z; without loss of generality let it be $l+1$. It is easy to see that if G_i and G_j (for $i \neq j$) both have leaves with state $l+1$, then they would not be separate equivalence classes in $(S - \{z\})/z$, and similarly if some G_i and $S - G$ both have leaves with state $l+1$. Hence, this labeling has load bounded by l.

The search graph can be constructed in time $O(nlkr^{lk+1})$ by noting that there are $O(r^{kl})$ nodes in the graph, and for each node the maximum number of incoming edges is $O(nrlk)$ and the maximum number of outgoing edges is $O(rlk)$. The rest of the algorithm can be made to run in $(O(nlkr^{lk+1}))$ time, which is linear in the size of the graph. This can be done by sorting the nodes $[G, x]$ according to the size of G and then processing the nodes $[G, x]$ in terms of increasing $|G|$ values. □

Comment. When individual load bounds l_α are given, the algorithm can be modified to run in $O(r^{L+1}Ln)$, where $L = \sum_{\alpha \in C} l_\alpha$.

4.2. A graph-theoretic algorithm for fixed *k* and *l*. In this section we give a graph-theoretic algorithm for the l-load perfect phylogeny problem. The algorithm we present is based upon a characterization of intersection graphs derived from l-load perfect phylogenies as a particular kind of vertex-colored triangulated (i.e., chordal) graphs. On the basis of this characterization we will derive an efficient algorithm for the l-load perfect phylogeny problem when we can fix both l and k.

4.2.1. Preliminary definitions. Let $G = (V, E)$ be a graph. A *vertex coloring* of G is a function color: $V \to Z$. We do not require that color be a *proper* coloring (a coloring function is proper iff $\forall(v,u) \in E$, color$(v) \neq$ color(u)).

The *neighbor set* $\Gamma(v)$ of a vertex v is the set of all vertices in the graph adjacent to v. A vertex v is *simplicial* if $\Gamma(v)$ is a clique.

Given a graph $G = (V, E)$ and a vertex coloring $c : V \to Z$, a *monochromatic clique* in G is a clique with vertex set $V_0 \subset V$ such that color(v)=color(w) for all $v, w \in V_0$. A graph $G = (V, E)$ is *triangulated* if it has no induced cycles of size four or greater. Given a vertex-colored graph $G = (V, E)$, we say that G is *l-triangulated* if G is both triangulated and has no monochromatic cliques of size greater than l. We say that G has an *l-triangulation* $G' = (V, E')$ if $E \subseteq E'$ and G' is l-triangulated.

Let $I = (S, C)$ be an input to the phylogeny problem. For $\alpha \in C$ we define $\alpha_i = \{s \in S : i \in \alpha(s)\}$. The *partition intersection graph* of I is the vertex-colored graph ($G_I = (V, E)$, color) defined by $V = \{\alpha_i : \alpha \in C\}, E = \{(\alpha_i, \beta_j) : \alpha_i \cap \beta_j \neq \emptyset$, where $i \neq j$ if $\alpha = \beta\}$ and for $\alpha \neq \beta$, color$(\alpha_i) =$ color$(\alpha_j) \neq$ color(β_s). Note that because the input I can have load greater than one, the coloring function color may not be proper.

The main results leading to the algorithm can be paraphrased as follows:

- Let I be an input to the l-load perfect phylogeny problem. Then there is an l-load perfect phylogeny for I iff the partition intersection graph G_I has an l-triangulation.
- Given a graph G which is vertex-colored using k colors (not necessarily properly colored) we can determine in time polynomial in fixed k and l whether G has an l-triangulation and construct the l-triangulation when it does.
- Given an l-triangulation G' of G_I we can construct an l-load perfect phylogeny in polynomial time.

As a consequence, we will provide an algorithm for determining if an l-load perfect phylogeny exists for k polymorphic characters defined on n species in $O((rk^3l^2)^{kl+1} + n(kl)^2)$ time.

4.2.2. Characterization of l-triangulated graphs. There is a well-known characterization of triangulated graphs as intersection graphs of subtrees of a tree [7]. In this section we will look at an extension of this particular characterization for l-triangulated graphs.

The following lemma will be useful in the proof of the characterization and also in later theorems. It describes the number of simplicial vertices in a triangulated graph. The proof is simple and is discussed in [17].

LEMMA 4.7. *Let G be a triangulated graph which is not a clique. Then G has at least two nonadjacent simplicial vertices.*

We can make a similar statement about l-triangulated graphs since these graphs are, by definition, also triangulated.

We now present the characterization of l-triangulated graphs.

THEOREM 4.8. *Let $G = (V(G), E(G))$ be a vertex-colored graph. Then G is l-triangulated iff $\exists$ a tree $T = (V(T), E(T))$ together with functions $\varphi : V(G) \to \{$subtrees of $T\}$ and $\phi : V(T) \overset{\text{bijection}}{\to} \{$maximal cliques of $G\}$ such that*

1. *$(v, w) \in E(G)$ iff $\varphi(v) \cap \varphi(w) \neq \emptyset$,*
2. *$\varphi(v) = \{u \in V(T) : v \in \phi(u)\}$,*
3. *$\forall v \in V(T), \phi(v)$ has at most l vertices of the same color.*

Proof. Suppose a tree T exists together with the functions φ and ϕ. We will first show that this, together with conditions 1 and 2, implies that G is triangulated. Let $\Lambda = a_1a_2 \cdots a_ia_1, i \geq 4$, be a simple cycle in G. We will show that Λ has a chord.

Working in *mod i* arithmetic, it can be seen that $\varphi(a_j) \cap \varphi(a_{j+1}) \neq \emptyset\ \forall 1 \leq j \leq i$. Let $\varphi(a_j) = T_j$. Thus $V(T_j) \cap V(T_{j+1}) \neq \emptyset\ \forall 1 \leq j \leq i$. It can be seen that $\exists j$ such that $V(T_{j-1}) \cap V(T_j) \cap V(T_{j+1}) \neq \emptyset$, as otherwise T will contain a cycle. Let $u \in V(T_{j-1}) \cap V(T_j) \cap V(T_{j+1})$. Thus $\phi(u)$ contains a_{j-1}, a_j and a_{j+1} and so $(a_{j-1}, a_{j+1}) \in E(G)$. Hence Λ contains a chord and so G is triangulated.

From condition 3, G can have maximum monochromatic clique size l. Thus G is l-triangulated.

We now prove the converse by induction on $|V(G)|$. Suppose the statement is true for all graphs having less than n vertices. Let G be a connected graph with n vertices and suppose G is l-triangulated. Now if G is complete, then T is a single vertex and the result is trivial. Assume that G is connected but not complete. Since G is l-triangulated, from Lemma 4.7, it contains a simplicial vertex v. Let $A = \{v\} \cup \Gamma(v)$. Note that A is a maximal clique of G and contains monochromatic cliques of size at most l. Let $B = \{u \in A : \Gamma(u) \subset A\}$ and let $X = A - B$. Note that B, X, and $V(G) - A$ are nonempty since G is connected but not complete. Observe that $G' = G|(V(G) - B)$ is l-triangulated and has fewer vertices than G. Applying the induction hypothesis, let T' be the tree and φ' and ϕ' be the functions satisfying the conditions of the theorem for G'. There are two cases to handle here. Case 1 is when X is a maximal clique in G' and Case 2 is when it is not (note that X is a clique in both G and G'):

Case 1. We can obtain T, ϕ, and φ from T', ϕ', and φ' as follows: Identify that vertex $v' \in V(T')$ such that $\phi'(v') = X$. Define $\phi(w) = \phi'(w)\ \forall w \neq v'$ and $\phi(v') = A$. Define $\varphi(y) = \varphi'(y)\ \forall y \notin B$ and $\varphi(y) = \{v'\}\ \forall y \in B$.

Case 2. Identify that vertex $v' \in V(T')$ such that $\phi'(v') \supset X$. Create a new vertex v and connect it to v'. Define $\phi(w) = \phi'(w)\ \forall w \neq v$ and $\phi(v) = A$. Define $\varphi(y) = \{v\} \cup \varphi'(y)\ \forall y \in X$ and $\varphi(y) = \{v\}\ \forall y \in B$.

Note that in both cases A contains at most l vertices from the same color class and that T, ϕ, and φ satisfy the stated conditions. □

THEOREM 4.9. *Given an instance I of the l-load perfect phylogeny problem, let G_I be the corresponding partition intersection graph. Then I has a solution iff G_I has an l-triangulation.*

Proof. Let T be the solution to the instance I of the l-load perfect phylogeny problem. By Theorem 4.8 there is a graph G which is l-triangulated and is related to T as mentioned in that theorem. It can be seen that G is a supergraph of G_I. Thus G_I can be l-triangulated.

Suppose G_I can be l-triangulated. Let G be the l-triangulation of G_I. Then there is a tree T associated with G satisfying the conditions of Theorem 4.8. It can be seen that in T all the character states are convex and each vertex in T has a label set containing at most l vertices of the same color. Thus T is a solution to the instance I of the l-load perfect phylogeny problem. □

4.2.3. *l*-triangulating a vertex-colored graph. In this section we turn to the problem of l-triangulating a vertex-colored graph. The solution to this problem makes use of several properties of triangulated graphs and also of a particular class of triangulated graphs called k-trees.

Further definitions. Triangulated graphs admit orderings $v_1, v_2, \ldots, v_n$ on the vertex set such that for each i, $N_i = \Gamma(v_i) \cap \{v_{i+1}, v_{i+2}, \ldots, v_n\}$ is a clique [17]. These orderings are called *perfect elimination schemes.*

Consider a graph $G = (V, E)$ with $|V| = n \geq k$ that contains at least one k-clique. Such a graph G is a *k-tree* if the nodes of G can be ordered $v_1, v_2, \ldots, v_n$, whereby $\Gamma_G(v_i) \cap \{v_{i+1}, v_{i+2}, \ldots, v_n\}$ is a k-clique for all i with $1 \leq i \leq n - k$. A k-tree also has the following recursive definition: The complete graph on k vertices is a k-tree; if $G = (V, E)$ is a k-tree, and $S \subset V$ is a k-clique, then the graph formed by adding a new vertex v and attaching it to each vertex in S is also a k-tree. Each k-tree may be constructed using several different sequences of these operations. The initial set $S \subset V$ is called a *basis* for the k-tree.

For a graph $G = (V, E)$ and vertex-separator $S \subset V$ with C a component of $G - S$, we define C ∪ cl(S) to be the graph formed by adding to the subgraph of G induced by $C \cup S$ sufficient edges to make S into a clique. Let $G = (V, E)$ be a k-colored graph. We say that G is a *(k,l)-partition intersection graph* if (a) the maximum monochromatic clique size is l, and (b) G is edge covered by kl-cliques. Note that the maximum clique size in a (k, l)-partition intersection graph is kl.

The algorithm we present for l-triangulating a vertex-colored graph is based on dynamic programming. We will need the following lemmas in our algorithm.

LEMMA 4.10. *Let $G = (V, E)$ be a connected graph which is vertex-colored (not necessarily properly colored) using k colors with $|V| \geq kl$, where l is the maximum monochromatic clique size in G. Let the maximum clique size in G be kl. Then G has an l-triangulation iff it has an l-triangulation that is a $(kl - 1)$-tree.*

Proof. Clearly, if G has an l-triangulation that is a $(kl - 1)$-tree, then G has an l-triangulation.

Now suppose that G has an l-triangulation. We will use induction to show that G has an l-triangulation that is a $(kl - 1)$-tree. Base case is when $|V(G)| = kl$, i.e., G is a clique. This is already a $(kl - 1)$-tree and it is l-triangulated.

Suppose the statement is true for all graphs with less than n vertices ($n > kl$) and containing maximum monochromatic clique size l and maximum clique size kl.

Let G be a graph with $|V(G)| = n$. Since G can be l-triangulated, let G' be the l-triangulation of G. From Lemma 4.7 there are at least two nonadjacent simplicial vertices in G'. Pick that simplicial vertex $v \in V(G')$ such that $G' - \{v\}$ still has maximum clique size kl. Let $\Gamma_{G'}(v)$ denote the neighbor set of v in G'. Observe that $G' - \{v\}$ is an l-triangulation of $G - \{v\}$. Thus, by the induction hypothesis, $G' - \{v\}$ can be l-triangulated into a $(kl-1)$-tree. Let G'' be the $(kl-1)$-tree. Let σ be a perfect elimination scheme for G''. Look at x which is the first vertex in $\Gamma_{G'}(v)$ to appear in σ. There are two cases to handle here. Case 1 is when x is within the sequence of last kl vertices appearing in σ. In this case make v adjacent to all vertices in the last kl positions of σ, except with some vertex $u \notin \Gamma_{G'}(v)$ and $\text{color}(u) = \text{color}(v)$. The resulting graph is an l-triangulated $(kl - 1)$-tree. Case 2 is when x is not within the sequence of the last kl vertices appearing in σ. Let A be the set of vertices following x which are neighbors of x. Clearly, $(\Gamma_{G'}(v) - x) \subset A$. Make v adjacent to all vertices in $\Gamma_{G'}(v)$ and also to all except the one vertex u appearing in $A - \Gamma_{G'}(v)$ such that $\text{color}(v) = \text{color}(u)$. The resulting graph is an l-triangulated $(kl - 1)$-tree.

Thus we have that if G has an l-triangulation then it has an l-triangulation which is a $(kl - 1)$-tree. ▯

LEMMA 4.11. *Let G be a (k, l)-partition intersection graph. Then G can be l-triangulated iff there exists a set $K \subseteq V$ of size $(kl - 1)$ which is a separator for G such that for all components C of $G - K$, $C \cup cl(K)$ can be l-triangulated.*

Proof. In Lemma 4.10 it was shown that G has an l-triangulation iff it has an l-triangulation G' which is a $(kl-1)$-tree. If such a G' exists, then G' has a separator of size $kl - 1$ which is a clique by [31]. The converse is straightforward. ▯

We are thus motivated to make the following definition.

DEFINITION 4.12. *Let $G = (V, E)$ be a vertex-colored graph with k colors and with* *maximum monochromatic clique size l. A potential basis for G', the l-triangulation of G, is a subset $V_0 \subseteq V$ such that* (a) $|V_0| = kl - 1$ *and* (b) V_0 *is a vertex separator for G. If $V_0 \subset V$ satisfies both these conditions then we say that V_0 is a* potential basis *for G, and call V_0 a pb-set.*

Our dynamic programming algorithm will solve the l-load problem when the input is a (k, l)-partition intersection graph. Because our input graphs may not be (k, l)-partition intersection graphs, we need the following result.

LEMMA 4.13. *Let $G = (V, E)$ be vertex-colored with a coloring function color (using k colors) and assume that the maximum monochromatic clique size is l. Then there exists a (k, l)-partition intersection graph $G' = (V', E')$ such that the following is true:*

- *For every pb-set $S \subseteq V'$ containing $(k - 1)$ colors and every component C of $G' - S$, $C \cup S$ has all k colors present.*
- *G can be l-triangulated iff G' can be l-triangulated.*
- *The number of vertices in G' is $|V| + |E|(kl - 2)$.*

Proof. For each edge $e = (v, w)$ in E, add $kl - 2$ vertices and sufficient edges so that the kl vertices together form a clique with k color classes of size l. Call the resultant graph G'.

Clearly, $|V(G')| = |V| + |E|(kl - 2)$. Also, since G' is now a (k, l)-partition intersection graph, every edge in G' is part of some kl-clique. Thus, for every *pb*-set S of G' containing $(k - 1)$ colors and for every component C of $G' - S$, $C \cup S$ will have all k colors present.

Finally, suppose G' has an l-triangulation G'_1. Then the subgraph of G'_1 induced by the vertex set $V(G)$ is also l-triangulated [17]. Thus G can be l-triangulated. For the other direction, suppose G has an l-triangulation G_1. Identify the edges in G_1 which were present in G and make each of the edges a part of a new kl-clique. This defines a graph G_1^* which can be verified to be a supergraph of G^* and is also l-triangulated. Thus G can be l-triangulated iff G' can be l-triangulated. □

We now have the basis for an algorithm for computing l-triangulations of vertex-colored graphs:

ALGORITHM B (l-triangulating k-colored graphs).

Step 1. Embed G in a (k, l)-partition intersection graph, G'.

Step 2. Compute all *pb*-sets $V_0 \subseteq V(G')$ and all components C of $G' - V_0$. The subproblems $C \cup cl(V_0)$ are then bucket sorted by size.

Step 3. Use dynamic programming to determine the answers for each subproblem in turn.

Step 4. If there is a *pb*-set V_0 such that for all components C of $G' - V_0$, $C \cup cl(V_0)$ is has an l-triangulation, then return (Yes), else return (No).

It is clear that we need to indicate how we implement Step 3.

Solving subproblems using dynamic programming. We have thus reduced the problem of determining whether the graph G can be l-triangulated to looking at graphs of the form $C \cup cl(S)$, where S is a *pb*-set, C is one of the components of $G' - S$, and we presume G' to be a (k, l)-partition intersection graph.

Rose, Tarjan, and Lueker [32] proved the following lemma about triangulated graphs.

LEMMA 4.14. *Let G be a triangulated graph, σ a perfect elimination scheme for G, and a, b vertices in G. If there is a path P from a to b in G such that every vertex in $P - \{a, b\}$ comes before a and b in the ordering σ, then (a, b) is an edge in G.*

We also observe the following lemma about $(kl - 1)$-trees.

LEMMA 4.15. *If G can be l-triangulated into a $(kl-1)$-tree G', then any $(kl-1)$-clique in G' can be a basis for G'.*

We now prove the following theorem. The proof for this theorem is along the same lines as the proof for Theorem 1 appearing in [27].

THEOREM 4.16. *Let $G = (V, E)$ be a (k, l)-partition intersection graph containing at least $kl+1$ vertices, S_0 pb-set of G, and C a component of $G - S_0$. Then $C \cup cl(S_0)$ can be l-triangulated iff there exists a family $\mathcal{F}$ of l-triangulated $(kl-1)$-trees and a vertex $v \in C$ such that*

1. *For every $F \in \mathcal{F}$ there exists a vertex $x \in S_0$ such that $V(F) = C' \cup cl(S)$, where $S = S_0 \cup \{v\} - \{x\}$ and C' is a component of both $G - S$ and $C \cup cl(S_0) - S$.*
2. *$|V(F)| < |V(C \cup cl(S_0))|$, for every $F \in \mathcal{F}$.*
3. *Every two graphs in $\mathcal{F}$ intersect only on $S_0 \cup \{v\}$.*
4. *$G|(C \cup S_0)$ is contained in $\bigcup_{F \in \mathcal{F}} F$.*

Proof. It is easy to see that if these conditions hold we can combine the l-triangulated $(kl-1)$-trees in $\mathcal{F}$ into one l-triangulated $(kl-1)$-tree covering $C \cup cl(S_0)$ since they intersect only on $S_0 \cup \{v\}$.

For the converse, suppose that $G_1 = C \cup cl(S_0)$ can be l-triangulated. Let G' be an l-triangulation of $C \cup cl(S_0)$. By Lemma 4.15 the $(kl-1)$-clique S_0 can be a basis for G'. Let v be the vertex added to the basis S_0 in the construction of G' and let $S' = S_0 \cup \{v\}$. Thus there is a perfect elimination scheme for G' in which the vertices of S' occur at the end. We will show that we can decompose $C \cup cl(S_0)$ into the union of l-triangulated $(kl-1)$-trees, T_K, each of which is based upon a $(kl-1)$-clique subset $K \subset S'$. We will then show that each such K forming the basis of one of these l-triangulated $(kl-1)$-trees will be a separator for G, so that $T_K - K$ has components $C_1, \ldots, C_r$. We can then in turn write each T_K as the union of possibly smaller $(kl-1)$-trees, $T_K^i = T_K|(C_i \cup K)$. These l-triangulated $(kl-1)$-trees are the ones of interest.

G' is built by adding vertices, one at a time, and making each new vertex adjacent to every vertex in some $(kl-1)$-clique. We will define G_i to be the subgraph of G' induced by the vertex set $\{v_i, v_{i+1}, \ldots, v_{|V|}\}$. Thus $G_{|V|-kl+2}$ is a $(kl-1)$-clique, and to form G_i we make vertex v_i adjacent to every vertex in some $(kl-1)$-clique in G_{i+1}. We will show that we can assign to each added vertex v_i (with $i < (|V| - kl + 1)$) a label $L(v_i)$ the *name* of a $(kl-1)$-clique $K \subset S'$, so that for each $K \subset S'$ the subgraph $T_K = G|V_K$, where $V_K = \{v : L(v) = K \text{ or } v \in K\}$, is an l-triangulated $(kl-1)$-tree. We will also show that every edge e in $G|(C - \{v\})$ is in one of these $(kl-1)$-trees and that the $(kl-1)$-cliques K forming the basis of the $(kl-1)$-trees T_K are separators of G. We will also need to show that the component C' of $C \cup cl(S_0) - L(v)$ containing v is a component of $G - L(v)$. This will prove our assertions.

We first need to show how we assign vertices to $(kl-1)$-clique subsets of S'. Let L be the assignment function we wish to define for every vertex not in S'. Suppose we have constructed the graph G_{i+1} and are now adding v_i to the graph and making it adjacent to every vertex in some $(kl-1)$-clique, R. If $R \subset S'$, then we set $L(v_i) = R$. Otherwise, the vertices in R will consist of (perhaps) some unlabeled vertices (these will be in S') and at least one labeled vertex. If all of the labels in R agree, then this is the label that we will assign to v_i. On the other hand, suppose for our construction that when we make v_i adjacent to every vertex in the $(kl-1)$-clique R not all the labels are the same and that this is the first vertex in this construction for which this happens. In this case, for some vertices v_j and v_k in R, $L(v_j) = X$

and $L(v_k) = Y$, for distinct subsets $X, Y \subset S'$. Without loss of generality we can assume that $i < j < k$. In constructing G_j we made v_j adjacent to every vertex in some $(kl-1)$-clique $C \subset G_{j+1}$. Note that $v_k \in C$ since v_j and v_k are adjacent and $k > j$. Since we were able to set $L(v_j) = X$ unambiguously, this means that either every vertex in C was unlabeled, and thus $X = C$, or that the labeled vertices were all labeled X. Since we have assumed v_k was labeled, we can infer that $L(v_k) = X$ and hence $X = Y$. Thus this assignment of vertices to $(kl-1)$-clique bases is well defined, and each label denotes a subset K of S'. It is easy to see that the subgraph $T_K = G'|V_K$ (for $V_K = \{v : L(v) = K \text{ or } v \in K\}$) is an l-triangulated $(kl-1)$-tree and that T_K is based upon the set K.

By our construction of the labeling function, it is also clear that no edge in G has different labels at its endpoints, so that every edge in $G|(C - \{v\})$ is in exactly one l-triangulated $(kl-1)$-tree, T_K.

We now show that each $(kl-1)$-clique $K \subset S'$ forming the basis of an l-triangulated $(kl-1)$-tree in $\mathcal{F}$ is a separator for $C \cup cl(S_0)$ and for G. We first show that K is a separator for $C \cup cl(S_0)$. Suppose to the contrary, so that for some set $K \subset S$ forming the basis of an l-triangulated $(kl-1)$-tree T_K, $C \cup cl(S_0) - K$ is connected. Let $K = S - \{x\}$. We will show that there is no path from x to any vertex in $C \cup cl(S_0) - K$. Let σ' be a perfect elimination scheme for $T_K \cup \{x\}$. Clearly, we can assume that x is the last vertex in σ' to occur before the vertices of K. Let a be the vertex immediately preceding x. If there is a path from x to a in $C \cup cl(S) - K$, then the edge (a, x) is in G by Lemma 4.14. But then $S \cup \{a\}$ is a $(kl+1)$-clique, contradicting that G has a supergraph which is a $(kl-1)$-tree. The proof can be modified to show that K is a separator for G as well. Hence the $(kl-1)$-trees T^i_K each contain fewer vertices than G.

We now complete our proof by showing that the components of $C \cup cl(S_0) - K$ are also components of $G - K$, where K is the basis of a $(kl-1)$-tree $F \in \mathcal{F}$. Recall that by our construction each such basis K is a set $L(a)$ for some $a \in V(F) - K$. So let C' be a component of $C \cup cl(S_0) - L(a)$, for some $a \in C$. It is easy to see that $L(a) = S_0 \cup \{v\} - \{x\}$ is a separator for $C \cup cl(S_0)$ and that every component X of $C \cup cl(S_0) - L(a)$, such that $x \notin X$, is also a component of $G - L(a)$. Thus we will show that $x \notin C'$, so that C' is a component of $G - L(a)$.

Suppose $x \in C'$. Then x is adjacent to at least one vertex z of $C' - \{x\}$. When we labeled the vertex z we labeled it with $L(a)$ implying that $x \in L(a)$, and yet, by our construction, $x \notin L(a)$. Hence the component C' of $C \cup cl(S_0) - L(a)$ containing a is a component of $G - L(a)$. This completes our proof. □

We can now state the following theorem.

THEOREM 4.17. *Let $G = (V, E)$ be a (k, l)-partition intersection graph with $|V| \geq kl + 1$. Let S_0 be a pb-set and let C be a component of $G - S_0$. Then $C \cup cl(S_0)$ can be l-triangulated iff there exists some vertex v in C and a family of pb-sets $\mathcal{M}$ such that the following is true:*

1. *For each $M \in \mathcal{M}$, $M \subset S_0 \cup \{v\}$ and M is a separator for $C \cup cl(S_0)$ and for G.*
2. *For each vertex $x \in S_0$ there is an $M_x \in \mathcal{M}$ and a component C_x of $G - M_x$ and of $C \cup cl(S_0) - M_x$ such that $|C_x| < |C|$ and $C_x \cup cl(M_x)$ can be l-triangulated.*
3. *Every edge in C is in exactly one C_x given above.*

Proof. Suppose that $C \cup cl(S_0)$ can be l-triangulated and let G' be a l-triangulation of $C \cup cl(S_0)$. From Theorem 4.16 we infer that there is a vertex $v \in C$ such that the subgraph of G' induced by the vertices of $C \cup cl(S_0)$ can be written as the union of the l-triangulated $(kl-1)$-trees T_K based upon *pb*-sets $K \subset S' = S_0 \cup \{v\}$. We will let $\mathcal{M}$ consist of these subsets K, which form the bases of the $(kl-1)$-trees T_K. From Theorem 4.16 it can be seen that $\mathcal{M}$ satisfies the conditions above.

For the converse, if such a family $\mathcal{M} = \{M_i : i \in I\}$ of *pb*-sets exists, then there exists $v \in C$ such that the graph $C \cup cl(S_0)$ is contained in the union of l-triangulatable graphs of the form $C_x \cup cl(M)$, where each $M \in \mathcal{M}$ is a *pb*-set and a subset of $S_0 \cup \{v\}$ and C_x is a component of $G - M$ and a proper subset of C. Since G is a (k,l)-partition intersection graph, these graphs each have all k colors and have monochromatic cliques of maximum size l and also have cliques of maximum size kl. Hence they can be completed to l-triangulated $(kl-1)$-trees T_x, where $V(T_x) = V(C_x \cup M)$. This family of $(kl-1)$-trees $\mathcal{F} = \{T_x : x \in C - \{v\}\}$ shows that $C \cup cl(S_0)$ can be l-triangulated. ∎

THEOREM 4.18. *Let $G = (V,E)$ be a (k,l)-partition intersection graph, $S \subset V$ be a pb-set, and C be a component of $G - S$. Then we can determine whether $C \cup cl(S)$ can be l-triangulated simply by knowing the "answer" for each smaller graph of the form $C' \cup cl(S')$, where S' is a pb-set and C' is a component of $G - S'$.*

Implementation details of the dynamic programming algorithm (Step 3 of Algorithm B). Data structure: A family $\mathcal{X} = \{M_i\}$ of *pb*-sets. For each set M_i in $\mathcal{X}$, and for each of the r_i components C_j of $G - M_i$, we denote by M_i^j, $j = 1, 2, \ldots, r_i$, the subgraph of G induced by $C_j \cup M_i$ with the addition of edges required to make M_i into a clique. Each such M_i^j either can be l-triangulated or cannot be. This will be determined during the algorithm, in order of increasing size of the M_i^j's, and an appropriate answer (yes or no) will be stored for each.

Recall that Step 2 of Algorithm B sorts the subproblems $C_j \cup M_i$ using bucket sort.

ALGORITHM. (the statements in italics denote comments).

```
(* Examine the $M_i^j$ in turn by order of number of vertices,
and determine whether each can be $l$-triangulated.
Any graph containing all $k$ colors with
$l$ vertices per color class can be $l$-triangulated *)
    IF $M_i^j$ has $kl$ vertices with $l$ vertices per color
    class, THEN set its answer to Yes.
    IF $M_i^j$ has $kl$ vertices such that there is one color
    class with more than $l$ vertices, THEN set its answer to No.
(* We will now apply Theorem 4.17 to each graph $M_i^j$
and search for a vertex $v \in M_i^j - M_i$
and family $\mathcal{M}$ satisfying the conditions of Theorem 4.17
to determine whether $M_i^j$ can be $l$-triangulated *)
FOR EACH graph $M_i^j$ in order of size $h > kl$ DO
    FOR EACH  $v \in M_i^j - M_i$ such that
    $M_i \cup \{v\}$ has no color class containing more than
    $l$ vertices, DO
    (* We now check whether for vertex $v$ there is a
    family $\mathcal{M}$ satisfying the conditions of Theorem 4.17 *)
        Examine all sets $M_m$ of vertices in $M_i \cup \{v\}$ which are pb-sets for $G$
        FOR EACH such $M_m$, let $L_m$ be the
        union of the $M_m^j$ which can be
```

```
                $l$-triangulated
                **IF** the union of the $L_m$ (for each
                $M_m$ above) contains $M_i^j - M_i - \{v\}$
                    **THEN** set the answer of $M_i^j$ to
                    Yes and **EXIT-DO**
        **END-DO**
        **IF** no answer was set for $M_i^j$
            **THEN** set the answer for $M_i^j$ to No
        (*Applying Lemma 4.11 now*)
        **IF** $G$ has a vertex-separator $M_i$ such that
            all $M_i^j$ graphs have the answer Yes,
            **THEN** ($G$ can be $l$-triangulated)
                **RETURN** (Yes)
            **ELSE RETURN** (No)
    **END-DO**
end of algorithm.
```

The above algorithm can be modified easily to give back an l-triangulation, if it exists.

Run time analysis of Algorithm B. Let $G = (V, E)$ be the (k, l)-partition intersection graph which is given as input to Step 2 of Algorithm B. Then, in Step 2, in the worst case the algorithm checks all subsets of size $kl - 1$ of which there are $O(|V|^{kl-1})$. Each of these is checked for being a pb-set, which involves checking the set to see if it is a vertex separator. This takes $O(|V|^2)$ for each subset. Bucket sorting the subproblems takes a total of $O(|V|^{kl})$. Step 3, which involves checking to see if a subgraph satisfies the conditions of Theorem 4.17, takes time linear in the number of vertices in the subgraph. Thus the overall complexity is $O(|V|^{kl+1})$.

We summarize with the following.

THEOREM 4.19. *Let $G = (V, E)$ be a (k, l)-partition intersection graph. We can in $O(|V|^{kl+1})$ time determine whether G can be l-triangulated and produce the l-triangulation when it exists.*

4.2.4. Summary of the algorithm to solve the l-load perfect phylogeny problem. Given I, compute the partition intersection graph, G_I, and embed G_I in a (k, l)-partition intersection graph G'_I. Use Algorithm B to determine if G'_I can be l-triangulated, and compute the triangulation $G = (V, E)$ if it exists. If there is no l-triangulation, return No. Else, use G to compute the l-load perfect phylogeny T.

We now briefly discuss how T can be constructed from the l-triangulated graph $G = (V, E)$. Recall that T is related to G by Theorem 4.8.

Let $\sigma = v_1 v_2 \dots v_{|V|}$ be a perfect elimination scheme for G. We will construct the tree inductively where T_i is the tree corresponding to $G|\{v_i, v_{i+1}, \dots, v_{|V|}\}$. Thus $T_1 = T$ is the tree we seek.

Let $A_p = \Gamma(v_p) \cap \{v_{p+1}, v_{p+2}, \dots, v_{|V|}\}$. Inductively, assume we are at vertex v_j in σ and assume we have the tree T_{j+1}. Let v_i be the first vertex following v_j (i.e., $j < i$) in σ which is in $\Gamma(v_j)$. Note that $(A_i \cup \{v_i\}) \supseteq A_j$. Since v_j and v_i are simplicial in $G|\{v_j, v_{j+1}, v_{j+2}, \dots, v_{|V|}\}$ and $G|\{v_i, v_{i+1}, v_{i+2}, \dots, v_{|V|}\}$, respectively, it follows that $|\{v_i\} \cup A_i| > |A_j|$ iff, in $G|\{v_j, v_{j+1}, v_{j+2}, \dots, v_{|V|}\}$, the subgraph induced by $\{v_i\} \cup A_i$ is a maximal clique.

Case 1. If the subgraph induced by $\{v_i\} \cup A_i$ is a maximal clique then it follows that $\{v_j\} \cup A_j$ also induces a maximal clique in $G|\{v_j, v_{j+1}, v_{j+2}, \dots, v_{|V|}\}$. Thus from Theorem 4.8 in T_j there will be a vertex which corresponds to $\{v_j\} \cup A_j$. To get T_j from T_{j+1}, we add a new vertex u to $V(T_{j+1})$ and add an edge from u to the vertex $u' \in V(T_{j+1})$ which corresponds to the maximal clique $\{v_i\} \cup A_i$.

Case 2. If $|\{v_i\} \cup A_i| = |A_j|$, then the subgraph induced by $\{v_i\} \cup A_i$ in $G|\{v_j, v_{j+1}, v_{j+2}, \ldots, v_{|V|}\}$ is not a maximal clique. Let v_q $(j \leq q)$ be the first vertex to the left of v_i in σ such that $v_i \in \Gamma(v_q)$ and $|\{v_i\} \cup A_i| = |A_q|$. If $q = j$, then to get T_j we relabel the vertex $u \in V(T_{j+1})$ corresponding to the maximal clique $\{v_i\} \cup A_i$ to now correspond to $\{v_j\} \cup A_j$. If $q \neq j$, then to get T_j from T_{j+1} we create a new vertex corresponding to $\{v_j\} \cup A_j$ and connect it to the vertex $u' \in V(T_{j+1})$ which corresponds to $\{v_q\} \cup A_q$.

It can be seen that the above operations of obtaining T_j from T_{j+1} can be implemented in $O(deg(v_j))$, where $deg(v)$ is the degree of vertex v. This can be achieved by associating two variables with every vertex $v_r \in \sigma$ $(j < r)$, one that corresponds to the vertex $u \in V(T_r)$ such that u represents the maximal clique $\{v_r\} \cup A_r$ in $G|\{v_r, v_{r+1}, \ldots, v_{|V|}\}$ and one that corresponds to a vertex v_s in σ such that v_s is the first vertex to the left of v_r for which $v_s \in \Gamma(v_r)$ and $|\{v_r\} \cup A_r| = |A_s|$.

The time taken for producing a perfect elimination scheme for G is $O(|V| + |E|)$ [17]. From the discussion above, it can be seen that T can then be constructed in $O(|V| + |E|)$. Hence we have the following theorem.

THEOREM 4.20. *Let $G = (V, E)$ be a vertex-colored graph which is l-triangulated. Then the tree T which satisfies the conditions in Theorem* 4.8 *can be constructed in $O(|V| + |E|)$.*

THEOREM 4.21. *The l-load perfect phylogeny problem for n species and k polymorphic characters can be solved and the l-load perfect phylogeny constructed (when it exists) in $O(nk^2l^2 + (rk^3l^2)^{kl+1})$ time.*

Proof. Let I be the input to the l-load perfect phylogeny problem and $G_I = (V, E)$ be the partition intersection graph. Then $|V| = rk$, and it can be shown that if $|E| > kl|V|$ then there is no l-triangulation [27]. Hence $|E| \leq k^2lr$. Let $G'_I = (V', E')$ be the (k, l)-partition intersection graph embedding of G_I, and note that $|V'| = |V| + |E|(kl - 2) \leq rk + k^3l^2r$. The rest follows. □

Comment. In the case where individual load bounds l_α are given, the algorithm can be modified to run in $O(nL^2 + (rkL^2)^{L+1})$, where $L = \sum_{\alpha \in C} l_\alpha$.

4.3. Inferring perfect phylogenies from mixed data. In the previous section we presented two algorithms for inferring perfect phylogenies from polymorphic character data; these algorithms had running times which were exponential in L, where $L = \sum_{\alpha \in C} l_\alpha$ and l_α is the load bound for the character α. We can use these algorithms directly for sets of characters when some of the characters are monomorphic and some are polymorphic, but the expense would be too large. This follows since in typical data sets the number of characters k is the largest parameter, often in the hundreds or thousands; since $L > k$, algorithms that are exponential in L are prohibitively costly. Instead, we propose a method that should be efficient when the number of monomorphic characters is sufficient to reduce the number of minimal perfect phylogenies to a small number. In practice, as the majority of the characters will be monomorphic, this is likely to be very efficient. The method we propose involves two steps and is efficient when the number of minimal perfect phylogenies generated from the monomorphic characters is small.

ALGORITHM C.

Step 1. Infer all minimal perfect phylogenies from the monomorphic characters, using [23].

Step 2. Determine whether any of the minimal perfect phylogenies obtained in Step 1 can be refined so that each polymorphic character is convex on it within the specified load bound.

Discussion of Step 1. The algorithm by Kannan and Warnow [23] has running time of $O(2^{2r+r^2}k_m^{r+3}+Mk_mn)$, where M is the number of minimal perfect phylogenies and k_m is the number of monomorphic characters. This is theoretically expensive if r, the number of states, is too large; however, in practice the algorithm works quickly as long as not too many of the characters have a large number of states. Also, in practice, as long as the monomorphic characters are independent of each other and comprise a suitably large set, there will be very few perfect phylogenies. Thus we expect Step 1 to be very fast and to produce very few minimal perfect phylogenies.

Discussion of Step 2. We consider the following problem.

Problem. Refining a tree.

Input. Leaf-labeled tree T and set C of polymorphic characters, each with an individual load bound.

Question. Does a perfect phylogeny T' exist for the polymorphic characters, subject to the constraint that T' is a refinement of T?

ALGORITHM D.

```
For each internal $v \in T$ which has degree greater than 3, do
  1. Let $\Gamma(v) = \Gamma_1(v) \cup \Gamma_2(v)$, where $\Gamma_1(v)$ consists of all the neighbors of $v$
     which are leaves and $\Gamma_2(v)$ consists of all the nonleaf neighbors of $v$. For each
     $u_j \in \Gamma_2(v)$ add a new node $w_j$ on the edge $(v, u_j)$. Compute the labeling of
     $w_j$ so as to make every character convex (each character must contain every
     state that appears on both sides of $w_j$).
  2. If some new node has a load for a character that exceeds the stated load
     bound for that character, RETURN(NO). Let $S_v = \Gamma_1(v) \cup \{w_j | w_j$ is a new
     node and $w_j$ is a neighbor of $v\}$. Use any of the algorithms from section 3
     to determine if there is a perfect phylogeny for $(S_v, C)$ satisfying the load
     bounds. If any $(S_v, C)$ fails to have a perfect phylogeny meeting the load
     bounds then RETURN(NO), else RETURN(YES).
```

Example. Consider $S = \{a, b, c, d, e, f, g\}$ and $C = \{\alpha, \beta\}$. Let α be a monomorphic character and β be a polymorphic character with load bounded by three. Also, let $\alpha(a) = \{0\}, \alpha(b) = \{0\}, \alpha(c) = \{1\}, \alpha(d) = \{1\}, \alpha(e) = \{1\}, \alpha(f) = \{1\}$, and $\alpha(g) = \{1\}$. In addition, let $\beta(a) = \{0\}, \beta(b) = \{3\}, \beta(c) = \{2, 4\}, \beta(d) = \{2, 3\}, \beta(e) = \{1, 4\}, \beta(f) = \{1, 3\}$, and $\beta(g) = \{0, 4\}$. Figure 4.1(i) shows the phylogeny T obtained by applying the perfect phylogeny algorithm to the monomorphic character α. Let $S' = \{h, c, d, e, f, g\}$ be the species set obtained at the end of step 1 of Algorithm D. Then $\beta(h) = \{0, 3\}$. The instance $(S', \{\beta\})$ has a perfect phylogeny with load bounded by three. This is shown in Figure 4.1. From this phylogeny it is possible to obtain the solution to the instance (S, C), and this is shown in Figure 4.1(iii).

THEOREM 4.22. *Algorithm D correctly determines whether a perfect phylogeny T' exists refining T within the stated load bounds, and it can be modified to produce the perfect phylogeny T' in time* $min\{O(r^{L+1}Ln^2), O(n^2L^2 + n(rkL^2)^{L+1})\}$.

Proof. If the algorithm returns NO, it is clear that no perfect phylogeny within the constraints of the problem exists. If it returns YES, then the perfect phylogenies refining each of the stars can be hooked up via the new nodes. The refinement can be done by using the algorithms in section 4. It can be shown that the algorithm takes $min\{O(r^{L+1}Ln^2), O(n^2L^2 + n(rkL^2)^{L+1})\}$. □

In Algorithm D, if $|S_v|$ is small then it may be cheaper in practice to look at all

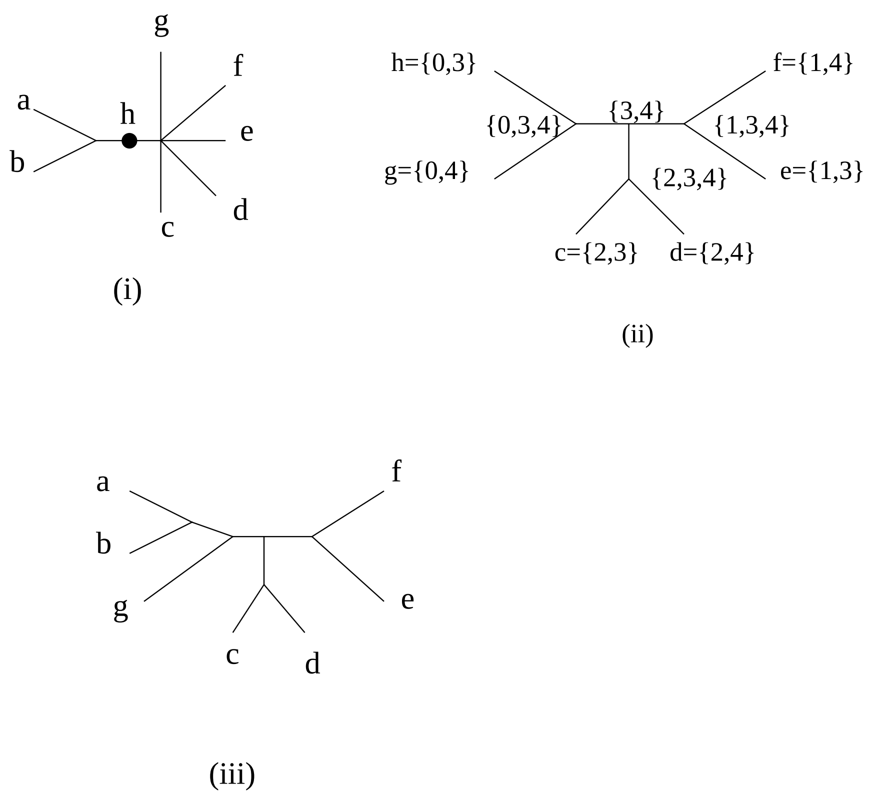


FIG. 4.1. *Example for* Algorithm D.

possible leaf-labeled topologies on S_v rather than use the algorithms of section 4 to determine the existence of perfect phylogenies on S_v.

5. Polymorphism in linguistics. Properly chosen and encoded characters in linguistics have been shown to be convex on the true tree, so that with proper scholarship we should be able to infer a *perfect phylogeny.* In recent work on an Indo-European data set, Warnow, Ringe, and Taylor [39] found that there was extensive presence of polymorphic characters. The degree of polymorphism for each polymorphic character could be determined from the data with high confidence, so that the question of inferring the correct tree amounted to determining if a perfect phylogeny existed in which each character was permitted a maximum degree of polymorphism (i.e., load) on the tree. Figure 5.1 shows the tree that they now posit. This was obtained using Algorithm D. This tree is in fact different from their earlier hypothesis and from the tree that was presented in [6]. This tree has been obtained as a result of using more data. This tree shows a limited support for Indo-Hittite, moderate support for the Italo-Celtic hypothesis, and significant support for a subgroup of Greek and Armenian.

6. Polymorphism in biology. The evolution of biological polymorphic characters can be modeled using the following operations [28]. A *mutation* changes one state into another. A *loss* drops a state from a polymorphic character from parent to child. A *duplication* replicates a state which subsequently mutates. This allows children to have higher load on a polymorphic character than their parents. We con-

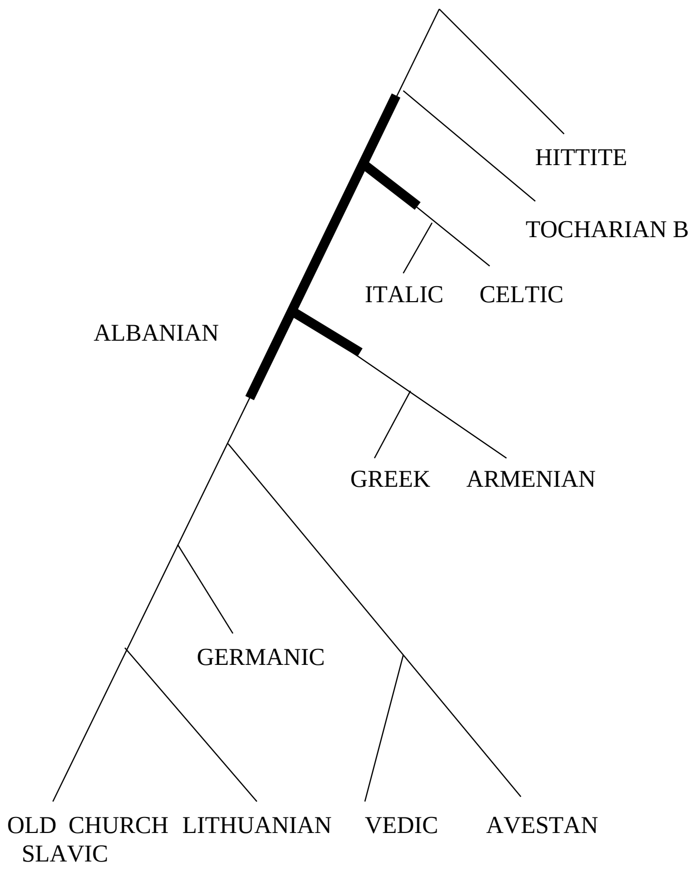


FIG. 5.1. *The tree on the Indo-European data set. Albanian can be on any of the thick edges. The tree indicates only a rooted topology without any edge lengths.*

sider two types of costs: (a) state-independent costs, in which any loss costs $cost_\ell$, any mutation costs $cost_m$, any duplication costs $cost_d$, and any match costs 0; and (b) state-dependent costs, in which the costs are dependent on the states involved.

Parsimony is a popular criterion for evaluating evolutionary trees from biomolecular data. A most parsimonious tree T minimizes $\sum_{e \in E(T)} cost(e)$. Traditionally, for monomorphic characters $cost(e)$ is the Hamming distance of the labels at the two endpoints of e. For unknown topology, the traditional parsimony problem is *NP-hard* [9, 10], but for fixed topology it is in P [16].

Consider the case where costs $cost_\ell, cost_m$, and $cost_d$ are not state-dependent. Let (u, v) be an edge in T with u above v. We define the cost $cost(\alpha, (u, v))$ of $\alpha \in C$ on (u, v) as follows: Let $X = \alpha(u) - \alpha(v), Y = \alpha(v) - \alpha(u)$, and $Z = \alpha(u) \cap \alpha(v)$.

- If $|X| = |Y|$ then $cost(\alpha, (u, v)) = cost_m|X|$ (all events are mutations, but shared states do not change).
- If $|X| > |Y|$ then $cost(\alpha, (u, v)) = cost_\ell[|X| - |Y|] + cost_m|Y|$.
- If $|X| < |Y|$ then $cost(\alpha, (u, v)) = cost_d[|Y| - |X|] + cost_m|X|$.

The cost of the edge (u, v) is then $\sum_{\alpha \in C} cost(\alpha, (u, v))$. For state-dependent costs we must also match states in the parents to states in the child for mutation and duplication events.

We consider the following problem: Given a fixed leaf-labeled topology and a maximum load l, what is the most parsimonious labeling of the internal nodes?

The problem is NP-complete for arbitrary loss, mutation, and duplication cost functions. If $cost_\ell = 0$, such as when we wish to maximize convexity, the problem becomes even harder.

THEOREM 6.1. *The following problems are NP-complete:*

- *Given a tree with leaves labeled by species each with load at most l and a value P, determine if the internal nodes can be labeled to create a phylogeny with load at most l and parsimony cost at most P for arbitrary $cost_\ell < cost_m < cost_d$.*
- *If $cost_\ell = 0$ and $cost_m \leq cost_d$ are arbitrary then given a tree with leaves labeled by species and values l and P, determine if the internal nodes can be labeled to create a phylogeny with load at most l and parsimony cost at most P. This problem remains NP-complete even if the tree is binary, no edges of weight 0 are allowed, and the input load is $1 \leq l_i \leq l$.*

Proof. In the fixed-topology setting, characters are independent. Therefore we consider only the case of a single character with r states.

Clearly the problem is in NP. We now show it is NP-hard. Our reduction is from the *three-dimensional matching problem* (3DM), known to be NP-complete [20], which is defined as follows. We are given three disjoint sets, A, B, and C, each with n elements, and a set X of m *triples*, $X = \{(a_i, b_j, c_k) : a_i \in A, b_j \in B, \text{and } c_k \in C\}$. We say that triple (a_i, b_j, c_k) *covers* a_i, b_j, and c_k. We wish to find a set of n triples that covers every element of A, B, and C exactly once. This set of n triples is called a *perfect matching*.

Given an instance of 3DM, we construct a tree T with leaves labeled by species each with load at most $m-n$. The internal nodes of T can be labeled with load $m-n$ and parsimony $(3mn - 3n^2)cost_m$ iff the instance of 3DM has a perfect matching.

We construct the tree T as follows. We begin by creating an internal *root* node. This root has $3n$ children, $a_1, \dots, a_n$, $b_1, \dots, b_n$, and $c_1, \dots, c_n$, which are all internal nodes. Let $n(a_i)$ for $1 \leq i \leq n$ be the number of triples that contain a_i. We have the following states for our character: m states $x_1, x_2, \dots, x_m$ corresponding to the m triples $x_j \in X$, and $d(a_i) \equiv m - n - n(a_i) + 1$ *dummy states* associated with each a_i (similarly we have $d(b_j) \equiv m - n - n(b_j) + 1$ dummy states for each b_j and $d(c_k) \equiv m - n - n(c_k) + 1$ dummy states for each c_k). Let $D(a_i)$ be the set of dummy states associated with a_i ($|D(a_i)| = d(a_i)$). Let $X(a_i)$ be the set of triples that contain a_i ($|X(a_i)| = n(a_i)$). For the remainder of this discussion we will concentrate on nodes a_i. The nodes b_j and c_k are treated symmetrically.

Node a_i has $n(a_i)$ leaf children. Let $x_1, x_2, \dots, x_{n(a_i)}$ be the states associated with the triples that contain a_i. The ith leaf under node a_i has all the dummy states $D(a_i)$ associated with a_i and all of $x_1, x_2, \dots, x_{n(a_i)}$ *except for* state x_i. Each child thus has load $m-n$.

It can be shown that we can label the internal nodes of this tree with load at most $m-n$ and cost at most $(3mn - 3n^2)cost_m$ iff the instance of 3DM has a perfect matching.

First suppose that $X_s \subset X$ is a set of n triples that forms a perfect matching. Label the root with states from the set $X - X_s$ and label a_i with all the leaves below it except x_q, where $x_q \in X_s$ and $a_i \in x_q$. Each internal node has load $m-n$ as required. The edge from the root to node a_i has cost $d(a_i)cost_m$ since none of the dummy states in $D(a_i)$ are in the root, all of the remaining states in a_i's label are in the root and the root has the same load as a_i (thus we have mutation rather than duplication or loss). Since $\sum_{i=1}^{n} d(a_i) = mn - n^2 + n - m$, we have (since the edges from the root to the b_j and c_k nodes are of similar cost) that the total cost of the edges from the root to its $3n$ children is $3mn - 3n^2 + 3n - 3m$. Now consider the cost of the edges from node a_i to its $n(a_i)$ children. If x_q is the triple in X_s that contains a_i, then the edge to the child missing x_q will have cost 0. All other edges have cost $cost_m$. Summing over all edges from a_i, b_j, or c_k nodes to their children, we have a cost of $(3m-3n)cost_m$. Thus the total cost is $(3mn-3n^2)$ and the parsimony bound is met.

Suppose instead that there is a labeling of the internal nodes of T so that the maximum load is $m-n$ and the total cost is at most $3mn-3n^2$.

We need the following lemma.

LEMMA 6.2. *If $cost_m < cost_d$, then in any optimal solution an internal node will always have load at least as high as the minimum load of any of its children.*

Proof. Consider a node p that is the parent of k children. Suppose there is a labeling of the nodes such that node p has a load smaller than all of its children. Thus in the cost of the tree there are at least k duplications associated with edges from p to its children. If we add to the label of p another state found in at least one of its children (such a state always exists since all children have more labels than p), then regardless of the labeling of p's parent we will decrease the cost of the tree. Adding the label costs at most $cost_d$ along the edge from node p to its parent. However, it saves at least $cost_d + (k-1)(cost_d - cost_m)$. Since $cost_d > cost_m$, adding the label always results in a net savings. Therefore we can assume that in the labeling we are given all internal nodes have load $m-n$, or we can add labels to these internal nodes and only reduce the cost. □

Thus from Lemma 6.2 all internal nodes of the input tree will have load $m-n$. Looking at the root, each state x_i is contained in the leaves of exactly three children. Each dummy state is contained in the leaves of only one child. Therefore a lowest-cost labeling of the root will have $m-n$ states x_i. If all the a_i, b_j, and c_k are also labeled by the states chosen by the root, then the minimum possible cost of the edges from the root to its children is $3mn - 3n^2 - 3m + 3n$. This is because each child a_i must mismatch on at least $m - n - n(a_i)$ labels. Summing over all children of the root, this give $3mn - 3n^2 - 3m$, but since n triples could not be in the label of the root, there is an additional cost of $3n$. Looking at the parsimony bound, even if the cost of the edges from the root to its children is minimum, the total cost of the a_i, b_j, and c_k nodes to their children can be at most $3m - 3n$. Consider a node a_i and its children. The minimum possible cost of the edges between these nodes is $n(a_i) - 1$, which comes from labeling a_i with all its dummy states and all but one of the triple states associated with it. If a_i is instead labeled with all of its triple states and all but one of its dummy states, the cost is $n(a_i)$. To achieve a total remaining cost of $3m - 3n$, however, the cost of each a_i to its children must be the minimum and therefore one of the triple states is missing from each a_i, b_j, and c_k. These must be the n triple states missing in the root or there will be higher cost associated with the edges from the root to its children and the parsimony bound will be exceeded. Thus the n triple states missing from the label of the root cover each of the a_i, b_j, and c_k and correspond to a perfect matching for the 3DM instance.

We now prove the second part of Theorem 6.1. Clearly the problem is in NP. We now show it is NP-hard. We again use a reduction from 3DM as in the proof of the first part of Theorem 6.1. We construct the tree as above with the following modifications. Each node a_i now has two children. For the case of load-1 input, each child is the root of a binary tree. Each of these trees has all the dummies in $D(a_i)$ represented in the leaf set and the states of $X(a_i)$ are arbitrarily divided among the children, appearing as a leaf just once in the subtree rooted at a_i. For other input loads, the labels of the leaves vary. For instance, for load L there are only two leaf children of a_i, one labeled with all the dummies in $D(a_i)$ and all but one state in $X(a_i)$, the other labeled with all the dummy states and the single state $x_q \in X(a_i)$ missing in the label of its sibling. For other loads, the children of a_i can also be made into binary trees where the input load is met by at least one leaf, all dummy states are represented in each child of a_i, and each state in $X(a_i)$ is represented exactly once. To make the whole tree binary, we form an arbitrary binary tree with the a_i as "leaves." (The two children of a_i will be attached.) We call this tree (without the children of a_i) the *A tree*. We make the root of the A tree a child of the global root. Similarly we form a B tree and a C tree and make them children of the global root.

Again, it can be shown that we can find labels for the internal nodes of this tree with load at most $m-n$ and cost at most $(3mn+6n-3n^2-3m)cost_m$ iff the instance of 3DM has a perfect matching.

First suppose that $X_s \subset X$ is a set of n triples that forms a perfect matching. Label the root with states from the set $X - X_s$ and label a_i with all the leaves below it except x_q where $x_q \in X_s$ and $a_i \in x_q$. Each internal node in the A tree is labeled with all of the x_q states appearing in the labels of its two children. It then picks an arbitrary set of the dummy states appearing in its two children so that its final load is $m - n$. Thus each internal node has load at most $m - n$. We now calculate the cost of this labeling. Each dummy state associated with a_i arises once in the A tree by mutation since all internal nodes in the A tree have load $m-n$, as does the global root, and the global root is not labeled with any dummy states. The total cost of all dummies is $(3mn + 3n - 3n^2 - 3m)cost_m$. In addition each of the n triple states not represented in the root arise three times in the tree by mutation for a total cost of $3ncost_m$. Thus the tree costs $(3mn + 6n - 3n^2 - 3m)cost_m$ and the parsimony bound is met.

Suppose instead that there is a labeling of the internal nodes of the tree so that the maximum load is $m-n$ and the total cost is at most $(3mn-+6n-3n^2-3m)cost_m$.

We need the following lemma.

LEMMA 6.3. *If $cost_\ell = 0$, then there exists an optimal solution where each internal node contains all the states in the subtree rooted at it or has maximum load.*

Proof. Suppose we are given a labeling where a node p has load l_p which is not the maximum load but its label does not contain a state r represented in the label of one of its children. Let P be the set of all nodes on the path from p to its first ancestor with load at least $l_p + 1$ or up to the root if no such ancestor exists. Add state r to the label of p. Choose some state that is in node p that is not in its parent (one always exists if the parent has load at most l_p) and add it to the label of its parent, and so on so that each node in set P has its load increased by 1. This increase in labeling costs at most $cost_m$ on the label from the highest node in P to its parent (nothing if the highest node in P is the root), and it saves $cost_m$ on the edge (p, c). Thus we have not increased the cost of the tree. Starting this process with internal nodes lowest in the tree gives us the above claim. ∎

Therefore we can assume that every internal node is labeled with all states in its subtree or has maximum load. In particular, we can assume that the root has maximum load $m - n$. Since each triple state x_q is represented in the leaves of all three children and the dummy states in only one child each, then the minimum possible cost associated with dropping states at the root is $(3mn+6n-3n^2-3m)cost_m$ (all the dummy states must arise somewhere in the tree and the n triple states not present in the root label must arise three times in the tree). Therefore, to meet the parsimony bounds, there can be no additional cost throughout the remainder of the tree. Considering a single node a_i, there are $m-n+1$ states in the subtree rooted at a_i. Since it can have load at most $m-n$, one of these states must be missing from the label at a_i. If one of the dummy states is dropped, then there will be an extra cost of $cost_m$ beyond what is forced by the root (the root allowed each dummy state to arise once and each dummy state appears in the subtrees rooted in both children of node a_i). Therefore node a_i must be labeled with all its dummy states and all but one of the $x_q \in X(a_i)$. These triple states must be passed up the A tree to the root, but there is sufficient capacity to do so (the states that are dropped at internal nodes of the A tree of dummy states, which ultimately had to be dropped anyway). To achieve the parsimony bound, this x_q missing from the label of node a_i must be one of the triples not represented at the root. Therefore the n triple states missing from the root label correspond to the perfect matching in the 3DM instance. ◻

We now consider algorithms for fixed load l. Since the topology is given, characters can be solved independently. We first give the algorithm for the most general possible cost function and then consider special cases which can be solved more efficiently. All the algorithms are standard bottom-up dynamic programming. A final pass downward from the root produces an optimal labeling of the tree in time $O(nlk)$.

THEOREM 6.4. *Given a tree on n species with k characters where r is the maximum number of states for any character,*

1. *there exists an $O(nkr^{2l})$-time algorithm to compute the most parsimonious load-l labeling for the tree for arbitrary state-dependent costs;*
2. *there exists an $O(nkl(2r)^l)$-time algorithm to compute the most parsimonious load-l labeling for the tree for arbitrary fixed costs $cost_\ell \leq cost_m \leq cost_d$;*
3. *there exists an $O(nk(2r)^l)$-time algorithm to compute the most parsimonious load-l labeling for the given tree when $cost_\ell = 0$.*

Proof. When the cost function is state-dependent, we convert our input to a weighted monomorphic parsimony problem. We define a new set of $O(r^l)$ states, one for each possible label of a node. Given two labels l_p and l_c, we can determine the cost of a parent-child edge with labels l_p and l_c. We must match states for mutations and duplications. We thus compute a matrix of edge costs. Because loss and duplication costs are not the same, this matrix is not symmetric in general. We then use the algorithm of Sankoff and Cedergren [34] for weighted parsimony which runs in time $O(nkj^2)$ for n species, k characters, and j states/characters. In our case we have r^l states, where r was the original number of states in the polymorphic character. Thus this algorithm has time $O(nkr^{2l})$.

The bottom-up dynamic programming algorithm for weighted parsimony proceeds as follows. For an internal node v, let $c(v, l_v)$ be cost of the best labeling of the subtree rooted at v provided that node v is labeled l_v. Then we have $c(v, l_v) = \sum_{v' \text{child of } v}(\min_{l_{v'}} c(v', l_{v'}) + w(l_v, l_{v'}))$, where $w(l_v, l_{v'})$ is the cost of the edge with parent label l_v and child label $l_{v'}$. Thus we consider every possible label for an internal node and compare it against every possible label for its children. For arbitrary weight function w, this will cost r^{2l} for each parent-child interaction.

For the case of arbitrary $cost_\ell \leq cost_m \leq cost_d$ (not state-dependent), we can reduce the overall time to $O(nkl(2r)^l)$. Again, we wish to consider every possible label for node v but we need not consider every possible label for its children. Suppose that for each child we know the best choice of label for each of load $1, 2, \ldots, l$, where some specific subset (possibly empty) of the label is specified. For example, we know the best load-3 labeling of the child where a and b are two of the three states. This is $O(lr^l)$ information. To find the best labeling of the subtree rooted at v provided v is labeled by l_v, the only labels we need to consider for the children of v are the best ones for each possible subset of l_v and each possible load. For example, if $l_v = \{a, b\}$, $l = 3$, and $*$ can be any state, then the only labels that must be considered for a child are $*$ (best tree with load-1 label), $**$, $***$, a, $a*$, $a**$, b, $b*$, $b**$, ab, and $ab*$. More formally, let $c(v, L, x)$ be the cost of the best subtree rooted at v where the label of v contains state set L and x other states. Then the cost of label l_v and node v is

$$c(v, l_v) = \sum_{\text{children } v'} \min_{L \subseteq l_v} \min_{0 \le l' \le l - |L|} (c(v', L, l') + w(l_v, L, l')),$$

where

$$w(l_v, L, l') = \begin{cases} l' cost_m + (|l_v| - |L| - l') cost_\ell & \text{if } |L| + l' \le |l_v| \\ (|l_v| - L) cost_m + (|L| + l' - |l_v|) cost_d & \text{otherwise.} \end{cases}$$

Thus to compute the cost of a label, each parent must check $O(l2^l)$ labels in each child. Once the label l_v is computed, it contributes to $O(2^l)$ minimizations used by its parent (each subset of l_v with load $|l_v|$). Since each of the $O(n)$ edges is checked $O(l2^l)$ times for each of the r^l possible parent labels, the overall cost is $O(nkl(2r)^l)$.

To prove the final part of the theorem, when $cost_\ell = 0$ (for example when we wish to maximize convexity), we note that whenever we have $cost_\ell = 0$ there exists an optimal solution where each internal node contains all the states in the subtree rooted at it or has maximum load. We begin by locating the highest internal nodes v with at most l states in the subtree rooted at them. We label node v by these states and make it a leaf by removing all its children. Now we can assume all internal nodes have load l. This saves a factor of l using the preceding algorithm since there is now only one value of l'. $\square$

7. Discussion. In this paper we introduced an algorithmic study of the problem of inferring the evolutionary tree in the presence of polymorphic data. We considered parsimony analysis for polymorphic data on fixed topologies and presented algorithms as well as hardness results. We also presented algorithms for inferring perfect phylogenies from such data, and we note that it is reasonable to seek perfect phylogenies for certain types of data. The results of our analysis of the an expanded Indo-European data set studied by Warnow, Ringe, and Taylor has led to a new hypothesis for the evolution of Indo-European languages.

REFERENCES

[1] R. Agarwala and D. Fernández-Baca, *A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed*, SIAM J. Comput., 23 (1994), pp. 1216–1224.

[2] R. Agarwala and D. Fernández-Baca, *Fast and Simple Algorithms for Perfect Phylogeny and Triangulating Colored Graphs*, Technical report TR94-51, DIMACS, to appear in special issue on algorithmic aspects of computational biology, Internat. J. Found. Comput. Sci.

[3] S. Arnborg, D. Corneil, and A. Proskurowski, *Complexity of finding embeddings in a k-Tree*, SIAM J. Alg. Discrete Methods, 8 (1987), pp. 277–284.

[4] H. Bodlaender, M. Fellows, and T. Warnow, *Two strikes against perfect phylogeny*, in Proc. 19th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, New York, 1992, pp. 273–283.

[5] H. Bodlaender and T. Kloks, *A simple linear time algorithm for triangulating three-colored graphs*, in Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, 1992, pp. 415–423, to appear in J. Algorithms.

[6] M. Bonet, C. Phillips, T. Warnow, and Shibu Yooseph, *Constructing evolutionary trees in the presence of polymorphic characters*, in Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing, Philadelphia, 1996, pp. 220–229.

[7] P. Buneman, *A characterization of rigid circuit graphs*, Discrete Math., 9 (1974), pp. 205–212.

[8] L. L. Cavalli-Sforza, P. Menozzi, and A. Piazza, *The History and Geography of Human Genes*, Princeton University Press, Princeton, NJ, 1994.

[9] W. H. E. Day, *Computationally difficult parsimony problems in phylogenetic systematics,* J. Theoret. Biol., 103 (1983), pp. 429–438.

[10] W. H. E. Day, D. S. Johnson, and D. Sankoff, *The computational complexity of inferring phylogenies by parsimony,* Math. Biosci., 81 (1986), pp. 33–42.

[11] W. H. E. Day and D. Sankoff, *Computational complexity of inferring phylogenies by compatibility*, Systematic Zool., 35 (1986), pp. 224–229.

[12] A. Dress and M. Steel, *Convex tree realizations of partitions*, Appl. Math. Lett., 5 (1992), pp. 3–6.

[13] G. F. Estabrook, *Cladistic methodology: A discussion of the theoretical basis for the induction of evolutionary history*, Annu. Rev. Ecol. Syst., 3 (1972), pp. 427–456.

[14] G. F. Estabrook, C. S. Johnson, Jr., and F. R. McMorris, *An idealized concept of the true cladistic character*, Math. Biosci., 23 (1975), pp. 263–272.

[15] J. Felsenstein, *Alternative methods of phylogenetic inference and their interrelationships,* Systematic Zool., 28 (1979), pp. 49–62.

[16] W. Fitch, *Towards defining the course of evolution: Minimum change for a specified tree topology*, Systematic Zool., 20 (1971), pp. 406–416.

[17] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[18] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks, 21 (1991), pp. 19–28.

[19] R. M. Idury and A. A. Schäffer, *Triangulating three-colored graphs in linear time and linear space*, SIAM J. Discrete Math., 6 (1993), pp. 289–293.

[20] R. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[21] S. Kannan and T. Warnow, *Inferring evolutionary history from DNA sequences*, SIAM J. Comput., 23 (1994), pp. 713–737.

[22] S. Kannan and T. Warnow, *Triangulating 3-colored graphs*, SIAM J. Discrete Math., 5 (1992), pp. 249–258.

[23] S. Kannan and T. Warnow, *A fast algorithm for the computation and enumeration of perfect phylogenies*, SIAM J. Comput., 26 (1997), pp. 1749–1763.

[24] W. J. Le Quesne, *A method of selection of characters in numerical taxonomy*, Systematic Zool., 18 (1969), pp. 201–205.

[25] W. J. Le Quesne, *Further studies based on the uniquely derived character concept*, Systematic Zool., 21 (1972), pp. 281–288.

[26] M. F. Mickevich and C. Mitter, *Treating polymorphic characters in systematics: A phylogenetic treatment of electrophoretic data*, in Advances in Cladistics, V. A. Funk and D. R. Brooks, eds., New York Botanical Garden, New York, 1981, pp. 45–58.

[27] F. R. McMorris, T. Warnow, and T. Wimer, *Triangulating vertex-colored graphs*, SIAM J. Discrete Math., 7 (1994), pp. 296–306.

[28] M. Nei, *Molecular Evolutionary Genetics*, Columbia University Press, New York, 1987.

[29] A. Proskurowski, *Separating subgraphs in k-trees: Cables and caterpillars*, Discrete Math., 49 (1984), pp. 275–285.

[30] D. Ringe, *personal communication*, 1995.

[31] D. J. Rose, *On simple characterization of k-trees*, Discrete Math., 7 (1974), pp. 317–322.

[32] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs,* SIAM J. Comput., 5 (1976), pp. 266–283.

[33] A. K. Roychoudhury and M. Nei, *Human Polymorphic Genes: World Distribution*, Oxford University Press, London, UK, 1988.

[34] D. SANKOFF AND R. J. CEDERGREN, *Simultaneous comparison of three or more sequences related by a tree*, in Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, D. Sankoff and J.B. Kruskal, eds., Addison–Wesley, Reading MA, 1983, pp. 253–263

[35] D. SANKOFF AND P. ROUSSEAU, *Locating the vertices of a Steiner tree in arbitrary space,* Math. Programming, 9 (1975), pp. 240–246.

[36] M. A. STEEL, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.

[37] T. WARNOW, *Constructing phylogenetic trees efficiently using compatibility criteria,* New Zealand J. Botany, 31 (1993), pp. 239–248.

[38] T. WARNOW, *Mathematical approaches to comparative linguistics,* in Proceedings of the National Academy of Sciences, vol. 94, 1997, pp. 6585–6590.

[39] T. WARNOW, D. RINGE, AND A. TAYLOR, *A character based method for reconstructing evolutionary history for natural languages*, Technical report, Institute for Research in Cognitive Science, 1995, and Proc. ACM/SIAM Symposium on Discrete Algorithms, Atlanta, GA, 1996.

[40] J. J. WEINS, *Polymorphic characters in phylogenetic systematics*, Systematic Biology, 44 (1995), pp. 482–500.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 132–158

THE COMPLEXITY OF TREE AUTOMATA AND LOGICS OF PROGRAMS*

E. ALLEN EMERSON† AND CHARANJIT S. JUTLA‡

Abstract. The complexity of testing nonemptiness of finite state automata on infinite trees is investigated. It is shown that for tree automata with the pairs (or complemented pairs) acceptance condition having m states and n pairs, nonemptiness can be tested in deterministic time $(mn)^{O(n)}$; however, it is shown that the problem is in general NP-complete (or co-NP-complete, respectively). The new nonemptiness algorithm yields exponentially improved, essentially tight upper bounds for numerous important modal logics of programs, interpreted with the usual semantics over structures generated by binary relations. For example, it follows that satisfiability for the full branching time logic CTL* can be tested in deterministic double exponential time. Another consequence is that satisfiability for propositional dynamic logic (PDL) with a repetition construct (PDL-delta) and for the propositional Mu-calculus ($L\mu$) can be tested in deterministic single exponential time.

Key words. complexity, tree automata, logics of programs, games

AMS subject classifications. 68Q68, 68Q60, 03B45

PII. S0097539793304741

1. Introduction.

There has been a resurgence of interest in automata on infinite objects [1] due to their intimate relation with temporal and modal logics of programs. They provide an important and uniform approach to the development of decision procedures for testing satisfiability of the propositional versions of these logics [43, 33]. Such logics and their corresponding decision procedures are not only of inherent mathematical interest, but are also potentially useful in the specification, verification, and synthesis of concurrent programs (cf. [27, 8, 25, 21]).

In the case of branching time temporal logic, the standard paradigm nowadays for testing satisfiability is the reduction to the nonemptiness problem for finite state automata on infinite trees; i.e., one builds a tree automaton which accepts essentially all models of the candidate formula and then tests nonemptiness of the tree automaton. Thus in order to improve the complexity there are two issues: (1) the size of the tree automaton and (2) the complexity of testing nonemptiness of the tree automaton.

In this paper we obtain new, improved, and essentially tight bounds on testing nonemptiness of tree automata that allow us to close an exponential gap which has existed between the upper and lower bounds of the satisfiability problem of numerous important modal logics of programs. These logics include CTL* (the full branching time logic [9]), PDL-delta (propositional dynamic logic with an infinite repetition construct [33]), and the propositional Mu-calculus ($L\mu$) (a language for characterizing temporal correctness properties in terms of extremal fixpoints of predicate transformers [19] (cf. [7, 2])).

*Received by the editors August 15, 1993; accepted for publication (in revised form) April 30, 1997; published electronically September 14, 1999. This work was supported in part by NSF grants CCR-941-5496 and CCR-980-4736, ONR URI contract N00014-86-K-0763, and Netherlands NWO grant nf-3/nfb 62-500.

http://www.siam.org/journals/sicomp/29-1/30474.html

†Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712 (emerson @cs.utexas.edu), and Mathematics and Computing Science Department, Eindhoven University of Technology, Eindhoven 5600 MB, The Netherlands.

‡IBM T.J. Watson Research Center, Yorktown Heights, NY 10598-0218 (csjutla@watson. ibm.com).

To obtain these improvements, we focus on the complexity of testing nonemptiness of tree automata. We first note, however, that the size of an automaton has two parameters: the number of states in the automaton's transition diagram and the number of pairs in its acceptance condition. We next make the following important observation: for most logics of programs, the number of pairs is logarithmic in the number of states.

We go on to analyze the complexity of testing nonemptiness of pairs tree automata [30] and show that it is NP-complete. However, a multiparameter analysis shows that there is an algorithm that runs in time $(mn)^{O(n)}$ which is polynomial in the number of states m and exponential in the number of pairs n in the acceptance condition of the automaton. The algorithm is based on a type of "pseudomodel checking" for certain restricted Mu-calculus formulae. Moreover, since the problem is NP-complete, it is unlikely to have a better algorithm which is polynomial in both parameters. The previous best known algorithm was in NP [6, 41].

The above nonemptiness algorithm now permits us to obtain a deterministic double exponential time decision procedure for CTL*, by using the reduction from CTL* to tree automata obtained in [13], in which the size of the automaton is double exponential in the length of the formula and the number of pairs is only exponential in the length of the formula. The bound follows by simple arithmetic, since a double exponential raised to a single exponential power is still a double exponential.

This amounts to an exponential improvement over the best previously known algorithm which was in nondeterministic double exponential time [6, 41], i.e., three exponentials when determinized. It is also essentially tight, since CTL* was shown to be double exponential time hard [41]; thus CTL* is deterministic double exponential time complete.

The above result has been obtained using only the classical pairs tree automata of Rabin [30]. However, we also consider the complemented pairs tree automata of Streett [33], which were specifically introduced to facilitate formulation of temporal decision procedures. We show that the nonemptiness problem of complemented pairs automata is co-NP-complete by reducing the complement of the problem to nonemptiness of the pairs automata and vice versa. The reduction employs the fact that infinite Borel games are determinate (Martin's theorem [22]). This reduction also gives a deterministic algorithm which is polynomial in the number of states and exponential in the number of pairs. We can thus reestablish the above upper bound for CTL* using complemented pairs automata as well.

Using the recent single exponential general McNaughton [23] construction of Safra [32] (i.e., construction for determinizing a Büchi finite automaton on infinite strings), our new nonemptiness algorithm also gives us a deterministic single exponential time decision procedure for both PDL-delta and the Mu-calculus, since the Safra construction allows us to reduce satisfiability of these logics to testing nonemptiness of a tree automaton with exponentially many states and polynomially many pairs. This represents an exponential improvement over the best known deterministic algorithms for these logics, which took deterministic double exponential time, corresponding to the nondeterministic exponential time upper bounds of [41]. The bounds are essentially tight also, since the exponential time lower bound follows from that established for ordinary PDL by Fischer and Ladner [14].

It is interesting to note that for most logics (including PDL-delta and the Mu-calculus but excluding CTL*), our nonemptiness algorithm(s) and Safra's construction both play a crucial role. Each is independent of the other. Moreover, each is needed and neither alone suffices. Our algorithm improves the complexity of testing nonemptiness by an exponential factor, while Safra's construction independently applies to reduce the size of the automaton by an exponential factor. For example, the "traditional" result of Streett [33] gave a deterministic triple exponential algorithm for PDL-delta. Our algorithm alone improves it to deterministic double exponential time. Alternatively, Safra's construction alone improves it to deterministic double exponential time. As shown in this paper, the constructions can be applied together to get a cumulative double exponential speedup for PDL-delta. In the case of CTL*, we already had the effect of Safra's construction, because [13] gave a way to determine with only a single exponential blowup the Büchi string automaton corresponding to a linear temporal logic formula by using the special structure of such automata (unique accepting run). Thus Safra's construction provides no help for the complexity of the CTL* logic.

The remainder of the paper is organized as follows. In section 2 we give preliminary definitions and terminology. In section 3 we establish a "small model theorem" for (pairs) tree automata. In section 4 we give the main technical results on testing nonemptiness of pairs tree automata. In section 5 we give the main results on nonemptiness of complemented pairs tree automata. Applications of the algorithms to testing satisfiability of modal logics of programs, including CTL*, PDL-delta, and the Mu-calculus, are described in section 6. Some concluding remarks are given in section 7.

2. Preliminaries.

2.1. Logics of programs.

2.1.1. Full branching time logic. The *full branching time logic* CTL* [9] derives its expressive power from the freedom of combining modalities which quantify over paths and modalities which quantify states along a particular path. These modalities are A, E, F, G, X_s, and U_w ("for all futures," "for some future," "sometime," "always," "strong nexttime," and "weak until," respectively), and they are allowed to appear in virtually arbitrary combinations. Formally, we inductively define a class of state formulae (true or false of states) and a class of path formulae (true or false of paths):

(S1) Any atomic proposition P is a state formula.
(S2) If p, q are state formulae, then so are $p \wedge q$, $\neg p$.
(S3) If p is a path formula, then Ep is a state formula.
(P1) Any state formula p is also a path formula.
(P2) If p, q are path formulae, then so are $p \wedge q$, $\neg p$.
(P3) If p, q are path formulae, then so are $X_s p$ and $p U_w q$.

The semantics of a formula are defined with respect to a structure $M = (S, R, L)$, where S is a nonempty set of states, R is a nonempty binary relation on S, and L is a labeling which assigns to each state a set of atomic propositions true in the state. A *fullpath* $(s_1, s_2, \ldots)$ is a maximal sequence of states such that $(s_i, s_{i+1}) \in R$ for all i. A fullpath is infinite unless for some s_k there is no s_{k+1} such that $(s_k, s_{k+1}) \in R$. We write $M, s \models p$ ($M, x \models p$) to mean that state formula p (path formula p) is true in structure M at state s (of fullpath x, respectively). When M is understood, we write simply $s \models p$ ($x \models p$). We define $\models$ inductively using the convention that $x = (s_1, s_2, \ldots)$ denotes a fullpath and x^i denotes the suffix fullpath $(s_i, s_{i+1}, \ldots)$, provided $i \leq |x|$, where $|x|$, the length of x, is ω when x is infinite and k when x is finite and of the form $(s_1, \ldots, s_k)$; otherwise x^i is undefined.

For a state s,

(S1) $s \models P$ iff $P \in L(s)$ for atomic proposition P,

(S2) $s \models p \wedge q$ iff $s \models p$ and $s \models q$,

$s \models \neg p$ iff not $(s \models p)$,

(S3) $s \models Ep$ iff for some fullpath x starting at s, $x \models p$.

For a fullpath $x = (s_1, s_2, \ldots)$,

(P1) $x \models p$ iff $s_1 \models p$ for any state formula p,

(P2) $x \models p \wedge q$ iff $x \models p$ and $x \models q$,

$x \models \neg p$ iff not $(x \models p)$,

(P3) $x \models X_s p$ iff x^2 is defined and $x^2 \models p$,

$x \models (p\, U_w\, q)$ iff for all $i \in [1 : |x|]$, if for all $j \in [1 : i]$ $x^j \models \neg q$, then $x^i \models p$.

We say that state formula p is *valid*, and write $\models p$, if for every structure M and every state s in M, $M, s \models p$. We say that state formula p is *satisfiable* iff for some structure M and some state s in M, $M, s \models p$. In this case we also say that M defines a *model* of p. We define validity and satisfiability for path formulae similarly.

We write $f \doteq g$ to mean that formula f abbreviates formula g. Other connectives can then be defined as abbreviations in the usual way: $p \vee q \doteq \neg(\neg p \wedge \neg q)$, $p \Rightarrow q \doteq \neg p \vee q$, $p \Leftrightarrow q \doteq (p \Rightarrow q) \wedge (q \Rightarrow p)$, $Ap \doteq \neg E \neg p$, $Gp \doteq p\, U_w\, false$, and $Fp \doteq \neg G \neg p$. Further operators may also be defined as follows:

$X_w p \doteq \neg X_s \neg p$ is the weak nexttime,

$p U_s q \doteq (p U_w q) \wedge Fq$ is the strong until,

$\overset{\infty}{F} p \doteq GFX_s p$ means infinitely often p,

$\overset{\infty}{G} p \doteq FGX_s p$ means almost everywhere p,

$inf \doteq GX_s true$ means the path is infinite, and

$fin \doteq FX_w false$ means the path is finite.

2.1.2. Propositional dynamic logic plus repeat. As opposed to CTL*, in which the models represent behaviors of the programs, in PDL-delta the programs are explicit in the models. The modalities in PDL-delta quantify the states reachable by programs explicitly stated in the modality. Thus, for a program B (which is obtained from atomic programs and tests using regular expressions), $\langle B \rangle p$ ($[B]p$) states that there is an execution of B leading to p (after all executions of B, p holds). Also included is the infinite repetition construct *delta* ($\triangle$) which makes PDL-delta much more expressive than PDL. $\triangle B$ states that it is possible to execute B repetitively infinitely many times. PDL-delta formulae are interpreted over structures $M = (S, R, L)$, where S is a set of states, $R : Prog \to 2^{S \times S}$ is a transition relation, $Prog$ is the set of atomic programs, and L is a labeling of S with propositions in $Prop$. For more details see [33].

2.1.3. Propositional Mu-calculus. A *least fixpoint* construct can be used to increase the power of simple modal logics. Thus, by adding this construct to PDL, we get $L\mu$, the *propositional Mu-calculus* [19], a logic which subsumes PDL-delta. A variant formulation of the Mu-calculus, which we use here, adds the least fixpoint construct to a simple subset of CTL*, including just nexttime (AX_s), the boolean connectives, and propositions (cf. [7]). The least fixpoint construct has the syntax $\mu Y.f(Y)$, where $f(Y)$ is any formula syntactically monotone in the propositional variable Y, i.e., all occurrences of Y in $f(Y)$ fall under an even number of negations. It is interpreted as the smallest set S of states such that $S = f(S)$. By the well-known Tarski–Knaster theorem, $\mu Y.\, f(Y) = \bigcup_i f^i(\text{false})$, where i ranges over all ordinals and f^i (intuitively) denotes the i-fold composition of f with itself; when the domain is finite we may take i as ranging over just the natural numbers. Its dual, the greatest fixpoint, is denoted $\nu Y.f(Y)$ ($\equiv \neg\mu Y.\neg f(\neg Y)$). Thus, e.g., $\mu Y.[B]Y$ is equivalent to $\neg \triangle B$ of PDL-delta. Similarly, using temporal logic, $\mu Y.P \vee AX_sY$ is equivalent to AFP (i.e., along all paths P eventually holds). Many correctness properties of concurrent programs can be characterized in terms of the Mu-calculus, including all those expressible in CTL* and PDL-delta. For more details, see, for example, [7, 19, 35, 12].

The *formulae* of the (propositional) Mu-calculus are

(1) propositional constants $P, Q, \dots,$

(2) propositional variables $Y, Z, \dots,$

(3) $\neg p$, $p \vee q$, and $p \wedge q$, where p and q are any formulae,

(4) EX_sp and AX_sp, where p is any formula,

(5) $\mu Y.f(Y)$ and $\nu Y.f(Y)$, where $f(Y)$ is any formula syntactically monotone in the propositional variable Y, i.e., all occurrences of Y in $f(Y)$ fall under even number of negations.

In what follows, we will use σ as a generic symbol for μ or ν. In a fixed point expression $\sigma Y.f(Y)$, we say that each occurrence of Y is *bound* to σY. If an occurrence of Y is not bound, then it is *free*. A *sentence* (or *closed formula*) is a formula containing no free propositional variables, i.e., no variables unbound by a μ or a ν operator.

Sentences are interpreted over structures $M = (S, R, L)$ as for CTL*. As usual we will write $M, s \models p$ to mean that in structure M at state s sentence p holds true. To give the technical definition of $\models$ we need some preliminaries.

The power set of S, 2^S, may be viewed as the complete lattice $(2^S, S, \phi, \subseteq, \cup, \cap)$. Intuitively, we identify a proposition with the set of states which make it true. Thus, $false$, which corresponds to the empty set, is the bottom element, $true$, which corresponds to S, is the top element, $\cup$ is join, $\cap$ is meet, and implication (for all $s \in S(P(s) \Rightarrow Q(s))$), which corresponds to simple set-theoretic containment ($P \subseteq Q$), provides the partial ordering on the lattice.

Let $\tau : 2^S \rightarrow 2^S$ be given; then we say that τ is *monotonic* provided $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$. A monotonic functional τ always has both a least fixpoint $\mu X.\tau(X)$ and a greatest fixpoint $\nu X.\tau(X)$.

For a formula or function $p(Y)$, we write $p^0(Y) = false$, $p^1(Y) = p(Y)$, $p^{i+1}(Y) = p(p^i)(Y)$ for successor ordinal $i+1$, and $p^j(Y) = \bigcup_{k<j} p^k(Y)$ for limit ordinal j.

THEOREM 2.1 (Tarski–Knaster). *Let $\tau : 2^S \rightarrow 2^S$ be a given monotonic functional. Then*

(a) $\mu Y.\tau(Y) = \bigcap\{Y : \tau(Y) = Y\} = \bigcap\{Y : \tau(Y) \subseteq Y\}$,

(b) $\nu Y.\tau(Y) = \bigcup\{Y : \tau(Y) = Y\} = \bigcup\{Y : \tau(Y) \supseteq Y\}$,

(c) $\mu Y.\tau(Y) = \bigcup_{i \leq |S|} \tau^i(false)$, *and*

(d) $\nu Y.\tau(Y) = \bigcap_{i \leq |S|} \tau^i(true)$.

A formula p with free variables $Y_0, Y_1, \dots, Y_n$ is thus interpreted as a mapping p^M from $(2^S)^{n+1}$ to 2^S, i.e., it is interpreted as a predicate transformer. We write $p(Y_0, Y_1, \dots, Y_n)$ to denote that all free variables of p are among $Y_0, Y_1, \dots, Y_n$. Let $V_0, V_1, \dots, V_n$ be subsets of S; then a *valuation* $\Upsilon = V_0, V_1, \dots, V_n$ is an assignment of $V_0, V_1, \dots, V_n$ to the free variables $Y_0, Y_1, \dots, Y_n$, respectively. $\Upsilon[Y_i \leftarrow V_i']$ denotes the valuation identical to Υ, except that Y_i is assigned V_i'. We use $p^M(\Upsilon)$ to denote the value of p in structure M on the arguments $V_0, V_1, \dots, V_n$. We drop M when it is understood from context. We then let $M, s \models p(\Upsilon)$ iff $s \in p^M(\Upsilon)$, and we define $\models$ inductively as follows:

(1) $s \models P(\Upsilon)$ iff $P \in L(s)$,

(2) $s \models Y(\Upsilon)$ iff $s \in \Upsilon(Y)$,
(3) $s \models (\neg p)(\Upsilon)$ iff $s \not\models p(\Upsilon)$,
$s \models (p \vee q)(\Upsilon)$ iff $s \models p(\Upsilon)$ or $s \models q(\Upsilon)$,
$s \models (p \wedge q)(\Upsilon)$ iff $s \models p(\Upsilon)$ and $s \models q(\Upsilon)$,
(4) $s \models (EX_s p)(\Upsilon)$ iff $\exists t(s,t) \in R$ and $t \models p(\Upsilon)$,
$s \models (AX_s p)(\Upsilon)$ iff (a) $\exists u\ (s,u) \in R$ and $u \models p$ and (b) for all $t\ (s,t) \in R$ implies $t \models p(\Upsilon)$,
(5) $s \models\ (\mu Y.f(Y))(\Upsilon)$ iff $s \in \bigcap\{S' \subseteq S | S' = \{t : t \models f(Y)(\Upsilon[Y \leftarrow S'])\}\}$,
$s \models\ (\nu Y.f(Y))(\Upsilon)$ iff $s \in \bigcup\{S' \subseteq S | S' = \{t : t \models f(Y)(\Upsilon[Y \leftarrow S'])\}\}$.

2.1.4. Conventions. To avoid a proliferation of unnecessary parentheses, we order the connectives from greatest to lowest *binding power* as follows: $\neg$ binds tighter than $F, G, X_w, X_s, \overset{\infty}{F}, \overset{\infty}{G}$, which bind tighter than $\wedge$, which binds tighter than $\vee$, which binds tighter than $\Rightarrow$, which binds tighter than U_w, U_s, which bind tighter than A, E, which bind tighter than μ, ν, which bind tighter than $\Leftrightarrow$.

If we write $M, s \models p$, it is implicit that s is a state of M. That is, $s \in S$ where $M = (S, R, L)$. A convenient abuse of notation is to write $s \in M$ in some places.

If p is a formula, then p^M denotes $\{s : M, s \models p\}$, the set of states s in M at which p is true.

We can write $M, s_1, \ldots, s_k \models p$ to abbreviate $M, s_1 \models p$ and $\ldots$ and $M, s_k \models p$.

We write $p \equiv q$ for $\models p \Leftrightarrow q$. In the context of a structure M, we can also write $p \equiv_M q$ for $p^M = q^M$. If M is understood, then we can drop the M and write just $p \equiv q$. It should be clear from context whether equivalence over all structures or over M is meant. We use $p_1 \equiv p_2 \equiv \cdots \equiv p_k$ as shorthand for $p_1 \equiv p_2$ and $\ldots$ and $p_{k-1} \equiv p_k$.

Technically, we distinguish between an atomic proposition symbol P and the associated set, viz., P^M, of states which are labeled with it in structure M. It is often convenient notation to use the uppercase sans serif symbol P corresponding to P to denote the set of states that are labeled with P.

Remark: We note the following identities:

$EX_w false \equiv AX_w false$ and asserts that a state has no successors.

$EX_s true \equiv AX_s true$ and asserts that a state has one or more successors.

2.2. Automata on infinite trees. We consider finite automata on labeled, infinite binary trees.[1] The set $\{0,1\}^*$ may be viewed as an infinite binary tree, where the empty string λ is the root node and each node u has two successors: the 0-successor $u0$ and the 1-successor $u1$. A finite (infinite) path through the tree is a finite (respectively, infinite) sequence $x = u_0, u_1, u_2, \ldots$ such that each node u_{i+1} is a successor of node u_i. If Σ is an alphabet of symbols, an infinite binary Σ-*tree* is a labeling L which maps $\{0,1\}^* \longrightarrow \Sigma$.

A *finite automaton* $\mathcal{A}$ on infinite binary Σ-trees consists of a tuple $(\Sigma, Q, \delta, q_0, \Phi)$, where

Σ is the finite, nonempty input alphabet labeling the nodes of the input tree,
Q is the finite, nonempty set of states of the automaton,
$\delta : Q \times \Sigma \rightarrow 2^{Q \times Q}$ is the nondeterministic transition function,

[1]We consider here only binary trees to simplify the exposition and for consistency with the classical theory of tree automata. CTL* and the other logics we study have the property that their models can be unwound into an infinite tree. In particular, in [13] it was shown that a CTL* formula of length k is satisfiable iff it has an infinite tree model with finite branching bounded by k, i.e., iff it is satisfiable over a k-ary tree. Our results on tree automata apply to such k-ary trees as explained at the end of the proof of Theorem 4.1.

$q_0 \in Q$ is the start state of the automaton, and

Φ is an acceptance condition described subsequently.

A *run* of $\mathcal{A}$ on the input Σ-tree L is a function $\rho : \{0,1\}^* \to Q$ such that for all $v \in \{0,1\}^*$, $(\rho(v0), \rho(v1)) \in \delta(\rho(v), L(v))$ and $\rho(\lambda) = q_0$. We say that $\mathcal{A}$ *accepts* input tree L iff there exists a run ρ of $\mathcal{A}$ on L such that for all infinite paths x starting at the root of L if $r = \rho \circ x$ is the sequence of states $\mathcal{A}$ goes through along path x, then the acceptance condition Φ holds along r.

For a *pairs* automaton (cf. [23, 30]) acceptance is defined in terms of a finite list $((\mathsf{RED}_1, \mathsf{GREEN}_1), \ldots, (\mathsf{RED}_k, \mathsf{GREEN}_k))$ of pairs of sets of automaton states (which may be thought of as pairs of colored lights where $\mathcal{A}$ flashes the red light of the first pair upon entering any state of the set RED_1, etc.): r satisfies the pairs condition iff there exists a pair $i \in [1..k]$ such that RED_i flashes finitely often and GREEN_i flashes infinitely often. We assume the pairs acceptance condition is given formally by a temporal logic formula $\Phi = \bigvee_{i \in [1..k]} (GF\ \mathrm{GREEN}_i \wedge \neg GF\ \mathrm{RED}_i)$.[2] Similarly, a *complemented pairs* (cf. [33]) automaton has the negation of the pairs condition as its acceptance condition; i.e., for all pairs $i \in [1..k]$, GREEN_i flashes infinitely often implies that RED_i flashes infinitely often, too. The complemented pairs acceptance condition is given formally by a temporal logic formula $\Phi = \bigwedge_{i \in [1:k]} GF\ \mathrm{GREEN}_i \Rightarrow GF\ \mathrm{RED}_i$.

3. Small model theorems.

3.1. Tree automata running on graphs.

Note that an infinite binary tree L' may be viewed as a "binary" structure $M = (S, R, L)$, where $S = \{0,1\}^*$, $R = R_0 \cup R_1$ with $R_0 = \{(s, s0) : s \in S\}$ and $R_1 = \{(s, s1) : s \in S\}$, and $L = L'$. We could alternatively write $M = (S, R_0, R_1, L)$.

We can also define a notion of a tree automaton running on certain appropriately labeled binary, directed graphs that are not binary trees. Such graphs, if accepted, are witnesses to the nonemptiness of tree automata. We make the following definitions.

A *binary structure* $M = (S, R_0, R_1, L)$ consists of a state set S and labeling L as before, plus a transition relation $R_0 \cup R_1$ decomposed into two partial functions: $R_0 : S \longrightarrow S$, where $R_0(s)$, when defined, specifies the 0-successor of s, and $R_1 : S \longrightarrow S$, where $R_1(s)$, when defined, specifies the 1-successor of s. We say that M is a *full binary structure* iff R_0 and R_1 are total.

A *run* of automaton $\mathcal{A}$ on binary structure $M = (S, R_0, R_1, L)$, if it exists, is a mapping $\rho : S \to Q$ such that for all $s \in S$, $(\rho(R_0(s)), \rho(R_1(s))) \in \delta(\rho(s), L(s))$, and $\rho(s_0) = q_0$. Intuitively, a run is a labeling of M with states of $\mathcal{A}$ consistent with the local structure of $\mathcal{A}$'s transition diagram. It will turn out that if an automaton accepts some binary tree, there does exist some finite binary graph on which there is a run that is *accepting*: all of the paths through the graph define state sequences of the automaton meeting its acceptance condition.

3.2. The transition diagram of a tree automaton.

The transition diagram of $\mathcal{A}$ can be viewed as an AND/OR-graph, where the set Q of states of $\mathcal{A}$ comprises the set of OR-nodes, while the AND-nodes define the allowable moves of the automaton. Intuitively, OR-nodes indicate that a nondeterministic choice has to be made (depending on the input label), while the AND-nodes force the automaton along all directions. For example, suppose that for automaton $\mathcal{A}$, $\delta(s, a) = \{(t_1, u_1), \ldots, (t_m, u_m)\}$ and $\delta(s, b) = \{(v_1, w_1), \ldots, (v_n, w_n)\}$; then the transition diagram contains the portion shown in Figure 3.1.

[2]We are assuming that each proposition symbol, such as GREEN_i, of formula Φ is associated

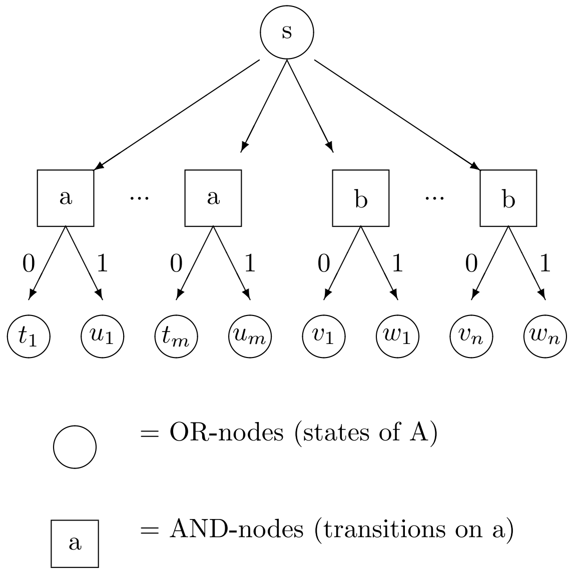


FIG. 3.1.

Formally, given a tree automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ with transition function $\delta : Q \times \Sigma \longrightarrow 2^{Q \times Q} : (q, a) \longmapsto \{(r_1, s_1), \ldots, (r_k, s_k)\}$, we may view it as defining a transition diagram T, which is an AND/OR-graph $(D, C, R_{DC}, R_{CD0}, R_{CD1}, L)$, where

$D = Q$ is the set of OR-nodes;

$C = \bigcup_{q \in D} \bigcup_{a \in \Sigma} \{((q, a), (r, s)) : (r, s) \in \delta(q, a)\}$ is the set of AND-nodes. Each AND-node corresponds to a transition. If $\delta(q, a) = \{(r_1, s_1), \ldots, (r_k, s_k)\}$, then the corresponding AND-nodes are essentially the pairs $(r_1, s_1), \ldots, (r_k, s_k)$. However, since each transition is associated with a unique current state/input symbol pair (q, a), we formally define the corresponding AND-nodes to be pairs of pairs: $((q, a), (r_1, s_1)), \ldots, ((q, a), (r_k, s_k))$.

$R_{DC} \subseteq D \times C$ specifies the AND-node successors of each OR-node. For each $q \in D$ as above, $R_{DC}(q) = \bigcup_{a \in \Sigma} \{((q, a), (r, s)) : (r, s) \in \delta(q, a)\}$;[3]

$R_{CD0}, R_{CD1} : C \longrightarrow D$ are partial[4] functions giving the 0-successor and 1-successor states, respectively:

$$R_{CD0}(((q, a), (r, s))) = r \quad \text{and} \quad R_{CD1}(((q, a), (r, s))) = s;$$

L is a labeling of nodes. For an AND-node $L(((q, a), (r, s))) = \{a\}$, where $a \in \Sigma$. For an OR-node, the labeling assigns propositions associated with the acceptance condition Φ so that all OR-nodes in the set GREEN_i (respectively, RED_i) are labeled with the corresponding proposition GREEN_i (respectively, RED_i).

with exactly the set of states of the corresponding name, in this case GREEN_i.

[3]We identify a relation such as $R_{DC} \subseteq D \times C$ with the corresponding function $R'_{DC} : D \longrightarrow 2^C$ defined by $R'_{DC}(d) = \{c \in C : (d, c) \in R_{DC}\}$ for each $d \in D$.

[4]For classically defined tree automata, the functions R_{CD0}, R_{CD1} are total so that there is always a 0-successor and 1-successor automaton state. For technical reasons it is convenient to allow R_{CD0}, R_{CD1} to be partial.

Thus, we may write a tree automaton A in the form (T, d_0, Φ), where T is the diagram, d_0 is the start state, and Φ is an acceptance condition.

3.3. One symbol alphabets. For purposes of testing nonemptiness, without loss of generality, we can restrict our attention to tree automata over a single letter alphabet and, thereby, subsequently ignore the input alphabet. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Phi)$ be a tree automaton over input alphabet Σ. Let $\mathcal{A}' = (Q, \Sigma', \delta', q_0, \Phi)$ be the tree automaton over one letter input alphabet $\Sigma' = \{c\}$ obtained from $\mathcal{A}$ by, intuitively, taking the same transition diagram but now making all transitions on symbol c. Formally, $\mathcal{A}'$ is identical to $\mathcal{A}$ except that the input alphabet is Σ' and the transition function δ' is defined by $\delta'(q, c) = \bigcup_{a \in \Sigma} \delta(q, a)$.

Observation 3.1. The set accepted by $\mathcal{A}$ is nonempty iff the set accepted by $\mathcal{A}'$ is nonempty.

Henceforth, we shall therefore assume that we are dealing with tree automata over a one symbol alphabet.

3.4. Generation and containment. It is helpful to reformulate the notion of run to take advantage of the AND/OR-graph organization of the transition diagram of an automaton. Intuitively, there is a run of diagram T on structure M provided M is "generated" from T by unwinding T so that each state of M is a copy of an OR-node of T.

Formally, we say that a binary structure $M = (S, R_0, R_1, L)$ is *generated by* a transition diagram $T = (D, C, R_{DC}, R_{CD0}, R_{CD1}, L_T)$ (starting at $s_0 \in S$ and $d_0 \in D$) iff $\exists$ is a total function $h : S \longrightarrow D$ such that for all $s \in S$

```
    if $s$ has any successors in $M$, then
        $\exists c \in R_{DC}(h(s))$
            for all $i \in \{0, 1\}$  $R_{CDi}(c)$ is defined iff $R_i(s)$ is defined and
                                      $R_{CDi}(c) = h(R_i(s))$ when both are defined
    and $L(s) = L_T(h(s))$
    (such that $h(s_0) = d_0$).
```

We say that a binary structure $M = (S, R_0, R_1, L)$ is *contained in* transition diagram T (starting at $s_0 \in M$ and $q_0 \in T$) provided M is generated by T (starting at $s_0 \in M$ and $q_0 \in T$), where the generation function h is the natural injection $h : S \longrightarrow D : s \in S \longmapsto s \in D$, so that all states of M are OR-nodes of T.

Note that if M is a structure generated by (respectively, contained in) T, there is an associated AND/OR-graph H generated by (respectively, contained in) T obtained from M by inserting between each state and its successors in M a copy of the AND-node that determines the successors of state via the generation function for M. We write $H = ao(M)$. Conversely, if H is an AND/OR-graph generated by (respectively, contained in) T, there is an associated structure M generated by (respectively, contained in) T obtained from H by eliding AND-nodes. Here we write $M = o(H)$.

3.5. Linear size model theorems. The following theorem (cf. [6]) is the basis of our method of testing nonemptiness of pairs automata. It shows that there is a small binary structure contained in the transition diagram and accepted by the automaton.

THEOREM 3.2 (linear size model theorem). *Let $\mathcal{A}$ be a tree automaton over a one symbol alphabet with pairs acceptance condition $\Phi = \bigvee_{i \in [1:k]} (\overset{\infty}{F} Q_i \wedge \overset{\infty}{G} P_i)$. Then automaton $\mathcal{A}$ accepts some tree T iff $\mathcal{A}$ accepts some binary model M of size linear in the size of $\mathcal{A}$, which is a structure contained in the transition diagram of $\mathcal{A}$.*

Proof. ($\Rightarrow$) For Φ a pairs condition, by the Hossley–Rackoff [18] finite model theorem, if $\mathcal{A}$ accepts some tree M_0, then it accepts some finite binary model M_1 starting at some state $s_0 \in M_1$. Thus, $M_1, s_0 \models A\Phi$ and M_1 is a structure generated by $\mathcal{A}$.

Given any such finite structure M_1 of $A\Phi$ generated by $\mathcal{A}$, we can obtain a finite structure M contained in $\mathcal{A}$ as follows. Let h be the generation function. If two distinct nodes s and t of M_1 have the same labeling with states of $\mathcal{A}$, i.e., $h(s) = h(t)$, then we can eliminate one of them as follows. Attempt to delete s by redirecting all its predecessors u to have t as a successor instead. More precisely, delete all edges of the form (u, s) and replace them by edges of the form (u, t). If the resulting structure, call it M^t, is a model of $A\Phi$, we have reduced the number of "duplicates," such as s and t, by one. If not, try replacing t by s instead. If M^s, the resulting structure is a model of $A\Phi$ and we are done.

However, if both of these replacements fail to yield a model of $A\Phi$, each must introduce a (not necessarily simple) bad cycle where the acceptance condition Φ fails. In M^t the bad cycle is of the form (where u is a predecessor of s in M_1) $u \rightarrow t \rightarrow \cdots \rightarrow u$, where except for the first transition (u, t) the suffix path from t to u is in the original M_1. In M^s the bad cycle is of the form (where v is a predecessor of t in M_1) $v \rightarrow s \rightarrow \cdots \rightarrow v$, where except for the first transition (v, s) the suffix path from s to v is in the original M_1.

However, these two suffix paths in M_1 together with the edges (u, s) and (v, t) in M_1 form a bad cycle in M_1: $u \rightarrow s \rightarrow \cdots \rightarrow v \rightarrow t \rightarrow \cdots \rightarrow u$. This contradicts that M_1 was a model of $A\Phi$.

By repeatedly eliminating duplicates in this way we eventually get the desired model M contained in $\mathcal{A}$.

($\Leftarrow$) Any model M contained in $\mathcal{A}$ such that $M, s_0 \models A\Phi$ can plainly be unwound into a tree that is accepted by $\mathcal{A}$. $\square$

A helpful generalization follows.

THEOREM 3.3 (generalized linear sized model theorem). *Let $s_0, s_1, \ldots, s_k$ ($k \geq 0$) be distinct elements of T and formula $g = P \wedge A(\Phi \vee FR)$, where $\Phi = \bigvee_{i \in [1:k]} (\overset{\infty}{F} Q_i \wedge \overset{\infty}{G} P_i)$ is a pairs acceptance condition and P, R are atomic propositions. Then,*

$\exists M'$ generated by T such that $M', s_0, s_1, \ldots, s_k \models g$

iff

$\exists M$ contained in T such that $M, s_0, s_1, \ldots, s_k \models g$.

Proof. ($\Rightarrow$) Assume the existence of M'. We first show that this implies the existence of M'' generated by T such that

$$M'', s_0, s_1, \ldots, s_k \models P \wedge A(\Phi \vee FR) \text{and for all} \quad s \in M'',\ M'', s \models A(\Phi \vee FR)$$

.

Intuitively, M'' is obtained by deleting spurious nodes from M', retaining just those nodes which are in the "cone" from some s_i until some R-node, if any, is encountered. Technically, let $S = \{t \in M'$: there exists a path from some s_i to t in M', all of whose nodes except (possibly) t itself satisfy $\neg R\}$. Let $S' = M' \setminus S$.

Thus, S' consists of all nodes that can be reached only from any s_i by going through some R-node. Now, delete all successor arcs of R-nodes, and then delete all nodes no longer reachable from some s_i. Note that the deleted nodes are exactly S'. Call the resulting substructure of M' so obtained M''. Note that the nodes retained in M'' are exactly S and that M'' is still generated by T.

Now observe that $A(\Phi \vee FR) \equiv A(A(\Phi \vee FR)\, U_w\, R)$. Therefore, for all $t \in S$, $M', t \models A(\Phi \vee FR)$. Hence, for all $t \in S$, $M'', t \models A(\Phi \vee FR)$, as the only nodes deleted in M'' are the ones which are reached from any s_i by going through some R-node. Thus, $M'', s_0, s_1, \ldots, s_k \models A(\Phi \vee FR) \wedge P$, and for all $s \in M''$, $M'', s \models A(\Phi \vee FR)$.

Any fullpath starting at any node in M'' either

(i) is infinite and satisfies Φ, or

(ii) is finite, terminating in an R-node.

We can now argue, just as in the linear size model theorem, that duplicates can be eliminated. If u and v are duplicates, we can attempt to replace u by v. The only problem is that it may introduce a "bad" cycle satisfying $\neg\Phi$. In that case, it must be possible to replace v by u, for otherwise there would be a bad cycle, i.e., an infinite fullpath satisfying $\neg\Phi$ in the original M''. Continuing in this fashion, we get a structure M with no duplicates. It is isomorphic to a structure contained in T with the desired properties, i.e., $M, s_0, s_1, \ldots, s_k \models P \wedge A(\Phi \vee FR)$.

($\Leftarrow$) By definition, any model M contained in T is also generated by T. ◻

3.6. Pseudomodels. The linear size model theorem also suggests the following.

DEFINITION 3.4. *Let $T = (D, C, R_{DC}, R_{CD0}, R_{CD1}, L_T)$ be the transition diagram of a tree automaton $\mathcal{A}$, let s be a state of $\mathcal{A}$, and let p be a formula.*

- We say that diagram T at state s is a *pseudomodel* of p (or that p is *satisfiable* in T at s) and write $T, s \,\|\!-_{con}\, p$ iff there exists a structure M contained in T such that $M, s \models p$. We also write $p^{T,con}$ for $\{s \in T : T, s \|\!-_{con} p\}$.
- We say p is *generable* in T at s and write $T, s \|\!-_{gen} p$ iff there exists a structure M generated by T such that $M, s \models p$. We also write $p^{T,gen}$ for $\{s \in T : T, s \|\!-_{gen} p\}$.
- We say p is *true* (or *modeled*) in T at s, considered simply as a structure with state set $D \cup C$ and transition relation $R_{CD} \cup R_{DC0} \cup R_{DC1}$ iff $T, s \models p$. We also write $p^{T,\mathrm{mod}}$ for $\{s \in T : T, s \models p\}$.

Remark. For a proposition symbol P we have $P^{T,\mathrm{mod}} = P^{T,con} = P^{T,gen}$.

Our overall approach to testing nonemptiness can now be summarized in the following rephrasing of the linear size model theorem.

THEOREM 3.5. *Automaton $\mathcal{A}$ with diagram T and pairs acceptance condition Φ is nonempty iff $q_0 \in A\Phi^{T,con}$.*

We will check nonemptiness by calculating $A\Phi^{T,con}$ by a process of "pseudomodel checking" (cf. [4]*).*

4. Complexity of pairs tree automata. In this section we prove that nonemptiness of pairs tree automata can be tested in time polynomial in the number of states and exponential in the number of pairs, even though, as we show, the problem is NP-complete in general.

THEOREM 4.1. *Nonemptiness of a pairs automaton $\mathcal{A}$ having m states and n pairs can be tested in deterministic time $(mn)^{O(n)}$.*

Proof. Let $\mathcal{A}$ be a tree automaton with transition diagram T of size m states and pairs acceptance condition $\Phi_\Gamma = \bigvee_{\gamma \in \Gamma} (GFQ_\gamma \wedge FGP_\gamma)$, where Γ is the index set $[1:n]$ of pairs. Here, Q_γ and P_γ stand for GREEN$_\gamma$ and $\neg$RED$_\gamma$, respectively. When Γ is understood from context we can drop it and write just Φ for Φ_Γ. We also let $\Phi_{-\gamma}$ denote $\Phi_{\Gamma \setminus \{\gamma\}}$.

The basic idea of the algorithm is to inductively compute the set of states in T at which $A\Phi$ is satisfiable, viz., $A\Phi^{T,con} = \{s \in T : T, s \|\!-_{con} A\Phi\}$. For this purpose, we use the fixpoint characterization $\mu Y.\tau(Y)$ of $A\Phi$ from Lemma 4.2 below and evaluate it iteratively using the Tarski–Knaster theorem specialized to pseudomodel checking in Lemma 4.3 below. To compute each $Y^{i+1} = \tau(Y^i)$ we evaluate the body $\tau(Y)$ compositionally,[5] using Lemmas 4.2–4.9 as appropriate. This in turn entails the recursive calculation of (essentially) $A\Phi_{-\gamma}$ with one fewer pair. The recursion terminates when $\Phi_{-\gamma} \equiv false$ has 0 pairs, in which case Lemma 4.4 is used instead of Lemma 4.3. For technical reasons, we actually pseudomodel check a modified pairs condition of the form $A(\Phi \vee FR)$, which specializes to the ordinary pairs conditions when $R \equiv false$.

If each OR-node in T had a unique successor, then the pseudomodel checking of $A(\Phi \vee FR)$ would simply amount to model checking in the Mu-calculus [12]. But in general, each OR-node has more than one successor. Therefore, our algorithm must simultaneously exhaust the search space of all structures contained in T and check if one of these structures is a model of the pairs condition. Lemmas 4.2–4.9 below permit both steps to be done together, thereby performing the desired pseudomodel checking. The proofs of the lemmas are given in Appendix A.

LEMMA 4.2 (fixpoint characterization for modified pairs condition). *Let* $\Phi = \vee_\gamma(\overset{\infty}{G}P_\gamma \wedge \overset{\infty}{F}Q_\gamma)$ *denote the pairs condition where* $\Gamma = [1:n]$*; let R be propositional. Then* $A(\Phi \vee FR) \equiv \mu Y.\tau(Y)$, *where* $\tau(Y) = R \vee \vee_\gamma AX_sA(g_\gamma U_w R)$ *and* $g_\gamma = A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma)) \wedge (P_\gamma \vee Y) \wedge AX_s true$.

LEMMA 4.3 (pseudomodel checking via Tarski–Knaster approximation[6]). *Set* $\mathsf{Y}^0 := false^{T,con}$*; set* $\mathsf{Y}^{i+1} := \tau(Y^i)^{T,con}$. *Then* $A(\Phi \vee FR)^{T,con} = \mu Y.\tau(Y)^{T,con} = \bigcup_{i \le |T|} \mathsf{Y}^i$.

LEMMA 4.4 (pseudomodel checking inevitability).

$$AFR^{T,con} = (\mu V.R \vee EX_sAX_sV)^{T,\mathrm{mod}}.$$

Observation 4.5 (pseudomodel checking of disjunctions).

$$(R \vee \bigvee_\gamma (AX_sA[g_\gamma U_w R]))^{T,con} = R^{T,con} \cup \bigcup_\gamma (AX_sA[g_\gamma U_w R])^{T,con}.$$

Observation 4.6 (pseudomodel checking of nexttime).

$$(AX_sA[g_\gamma U_w R])^{T,con} = \{s \in T : T, s \models EX_sAX_sW\}, \mathrm{where} \mathsf{W} = A[g_\gamma U_w R]^{T,con}.$$

DEFINITION 4.7. *Let U be a transition diagram for an automaton, and let Z be a set of OR-nodes of U. We define the AND/OR-graph denoted $U|\mathsf{Z}$, called U* restricted to Z, *to be the result of deleting from U all OR-nodes not in Z and incident arcs, and then deleting all AND-nodes, which do not have all successors in Z, along with all incident arcs. By a convenient abuse of notation, we shall write Z for $U|\mathsf{Z}$ when it is clear that an AND/OR-subgraph of U is intended. In particular, if f is a temporal formula, we write $f^{\mathsf{Z},con}$ for $f^{U|\mathsf{Z},con}$.*

LEMMA 4.8 (pseudomodel checking of weak until). *Set* $\mathsf{Z}^0 := true^{T,con}$*; set* $\mathsf{Z}^{i+1} := ((g_\gamma \wedge AX_sZ^i) \vee R)^{\mathsf{Z}^i,con}$. *Then for some least k,* $\mathsf{Z}^k = \mathsf{Z}^{k+1}$ *is the fixpoint of the descending chain* $\mathsf{Z}^i \supseteq \mathsf{Z}^{i+1}$ *and* $A[g_\gamma U_w R]^{T,con} = \mathsf{Z}^k$.

[5]That is, by induction on formula structure.

[6]It is to be understood below that "set $\mathsf{Y}^{i+1} := \tau(Y^i)^{T,con}$," for example, means to assign to the set Y^{i+1} exactly the set of states in $\tau(Y^i)^{T,con}$ and label the resulting states in the set Y^{i+1} with the symbol Y^{i+1}

LEMMA 4.9 (pseudomodel checking of conjunctions in g_γ). $g_\gamma^{T,con} = A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma))^{T,con} \cap (P_\gamma \vee Y)^{T,con} \cap AX_s true^{T,con}$.

Finally, we analyze the complexity as follows. Let $Com(T, f)$ stand for the complexity of computing $\{s|T, s \,\|\!- f\}$. For size of the transition diagram $|T| = m$, and number of pairs $|\Gamma| = n$, let $C(m, n)$ denote $Com(T, A(\Phi_\Gamma \ \vee \ R))$.

$$\begin{aligned} Com(T, A(\Phi_\Gamma \ \vee \ FR)) &\leq O(m) \cdot Com(T, Y^i), \\ Com(T, Y^i) &\leq n \cdot Com(T, R \vee AX_s A(g_\gamma U_w R)), \\ Com(T, R \vee AX_s A(g_\gamma U_w R)) &\leq O(m) + O(m) \cdot Com(Z^i, g_\gamma \ \wedge \ AX_s Z^i), \\ Com(Z^i, g_\gamma \ \wedge \ AX_s Z^i) &\leq O(m) + Com(U, A(\Phi_{\Gamma \setminus \{\gamma\}} \ \vee FR')), \end{aligned}$$

where $|Z^i|, |U| \ \leq \ |T|$. Thus, for some constant $k > 0$ we have

$$\begin{aligned} C(m, n) &\leq kmn(km + km(km + C(m, n-1))) \\ &= kmn(km + k^2m^2 + kmC(m, n-1)) \\ &= k^2m^2n + k^3m^3n + k^2m^2nC(m, n-1) \\ &\leq 2k^3m^3n + k^2m^2nC(m, n-1). \end{aligned}$$

The above accounts for the cost for 1 or more pairs ($n > 0$). Lemma 4.4 implies $n = 0$ pairs can be handled in $O(m)$ time. Hence, for some sufficiently large constant $c \geq 2k^3$ we have

$$\begin{aligned} &C(m, n) \leq cm^3n + cm^2nC(m, n-1), \\ &C(m, 0) \leq cm. \end{aligned}$$

We must thus solve a recurrence of the form

$$\begin{aligned} &C(m, n) = x + yC(m, n-1), \\ &C(m, 0) = z, \end{aligned}$$

which is readily expanded to show that its solution is $x(y^n - 1) + y^n z \leq y^n(x + z)$. It follows that $C(m, n)$ is at most

$$\begin{aligned} &= (cm + cm^3n)(cm^2n)^n \\ &\leq (dm^3n)(dm^2n)^n \text{ for some constant } d > 2c \\ &= d^{n+1}m^{2n+3}n^{n+1} \\ &\leq n^{n+1}m^{2n+3}n^{n+1} \text{ for all } n \geq d \\ &\leq (mn)^{2n+3} \\ &= (mn)^{O(n)} \text{ for all } n, m \geq 1. \end{aligned}$$

Note that m is the size of the transition diagram, which for an automaton on binary trees can be cubic in the number of states, m_0, i.e., $m = O(m_0^3)$. As a function of the number of states, however, we still get the same order of growth, i.e., $(mn)^{O(n)} = (m_0^3 n)^{O(n)} = (m_0 n)^{3 \cdot O(n)} = (m_0 n)^{O(n)}$. In general, for k-ary trees, the size of the transition diagram m can be $O(m_0^{k+1})$. Thus we get time $(m_0 n)^{(k+1)O(n)}$. This is still $(m_0 n)^{O(n)}$ for fixed k. Moreover, if k grows linearly with n, the bound is still $(m_0 n)^{O(n^2)}$, which is sufficient for obtaining the complexity bounds on logics of programs of section 6. $\square$

THEOREM 4.10. *Pairs tree automaton nonemptiness is NP-complete.*

Proof. Pairs automaton nonemptiness was shown to be in NP in [6]: a nondeterministic Turing machine can guess a linear size model M contained in the automaton diagram and verify that it satisfies the acceptance condition using the model checking algorithm for fairness of [11b] (cf. [41]).

To show NP-hardness we reduce from 3-SAT. Let f be a 3-CNF formula with m clauses $C_1, C_2, \ldots, C_m$ over n variables $v_1, \ldots, v_n$. We will reduce satisfiability of f to the nonemptiness problem of a pairs tree automaton $\mathcal{A}$ with number of states and pairs polynomial in $|f|$.

We will describe $\mathcal{A}$ in terms of its AND/OR-graph transition diagram as usual and pairs acceptance condition.[7] Since we are only concerned about the nonemptiness problem, $\mathcal{A}$ is assumed to have a one symbol alphabet. Corresponding to each clause C_i of f, $\mathcal{A}$ will have an OR-node of the same name, C_i. For each possible literal x, i.e., for each variable v_j and its negation $\neg v_j$, there is an AND-node of the same name, x. If clause C_i contains literal x, then there is an edge $C_i \to x$ in the diagram. If clause C_i contains the literal $\neg x$, then there is an edge $x \to C_i$ in the diagram. Here we identify $\neg\neg x$ with x. We also let S_0 be the start state OR-node with a single AND-node successor which has as its successors $C_1, \ldots, C_m$. Finally, for each literal x there is a pair of lights (GREEN_x, RED_x) such that GREEN_x colors AND-node x and RED_x colors AND-node $\neg x$. Let Φ be the corresponding pairs condition.

The basic idea is that a structure M contained in (the diagram of) $\mathcal{A}$ specifies a choice of literal for each clause. From these literals we can try to recover a satisfying truth assignment for f and vice versa. However, there may be conflicts with one clause using x, another $\neg x$. The following argument shows that there is a conflict-free choice of literals when there is an M satisfying $A\Phi$.

The following claims are equivalent:

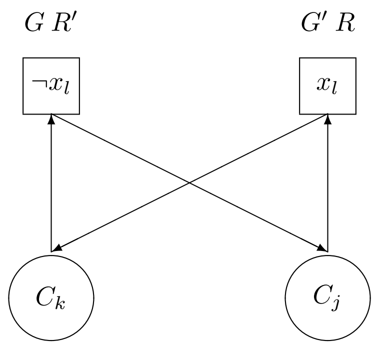
(i) $\mathcal{A}$ is nonempty.
(ii) There is a structure M contained in $\mathcal{A}$, accepted by $\mathcal{A}$ so that $M, S_0 \models A\Phi$.
(iii) There exists $H = ao(M)$, where M is contained in $\mathcal{A}$, such that in H, each C_j has an edge to some x_l and no other C_k has an edge to the negation of x_l.
(iv) f is satisfiable.

Claims (i) and (ii) are equivalent by the linear size model theorem. The equivalence of (iii) and (iv) is immediate from the construction of $\mathcal{A}$. The equivalence of (ii) and (iii) follows from a slightly sharper equivalence. Let $H = ao(M)$, where M is contained in $\mathcal{A}$. Then the following are equivalent:

(ii)′ $M, S_0 \models A\Phi$.
(iii)′ In H, each C_j has an edge to some x_l and no other C_k has an edge to the negation of x_l.

Condition (iii)′ implies that for every green light labeling a node in H, its corresponding red light does not occur in H. Hence, along every infinite path some green light appears infinitely often but its corresponding red light never appears, and Φ holds along the path. Condition (ii)′ follows. Now assume (ii)′. Then M is total as is H. It must be that (iii)′ holds of H. Otherwise, there is a C_j with an edge to x_l and a C_k with an edge to $\neg x_l$. Then we have the following subgraph in H.

[7] We will actually describe the diagram of an automaton on trees of arity at most m, where, moreover, the propositions (or "colors") defining acceptance label AND-nodes rather than OR-nodes. It is not difficult to show that it can be converted into an "equivalent, ordinary" automaton. Here "equivalent" means that the original automaton accepts iff the derived automaton accepts. Conversion to a binary tree automaton is effected by inserting "dummy" nodes in the transition diagram to reduce fan-in m to fan-in 2. This causes a linear blowup in size. The acceptance condition colors can be moved to the OR-nodes by having each derived node be of the form (OR-node, AND-node predecessor causing transition). This can cause a quadratic blowup in size.



Here $(G,\ R)$ and $(G',\ R')$ are the pairs corresponding to the literals x_l and $\neg x_l$, respectively. The arcs $x_l \to C_k$ and $\neg x_l \to C_j$ exist by definition of $\mathcal{A}$, considering that $\neg x_l$ is in C_k and x_l is in C_j. The above subgraph generates a path which has for all green lights flashing infinitely often its corresponding red light flashing infinitely often, too. But then there would be a "bad" path in M violating Φ, contrary to our assumption of (ii)$'$. The equivalence of (ii) and (iii) follows. This completes the reduction. □

5. Complexity of complemented pairs tree automata. In this section, we will show that there is also a deterministic algorithm to test nonemptiness of a complemented pairs tree automaton (cf. [33]) which runs in time $(mn)^{O(n)}$, where m is the number of states of the automaton and n is the number of pairs in its acceptance condition. Moreover, we will show that testing nonemptiness of such automata is co-NP-complete.

The key idea is that for a tree automaton $\mathcal{A}$ over a one symbol alphabet we can define its dual tree automaton $\tilde{\mathcal{A}}$ such that $\mathcal{A}$ is nonempty iff $\tilde{\mathcal{A}}$ is empty. The dual is essentially obtained by swapping AND-nodes with OR-nodes and complementing the acceptance condition. Since we view the transition diagram of a nondeterministic tree automaton as a bipartite AND/OR graph, the dual is also a bipartite AND/OR graph and hence can be viewed as a nondeterministic automaton. Moreover, the dual of a pairs automaton is a complemented pairs automaton and vice versa. Muller and Schupp also define a dual automaton in [24], but for a nondeterministic automaton their dual automaton is in general an alternating automaton. For the purpose of checking nonemptiness of automata over one symbol alphabets, the dual defined in this new way suffices.

For an automaton $\mathcal{A}$ and its dual $\tilde{\mathcal{A}}$, we are interested in showing that $\mathcal{A}$ is nonempty iff $\tilde{\mathcal{A}}$ is empty. In other words, we must show that

$$\exists \text{ Run } \rho \text{ of } \mathcal{A} \quad \text{for all paths in } \rho \quad \Phi_{\mathcal{A}} \tag{5.1}$$

$$\equiv \neg\ \exists \text{ Run } \rho \text{ of } \tilde{\mathcal{A}} \quad \text{for all paths in } \rho \quad \Phi_{\tilde{\mathcal{A}}}, \tag{5.2}$$

where $\Phi_{\mathcal{A}}$ is the acceptance condition of $\mathcal{A}$ and $\Phi_{\tilde{\mathcal{A}}} \equiv \neg\Phi_{\mathcal{A}}$ is the acceptance condition of $\tilde{\mathcal{A}}$. We can argue this informally as follows. By expanding (1) we get that

$$(1) \equiv \forall! d_0 \exists c_0 \forall d_1 \exists c_1 \forall d_2 \exists c_2 \ldots\ d_0 c_0 d_1 c_1 d_2 c_2 \ldots \models \Phi_{\mathcal{A}},$$

meaning that for the unique start OR-node d_0 of $\mathcal{A}$ there exists an AND-node successor c_0 such that for all OR-node successors d_1 of c_0 there exists an AND-node successor

c_1, etc., ad infinitum such that $d_0c_0d_1c_1d_2c_2\ldots \models \Phi_{\mathcal{A}}$. Similarly, by expanding (2), we get that

$$(2) \equiv \neg\forall! e_0 \exists! d_0 \forall c_0 \exists d_1 \forall c_1 \exists d_2 \forall c_2 \ldots\ e_0d_0c_0d_1c_1d_2c_2\ldots \models \neg\Phi_{\mathcal{A}},$$

where e_0 is the unique "dummy" start state of $\tilde{\mathcal{A}}$, all of whose successors are d_0. (This is explained in more detail below.) Applying the well-known quantifier negation law to the infinite string of quantifiers for (2), we get after driving the negation inside that

$$(2) \equiv \exists! e_0 \forall! d_0 \exists c_0 \forall d_1 \exists c_1 \forall d_2 \exists c2 \ldots\ e_0d_0c_0d_1c_1d_2c_2\ldots \models \Phi_{\mathcal{A}}.$$

Since the truth of $\Phi_{\mathcal{A}}$ is oblivious to the initial e_0 we should be able to conclude that (1) $\equiv$ (2) as desired. But the quantifier negation law for infinite strings of quantifiers is known to contradict the axiom of choice in general. Therefore to formally prove the above, we show that the set of paths $\Phi_{\mathcal{A}}$ and its complement are "nice" in the sense that they are finite automaton definable (and hence Borel) and conclude the desired equivalence using Martin's theorem on determinacy of infinite Borel games (cf. [22]).[8]

The above notions are formalized in what follows.

Definition 5.1. *Let λ denote the empty sequence. For an infinite sequence $x_0x_1\ldots$, let x^i be the suffix $x_ix_{i+1}\ldots$. In particular, tail(x) stands for x^1. If $x = x_0x_1\ldots$ and $y = y_0y_1\ldots$ are two infinite sequences, then let $x\hat{\ }y$ ("zip") denote the sequence $x_0y_0x_1y_1\ldots$. We let $\lambda^\omega = \lambda^n = \lambda$ and $x\lambda = x$. Then we can define zip of two finite/infinite sequences by appending finite sequences with λ^ω.*

We are given an automaton $\mathcal{A}$ over a one symbol alphabet with acceptance condition $\Phi_{\mathcal{A}}$. Without loss of generality we may stipulate that each node of its transition diagram has exactly two successors.[9] Thus we may assume $\mathcal{A}$ is of the form (D, C, R, d_0, L) with OR-node set D, AND-node set C, start OR-node d_0, labeling L, and transition relation $R = R_{DC} \cup R_{CD0} \cup R_{CD1}$. Since each node has exactly two successors we may view R as a function so that $R(d, i) = c$ indicates that c is the AND-node successor of OR-node d of index $i \in \{0, 1\}$ and $R(c, j) = d$ indicates that d is the OR-node successor of AND-node c in direction $j \in \{0, 1\}$. We also assume that the labeling L assigning "colors" associated with $\Phi_{\mathcal{A}}$ to each node has been extended to AND-nodes so that each AND-node is colored exactly as is its unique OR-node parent.[10]

We shall define the dual automaton $\tilde{\mathcal{A}}$ essentially by swapping OR-nodes and AND-nodes in $\mathcal{A}$. However, we shall have to do a bit more to ensure that the result meets the technical definition of being an automaton as we have defined them to be.

Definition 5.2. *We first define $\hat{\mathcal{A}}$ with diagram of the form $(\hat{D}, \hat{C}, \hat{R}, \hat{d}_0, \hat{L})$, where $\hat{D} = C \cup \{e_0\}$, $\hat{C} = D$, $\hat{R}(e_0, 0) = \hat{R}(e_0, 1) = d_0$ with $\hat{R}(b, i) = R(b, i)$ for all $b \in C \cup D$ and $i \in [0, 1]$, and $\hat{L}$ is the same as L but it includes e_0 in its domain, assigning it the empty set of color propositions. We see that $\hat{\mathcal{A}}$ is the result of swapping OR-nodes and AND-nodes from $\mathcal{A}$ and adding a new dummy start state e_0 both of whose successors are d_0, start state of $\mathcal{A}$. $\hat{\mathcal{A}}$ is almost what we want for the dual except that its AND-nodes, which were OR-nodes of $\mathcal{A}$, are not guaranteed to have unique predecessors. Technically, this violates the definition of an automaton.*

[8]Since $\Phi_{\mathcal{A}}$ is finite automaton definable, a weaker result of Büchi–Landweber [3], showing determinacy of such games, will do.

[9]The argument generalizes in a straightforward way to k successors.

[10]Thus, if x is a path through $\mathcal{A}$, we have $x \models \Phi_{\mathcal{A}}$ iff $x|D \models \Phi_{\mathcal{A}}$ iff $x|C \models \Phi_{\mathcal{A}}$.

Therefore, to get $\tilde{\mathcal{A}}$ from $\hat{\mathcal{A}}$ we must create duplicates of the AND-nodes in $\hat{C}$. In other words, we locally unravel $\hat{\mathcal{A}}$: if $\hat{d} \to \hat{c}$ and $\hat{c} \to \hat{d}'$ appear in $\hat{\mathcal{A}}$, then $\hat{d} \to (\hat{d}, \hat{c})$ and $(\hat{d}, \hat{c}) \to \hat{d}'$ appear in $\tilde{\mathcal{A}}$ with the AND-node $(\hat{d}, \hat{c})$ of $\tilde{\mathcal{A}}$ labeled exactly as is AND-node $\hat{c}$ of $\hat{ca}$. The result is the underlying diagram of $\tilde{\mathcal{A}}$. Formally, we let $\tilde{A}$ have diagram $(\tilde{D}, \tilde{C}, \tilde{R}, \tilde{d_0}, \tilde{L})$. Here $\tilde{D} = \hat{D}$ and $\tilde{d_0} = \hat{d_0}$. We define $\tilde{C} \subseteq \hat{D} \times \hat{C}$ such that if AND-node $\hat{c}$ has OR-node predecessors $\hat{d_1}, \dots, \hat{d_k}$ in $\hat{\mathcal{A}}$, then AND-nodes $(\hat{d_1}, \hat{c}), \dots, (\hat{d_k}, \hat{c})$ are in $\tilde{C}$. If $\hat{R}(\hat{d}, i) = \hat{c}$, then $\tilde{R}(\hat{d}, i) = (\hat{d}, \hat{c})$ for $i \in \{0,1\}$. If $\hat{R}(\hat{c}, j) = \hat{d}$, then $\tilde{R}((\hat{d_l}, \hat{c}), j) = \hat{d}$ for $j \in \{0,1\}$ and where $\hat{d_l}$ is any predecessor of $\hat{c}$ in $\hat{\mathcal{A}}$. Finally, $\tilde{L}(\hat{d}) = \hat{L}(\hat{d})$ and $\tilde{L}((\hat{d_l}, \hat{c})) = \hat{L}(\hat{c})$. This transformation of $\hat{\mathcal{A}}$ into $\tilde{\mathcal{A}}$ can cause at worst a quadratic blowup in size.

DEFINITION 5.3. *Given an infinite string $z \in \{0,1\}^\omega$ and automaton $\mathcal{A}$ with diagram (D, C, R, d_0, L), we can associate with z, by starting at d_0 and following the arc labels spelling out z, a unique infinite path through the diagram of $\mathcal{A}$ and vice versa.*

Formally, we define $path_{\mathcal{A}} : \{0,1\}^\omega \to (DC)^\omega$ such that for $z = x\hat{\ }y$, $path_{\mathcal{A}}(z) = s\hat{\ }t$, where $x = x_0x_1\dots$, $y = y_0y_1\dots$, $s = s_0s_1\dots$, $t = t_0t_1\dots$, $s_0 = d_0$ is the start node in D, $t_n = R(s_n, x_n)$, and $s_{n+1} = R(t_n, y_n)$.

Observation 5.4. $\tilde{\pi} = path_{\tilde{\mathcal{A}}}(z)$ is the same as $\hat{\pi} = path_{\hat{\mathcal{A}}}(z)$, except that each AND-node $\tilde{c}$ along $\tilde{\pi}$ is a duplicate of the corresponding AND-node $\hat{c}$ along $\hat{\pi}$ of the form $(\hat{d}, \hat{c})$, where $\hat{d}$ is the predecessor of $\hat{c}$ along $\hat{\pi}$. Therefore, $\tilde{\pi} \models \Phi_{\mathcal{A}}$ iff $\hat{\pi} \models \Phi_{\mathcal{A}}$.

Observation 5.5. $path_{\tilde{\mathcal{A}}}(x\hat{\ }y) = e_0 \cdot path_{\mathcal{A}}(y\hat{\ }tail(x))$.

This follows because, all successors of e_0 are d_0, and hence x_0 is redundant.

DEFINITION 5.6. *We now define a* two player infinite game $\mathcal{G}$ *associated with* $\mathcal{A}$. *There are two players* I *and* II. I *goes first and picks $x_0 \in \{0,1\}$, then* II *picks some $y_0 \in \{0,1\}$, then* I *picks some $x_1 \in \{0,1\}$, and so on alternatively. The resulting infinite string $z = x_0y_0x_1y_1\dots$ is a* play *of the game. Player* I *wins this particular play z if z is in the* winning set Γ, *which is defined by letting $\Gamma = \{x : x \in \{0,1\}^\omega$ $path_G(x)$ satisfies $\Phi_{\mathcal{A}}\}$; otherwise* II *wins the play.*

A strategy *for* I *is a function f_{I} $\{0,1\}^* \to \{0,1\}$, and a strategy for* II *is a function $f_{\mathrm{II}} : \{0,1\}^+ \to \{0,1\}$. The family of all such strategy function will be denoted Strat* I *and Strat* II, *respectively, with typical elements denoted f_{I} and f_{II}, respectively.*

Any function f $\{0,1\}^* \to \Sigma$ induces a map $\overline{f} : \{0,1\}^\omega \to \Sigma^\omega$. If $z = z_0z_1\dots$ and $x = x_0x_1\dots$, $\overline{f}(x) = z$ such that, for all n, $z_n = f(x_0x_1\dots x_{n-1})$ and $z_0 = f(\lambda)$. Similarly, a function $f : \{0,1\}^+ \to \Sigma$ induces a map $\overline{f} : \{0,1\}^\omega \to \Sigma^\omega$ such that $z_n = f(x_0x_1\dots x_n)$.

We say that Player I *follows strategy f_{I} if it chooses $f_{\mathrm{I}}(y_0y_1\dots y_{n-1})$ when* II *has chosen $y_0y_1\dots y_{n-1}$. Similarly for Player* II. *If* II *builds up y during a play, and* I *follows f_{I}, then the resulting play is $\overline{f_{\mathrm{I}}}(y)\hat{\ }y$. Similarly, if I builds up x, and* II *follows f_{II}, the resulting play is $x\hat{\ }\overline{f_{\mathrm{II}}}(x)$.*

We say that f_{I} is a winning strategy *for* I *if for all $y \in \{0,1\}^\omega$ $\overline{f_{\mathrm{I}}}(y)\hat{\ }y \in \Gamma$. Similarly, f_{II} is a winning strategy for* II *if for all $y \in \{0,1\}^\omega$ $y\hat{\ }\overline{f_{\mathrm{II}}}(y) \notin \Gamma$.*

Remark. When the transition diagram is presented as in this section, the definition of a run ρ is formulated as

$\rho(\lambda) = d_0$

for all $y \in \{0,1\}^*$ $\exists i \in \{0,1\}$ $\exists c \in C$ $c = R(\rho(y), i)$ and for all $j \in \{0,1\}$ $\rho(yj) = R(b, j)$.

The first condition asserts that ρ annotates the root node with the start state. The second condition asserts that each node y of the tree has its successors $y0, y1$ annotated in a manner consistent with the diagram because there is an AND-node

c from OR-node $\rho(y)$ to OR-nodes $\rho(y0), \rho(y1)$ in the diagram. Note also that the condition that "along all paths of a run ρ, $\Phi_{\mathcal{A}}$ holds," is now formulated as "for all $y \in \{0,1\}^\omega$ $\overline{\rho}(y)$ satisfies $\Phi_{\mathcal{A}}$," since y here is spelling out a path through the tree in terms of edge labels.

LEMMA 5.7. *Let $\mathcal{A}$ be an automaton with acceptance condition $\Phi_{\mathcal{A}}$. Then,*

$\exists$ run ρ of $\mathcal{A}$ for all $y \in \{0,1\}^\omega$ $\overline{\rho}(y)$ satisfies $\Phi_{\mathcal{A}}$

iff

$\exists f_I$ for all $y \in \{0,1\}^\omega$ $path_{\mathcal{A}}(\overline{f_I}(y)\hat{\ }y)$ satisfies $\Phi_{\mathcal{A}}$.

This lemma says that an accepting run defines a corresponding winning strategy and vice versa. Intuitively, given a run as shown in Figure 5.1 (a) we can extend it to indicate the intermediate AND-nodes and edges, indexed by 0 or 1, between nodes, as shown in Figure 5.1 (b). These edges indicating that c_n is the x_n-successor of d_n, that d_{n+1} is the y_{n+1}-successor of c_n, and so forth constitute an edge-labeled tree defining the strategy, as shown in Figure 5.1 (c). Conversely, given the strategy we can view it as an edge-labeled tree, and given the start node d_0, infer the corresponding run. This argument is formalized below.

Proof. Given ρ, let $f_{\mathrm{I}}(y) = \min\{i : \exists b\ b = R(\rho(y), i)$ and for all j $\rho(yj) = R(b,j)\}$ so that f_{I} picks, for the sake of definiteness, the AND-node successor of least index. The above definition is well defined by the definition of a run.

We show that $\overline{\rho}(y)$ is exactly s, where $s\hat{\ }t = path_{\mathcal{A}}(\overline{f_I}(y)\hat{\ }x)$. By the definition of $path_{\mathcal{A}}$, $s_{n+1} = R(t_n, y_n)$ and $t_n = R(s_n, x_n)$, where $x = \overline{f_I}(y)$. We show by induction that $\rho(y_0y_1\ldots y_n) = s_{n+1}$.

The base case is trivial because $\rho(\lambda) = s_0$. By induction hypothesis, $\rho(y_0y_1\ldots y_{n-1}) = s_n$. By definition of f_{I}, $t_n = g(\rho(y_0y_1\ldots y_{n-1}), x_n) = b$ and $\rho(y_0y_1\ldots y_n) = g(b, y_n) = g(t_n, y_n) = s_{n+1}$.

For the other direction, given f_I we define ρ:

$\rho(\lambda) = s_0$,

$\rho(yj) = R(R(\rho(y), f_{\mathrm{I}}(y)), j)$.

It follows from this definition of ρ and the definition of $path_{\mathcal{A}}$ that, if $s\hat{\ }t = path_{\mathcal{A}}(\overline{f_I}(y)\hat{\ }y)$, then $s = \overline{\rho}(y)$.

In the definition of $\tilde{\mathcal{A}}$, we required that the proposition labels be the same on the OR-nodes and their AND-node successors. For the dual automaton, it follows that the labels on the OR-nodes are the same as on their predecessor AND-nodes. Thus, $path_{\mathcal{A}}(\overline{f_I}(y)\hat{\ }x)$ satisfies $\Phi_{\mathcal{A}}$ iff $\overline{\rho}(y)$ satisfies $\Phi_{\mathcal{A}}$. □

LEMMA 5.8. *Let $\tilde{\mathcal{A}}$ be the dual automaton which has acceptance condition $\neg\Phi_{\mathcal{A}}$. Then,*

$\exists$ run ρ of $\tilde{\mathcal{A}}$ for all $y \in \{0,1\}^\omega$ $\overline{\rho}(y)$ satisfies $\neg\Phi_{\mathcal{A}}$

iff

$\exists f_{\mathrm{II}}$ for all $x \in \{0,1\}^\omega$ $path_{\tilde{\mathcal{A}}}(x\hat{\ }\overline{f_{\mathrm{II}}}(x))$ satisfies $\neg\Phi_{\mathcal{A}}$.

Proof. We argue as follows:

$\exists f_{\mathrm{II}}$ for all $x \in \{0,1\}^\omega$ $path_{\mathcal{A}}(x\hat{\ }\overline{f_{\mathrm{II}}}(x))$ satisfies $\neg\Phi_{\mathcal{A}}$

iff (since $\Phi_{\mathcal{A}}$ is oblivious to finite prefixes being altered)

$\exists f_{\mathrm{II}}$ for all $x \in \{0,1\}^\omega$ $e_0 \cdot path_{\mathcal{A}}(x\hat{\ }\overline{f_{\mathrm{II}}}(x))$ satisfies $\neg\Phi_{\mathcal{A}}$

iff (taking f_{I} to be the same as f_{II}, except $f_{\mathrm{I}}(\lambda)$ is defined arbitrarily)

$\exists f_{\mathrm{I}}$ for all $x \in \{0,1\}^\omega$ $e_0 \cdot path_{\mathcal{A}}(x\hat{\ }tail(\overline{f_{\mathrm{I}}}(x)))$ satisfies $\neg\Phi_{\mathcal{A}}$

iff (by Observation 5.4)

$\exists f_{\mathrm{I}}$ for all $x \in \{0,1\}^\omega$ $path_{\tilde{\mathcal{A}}}(\overline{f_{\mathrm{I}}}(x)\hat{\ }x)$ satisfies $\neg\Phi_{\mathcal{A}}$

iff (by Observation 5.5)

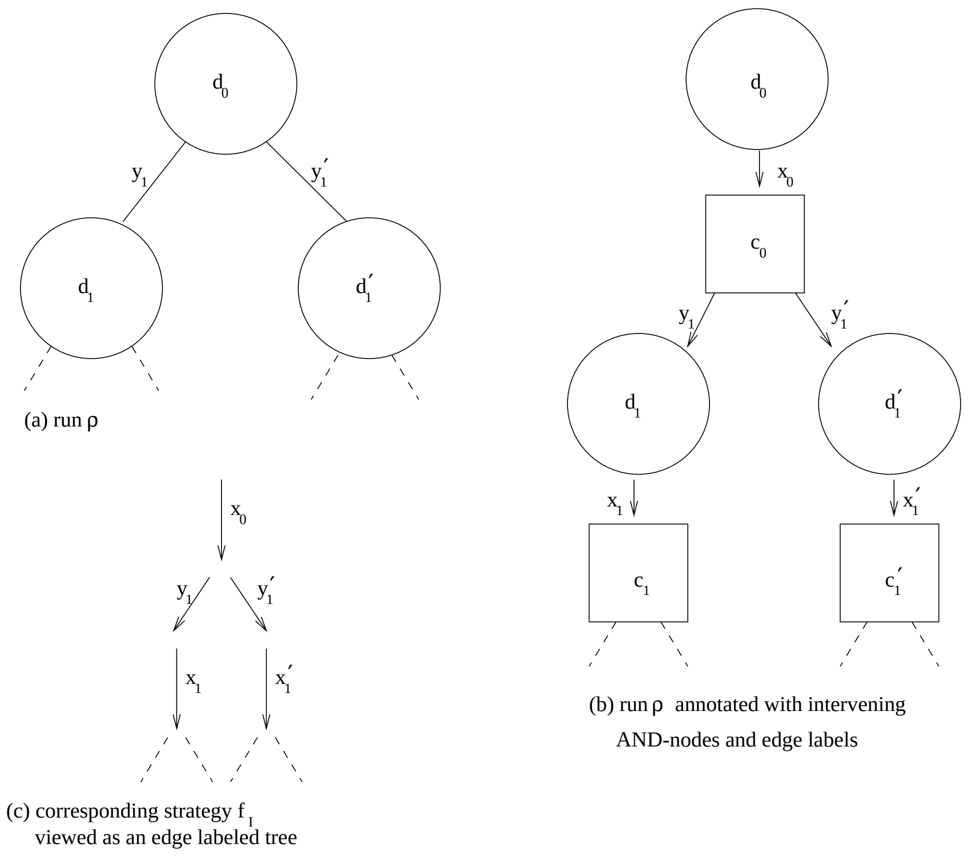

d0
y1
y1′
d1
d1′
(a) run ρ
x0
c0
(b) run ρ annotated with intervening AND-nodes and edge labels
x1
x1′
c1
c1′
(c) corresponding strategy f I viewed as an edge labeled tree

FIG. 5.1.

$\exists f_{\mathrm{I}}$ for all $x \in \{0,1\}^\omega$ $path_{\tilde{A}}(\overline{f_I}(x)\hat{\ }x)$ satisfies $\neg\Phi_{\mathcal{A}}$
iff (by Lemma 5.7)
$\exists$ run ρ of $\tilde{A}$ for all $x \in \{0,1\}^\omega$ $\overline{\rho}(x)$ satisfies $\neg\Phi_{\mathcal{A}}$.

This completes the proof. ∎

LEMMA 5.9. *Γ is Borel.*

Proof. We show that Γ is accepted by a deterministic string automata $\bar{A}$ with acceptance condition $\Phi_{\mathcal{A}}$. $\bar{A}$ has as its alphabet $\{0,1\}$. Let the states of $\bar{A}$ be from $S \cup T$. The transition function $\overline{\delta}$ is g itself, i.e., $\overline{\delta}(u,i) = R(u,i)$. A routine inspection shows that $\bar{A}$ accepts Γ.

Landweber [20] showed that every set accepted by a deterministic Muller string automaton is in the Borel class $G_{\delta\omega} \cap F_{\omega\delta}$ with the usual product topology on the set of all infinite binary sequences. Since we are interested in $\Phi_{\mathcal{A}}$ being either pairs or complemented pairs acceptance condition, the ω-string automaton with such $\Phi_{\mathcal{A}}$ are well known to be equivalent to deterministic Muller string automaton (see, e.g., [20]). ∎

Now, we are ready to prove the main lemma.

LEMMA 5.10. *$\mathcal{A}$ is nonempty iff $\tilde{\mathcal{A}}$ is empty.*

Proof. By Lemma 5.9, the acceptance condition of $\mathcal{A}$, Γ, is Borel, and hence the game $\mathcal{G}$ associated with Γ is determined by Martin's theorem [22]. Thus, I has a winning strategy iff II does not have a winning strategy. By Lemma 5.7, Player I has a winning strategy iff $\mathcal{A}$ is nonempty. By Lemma 5.8, Player II has a winning strategy iff $\tilde{\mathcal{A}}$ is nonempty. It immediately follows then that $\mathcal{A}$ is nonempty iff $\tilde{\mathcal{A}}$ is empty. □

Given a complemented pairs automaton $\mathcal{A}$, we can construct with at most a quadratic blowup its pairs dual automaton $\tilde{\mathcal{A}}$ such that $\mathcal{A}$ is nonempty iff $\tilde{\mathcal{A}}$. Since by Theorem 4.10 of the previous section testing nonemptiness of pairs automata is NP-complete, we immediately get the following corollary.

COROLLARY 5.11. *Testing nonemptiness of complemented pairs automata is co-NP-complete.*

Since the quadratic reduction is deterministic, we also get the following.

COROLLARY 5.12. *There is a deterministic algorithm to test nonemptiness of complemented pairs automata with m states and n pairs that runs in time $(mn)^{O(n)}$.*

6. Applications to logics of programs.

Using existing reductions and our new algorithms for tree automata, we give essentially optimal algorithms for CTL*, PDL-delta, and the Mu-calculus. The "standard" algorithms for each of these logics are of triple exponential complexity (cf. [13, 33, 35]). The existing reductions to which we appeal reduce satisfiability of these logics to nonemptiness of tree automata. These reductions employ determinization of a nondeterministic Büchi automaton on ω-strings. With preexisting algorithms for testing nonemptiness of (pairs) tree automata of [6] (cf. [41]) and the McNaughton determinization construction [23],[11] we can get a nondeterministic double exponential algorithm for testing (un)satisfiability of PDL-delta and the Mu-calculus. Using the new ω-string automaton determinization construct of Safra [32], we can get a nondeterministic single exponential time algorithm for these two logics. As described below, using our new nonemptiness testing algorithms we get a deterministic single exponential time algorithm. The best previously existing upper bounds for these logics, viz., nondeterministic single exponential time (which amounts to deterministic double exponential time in practice), employed "hybrid automata" (cf. [41]). The Safra construction alone does not help reduce the complexity using the hybrid automata. For CTL*, the existing reductions [13], which use a special structure of the ω-string automata associated with the logic, viz., uniqueness of accepting runs, to determinize with only a single exponential blowup, Safra's construction provides no additional help. The previous nonemptiness algorithms give a nondeterministic double exponential time algorithm. Our new nonemptiness algorithm reduces the upper bound to deterministic double exponential time. In what follows $\exp(n)$ denotes the class of functions bounded above by $2^{q(n)}$ for some polynomial $q(n)$.

THEOREM 6.1. *There is an* $\exp(\exp(|p|))$ *time reduction of a CTL* formula p into a pairs automaton $\mathcal{A}_p$ on infinite trees with* $\exp(\exp(|p|))$ *states and* $\exp(|p|)$ *pairs such that p is satisfiable iff $\mathcal{A}_p$ is nonempty.*

THEOREM 6.2. *The satisfiability problem for CTL* is complete for deterministic double exponential time.*

Proof. Applying the nonemptiness algorithm (Theorem 4.1) to the pairs tree automaton obtained by Theorem 6.1, it follows directly that CTL* can be decided in deterministic double exponential time. CTL* was shown hard for double exponential time in [41]. □

THEOREM 6.3. *Satisfiability of the Mu-calculus is in deterministic exponential time.*

[11] Here we also use our Lemma 5.10.

Proof. For a Mu-calculus formula p a complemented pairs automaton $\mathcal{A}_p$ of size $\exp(|p|)$ and number of pairs polynomial in $|p|$ can be constructed in time $\exp(|p|)$ such that p is satisfiable iff $\mathcal{A}_p$ is nonempty. This reduction follows by the technique described in [35], but by using Safra's construction instead of McNaughton's construction. The upper bound follows by using the nonemptiness algorithm for complemented pairs automata (Corollary 5.12). $\square$

PDL-delta has a linear blowup translation into the Mu-calculus [12, 19] (provided we allow consolidation of common subformulae). Hence, we have the following theorem.

THEOREM 6.4. *Satisfiability of PDL-delta is in deterministic exponential time.*

Remark. A direct algorithm, similar to the one mentioned for the Mu-calculus, can also be given (see [33, 41]).

Because of the known deterministic exponential time lower bound for PDL [14], it follows that the above algorithms for the Mu-calculus and PDL-delta are essentially optimal (i.e., up to polynomial blowup). Moreover, we have the next corollary.

COROLLARY 6.5. *Satisfiability of the Mu-calculus and PDL-delta are complete for deterministic exponential time.*

7. Conclusion.

We have investigated the complexity of testing nonemptiness of finite state automata on infinite trees. For the classical pairs acceptance condition of Rabin [30] we show the problem is NP-complete, while for the complemented pairs condition of Streett [33] we show the problem is co-NP-complete. In both cases, we are still able to give a deterministic algorithm that runs in time polynomial in the number of states but exponential in the number of pairs. These nonemptiness algorithms improve previous results in the literature and permit us to give exponentially improved, essentially tight, upper bounds on the complexity of testing satisfiability for a number of important modal logics of programs including CTL*, PDL-delta, and the propositional Mu-calculus. Moreover, we believe that the technique of pseudomodel checking may be useful in connection with other types of automata (cf. [11]).

Among related work we mention the following. Historically, Rabin [30] gave an exponential time algorithm for pairs tree automaton nonemptiness but did not perform a multiparameter analysis or provide a lower bound. Hossley and Rackoff [18] gave an elegant reduction to the nonemptiness problem for automata on finite trees, but the complexity stated for their algorithm was triple exponential. More recently, subsequent to the appearance of the preliminary version of this work in [10], Pnueli and Rosner [28] independently developed a different nonemptiness algorithm for pairs tree automata providing essentially the same upper bound as ours and intended for program synthesis applications; however, they did not consider lower bounds. Their work was extended to a corresponding upper bound for complemented pairs tree automata in [29]. A control-theoretic view of tree automata is given in [36] and [37], while an authoritative survey of automata on infinite objects is presented in [38].

Appendix A.

Proof of Lemma 4.2. We will establish the dual claim. Note that given a pairs condition $\Phi = \bigvee_\gamma(\overset{\infty}{G}P_\gamma \wedge \overset{\infty}{F}Q_\gamma)$, the corresponding "complemented" pairs condition $\hat{\Phi} = \wedge_\gamma(\overset{\infty}{F}P_\gamma \vee \overset{\infty}{G}Q_\gamma)$ is intended to capture the dual property. Interpreted over fullpaths in total structures, which must be infinite paths, $\hat{\Phi} \equiv \tilde{\Phi}$, the actual dual of Φ, i.e., $\neg\Phi(\neg P_1, \ldots, \neg P_\gamma, \neg Q_1, \ldots, \neg Q_\gamma)$. However, over partial structures, as we are permitting, finite fullpaths are allowed and we do not necessarily have the equivalence of $\hat{\Phi}$ and $\tilde{\Phi}$. Rather, $\Phi \equiv \Phi \wedge \mathrm{inf} \equiv \vee_\gamma(FGP_\gamma \wedge GFQ_\gamma) \wedge \mathrm{inf}$. Thus, $\tilde{\Phi} \equiv \wedge_\gamma(GFP_\gamma \vee$

$FGQ_\gamma) \vee fin \equiv (\wedge_\gamma(GFP_\gamma \vee FGQ_\gamma) \wedge \text{inf}) \vee fin \equiv \hat{\Phi} \vee fin$. We thus have that $A(\Phi \vee FR) \equiv A((\Phi \wedge \text{inf}) \vee FR)$ so that the dual $E(\tilde{\Phi} \wedge GR) \equiv E((\hat{\Phi} \vee fin) \wedge GR)$.

We must show that $E(\tilde{\Phi} \wedge GR) \equiv \nu Y.\tilde{\tau}(Y)$, where

$$\tilde{\tau}(Y) = R \wedge \wedge_\gamma EX_w E[RU_s R \wedge (E(\tilde{\Phi}_{-\gamma} \wedge G(R \wedge Q_\gamma)) \vee (P_\gamma \wedge Y) \vee AX_w false)].$$

In the following, let M be an arbitrary structure, which is understood but not explicitly mentioned.

We first show that $E(\tilde{\Phi} \wedge GR)$ is a fixpoint of $\tilde{\tau}(Y)$, i.e., $E(\tilde{\Phi} \wedge GR) \equiv \tilde{\tau}(E(\tilde{\Phi} \wedge GR))$ is valid. For the forward ($\Rightarrow$) direction, assume $s_0 \models E(\tilde{\Phi} \wedge GR)$. Then there is a fullpath x starting at s_0 satisfying $\tilde{\Phi} \wedge GR$, which is equivalent to $(\hat{\Phi} \vee fin) \wedge GR$. By virtue of x and by definition of $\tilde{\Phi}$, for each γ we also have $s_0 \models R \wedge EX_w E[RU_s R \wedge (AX_w false \vee \ (P_\gamma \wedge E(\tilde{\Phi} \wedge GR)) \vee \ E(\tilde{\Phi}_{-\gamma} \wedge G(R \wedge Q_\gamma)))]$, where the $AX_w false$ disjunct ultimately obtains if the fullpath x is finite, the $(P_\gamma \wedge E(\tilde{\Phi} \wedge GR))$ disjunct ultimately obtains if P_γ occurs along the fullpath x, and the $E(\tilde{\Phi}_{-\gamma} \wedge G(R \wedge Q_\gamma))$ disjunct ultimately obtains otherwise, since if P_γ never occurs, then eventually Q_γ must always hold. Hence, s_0 satisfies all conjuncts of $\tilde{\tau}(E(\tilde{\Phi} \wedge R))$, and $s_0 \models \tilde{\tau}(E(\tilde{\Phi} \wedge GR))$. For the converse ($\Leftarrow$) direction, assume $s_0 \models \tilde{\tau}(E(\tilde{\Phi} \wedge GR))$. Pick an arbitrary γ. We have by definition of $\tilde{\tau}(Y)$ that $s_0 \models R \wedge EX_w E[RU_s R \wedge (AX_w false \vee \ (P_\gamma \wedge E(\tilde{\Phi} \wedge GR)) \vee \ E(\tilde{\Phi}_{-\gamma} \wedge G(R \wedge Q_\gamma)))]$. Whichever disjunct ultimately obtains, it follows that $s_0 \models E(\tilde{\Phi} \wedge GR)$. Thus $E(\tilde{\Phi} \wedge GR)$ is a fixpoint of $\tilde{\tau}(Y)$ as desired.

We now show that $E(\tilde{\Phi} \wedge GR)$ is the greatest fixpoint of $\tilde{\tau}(Y)$. Suppose $Z \equiv \tilde{\tau}(Z)$ is an arbitrary fixpoint. We will show that $Z \Rightarrow E(\tilde{\Phi} \wedge GR)$ is valid.

For any state s such that $s \models Z$, since Z is a fixpoint, we have two cases by analyzing $Z \equiv \tilde{\tau}(Z)$.

(a) For some index γ,

$$s \models R \wedge EX_w E[RU_s R \wedge (E(\Phi_{-\gamma} \wedge G(R \wedge Q_\gamma)) \vee AX_w false)],$$

or

(b) for all indices γ, $s \models R \wedge EX_w E[RU_s R \wedge (P_\gamma \wedge Z)]$.

If (b) holds, then we either have the strengthened

(b)$'$ for all indices γ, $s \models R \wedge EX_s E[RU_s R \wedge (P_\gamma \wedge Z)]$, or

(b)$''$ for some index γ, $s \models R \wedge EX_w false$.

But case (b)$''$ implies case (a). Thus either (a) or (b)$'$ must obtain.

Observation A.1. If state s can reach state t by a finite path y where R holds everywhere along y, then if case (a) applies to t, it applies to s, also.

Now let s_0 be an arbitrary state such that $s_0 \models Z$. If case (a) applies to s_0, then $s_0 \models E((\hat{\Phi} \vee fin) \wedge GR)$ and we are done. Otherwise, case (b)$'$ applies. We will show how to use the recursion $Z \equiv \tilde{\tau}(Z)$ infinitely many times to construct an infinite path satisfying $E(\tilde{\Phi} \wedge GR)$.

Since (b)$'$ applies to s_0, there exists a path from s_0 to some state s_1 of the form $s_0 = t_0, t_1, \ldots, t_k = s_1$ ($k \geq 0$) such that $s_1 \models P_1 \wedge Z$ and for all $i \in [0:k]$, $t_i \models R$.

We now do case analysis on s_1. By Observation A.1, we see that case (a) cannot apply to s_1, for if it did, it would apply to s_0 also. So case (b)$'$ applies to s_1, and there exists a path from s_1 to some state s_2 of the form $s_1 = u_0, u_1, \ldots, u_l$ ($l \geq 0$) such that $s_2 \models P_2 \wedge Z$ and for all $j \in [0:l]$, $u_j \models R$.

Continuing in this fashion, we construct an infinite path x of the form

$$s_0 \ldots s_1 \ldots s_2 \ldots s_i \ldots$$

along which R always holds and which successively visits states where $P_1, P_2, \ldots, P_m$, $P_1, P_2, \ldots, P_m, \ldots$, etc. hold in succession. Technically, we have $x \models GR$, and for each i, $s_i \models P_{i \text{ modd } m}$, where i modd m is defined to be i mod m (the remainder of i on division by m) if i mod $m \neq 0$ and to be m if i mod $m = 0$. ▯

Proof of Lemma 4.3. We have $\bigcup_i \tau^i(false)^{T,con} = A(\Phi \vee FR)^{T,con} = A(\Phi \vee FR)^{T,gen} = \bigcup_i \tau^i(false)^{T,gen}$, where the first and third equalities follow from the Tarski–Knaster theorem together with the disjunctivity of *con* and *gen*, while the second equality follows from the generalized linear size model theorem.

It will thus be sufficient to show that for all i,

$$(*) \quad \tau^i(false)^{T,con} \subseteq \mathsf{Y}^{\mathsf{i}} \subseteq \tau^i(false)^{T,gen}.$$

We do this by induction on i.

The base case $i = 0$ is immediate since $\mathsf{Y}^0 = false^{T,con}$.

For the induction step, we assume that (*) holds for i and argue that it holds for $i + 1$.

To establish the first containment $\tau^{i+1}(false)^{T,con} \subseteq \mathsf{Y}^{\mathsf{i+1}}$, assume that $s \in \tau(\tau^i(false))^{T,con}$. Then $M, s \models \tau(\tau^i(false))$ for some M contained in T. It follows, as justified below, that $M, s \models \tau(Y^i)$, from which we conclude that $s \in \tau(Y^i)^{T,con} = \mathsf{Y}^{\mathsf{i+1}}$.

The key step to be justified is that $\tau^i(false)^M \subseteq (Y^i)^M$. Note that for any temporal formula f and for any M contained in T, $f^M \subseteq f^{T,con} \cap M$. And we have by induction hypothesis that $\tau^i(false)^{T,con} \subseteq (Y^i)^{T,con}$. Thus

$$\tau^i(false)^M \subseteq \tau^i(false)^{T,con} \cap M \subseteq (Y^i)^{T,con} \cap M = (Y^i)^M$$

. To establish the second containment $\mathsf{Y}^{\mathsf{i+1}} \subseteq \tau^{i+1}(false)^{T,gen}$, assume $s \in \mathsf{Y}^{\mathsf{i+1}} = \tau(Y^i)^{T,con}$. There exists M contained in T such that $M, s \models \tau(Y^i)$. We will construct from M a new structure M' that will be generated by T and will satisfy $\tau^{i+1}(false)$ at s.

For each state $y \in M \cap \mathsf{Y}^{\mathsf{i}}$, let M_y be a structure generated by T rooted at $\hat{y}$ (a copy of y) such that $M_y, \hat{y} \models \tau^i(false)$. Such an M_y is guaranteed to exist by the induction hypothesis. Let M' be the structure obtained from M by replacing each y by M_y, i.e., redirecting edges from predecessors of y into $\hat{y}$.

It follows that M' is generated by T by virtue of its construction from structures generated by T. It is also the case that $M', s \models \tau(Y^i)$ since each $\hat{y}$ has the same labeling with Y^i as y. Moreover, each $\hat{y}$ is the root of M_y, so $M', s \models \tau^{i+1}(false)$ and $s \in \tau^{i+1}(false)^{T,gen}$. Hence, $s \in \tau^{i+1}(false)^{T,gen}$, establishing the second containment. ▯

Remark. In the construction above, there may be multiple copies of nodes from T in M'; since we do not have a generalized linear size model theorem for $\tau^j(false)$ for $j \geq 2$, it is not, in general, possible to get a structure contained in T. For this reason $Y^i \neq \tau^i(false)^{T,con}$. However, a more involved argument than that above can be given to show that $Y^i = \tau^i(false)^{T,gen}$.

Proof of Lemma 4.4. We use the fixpoint characterization $\mu V.(R \vee AX_s V)$ for AFR. Let $q(V) = R \vee AX_s V$ and $q'(V) = R \vee EX_s AX_s V$.

Now, $T, s \mathrel{\|\!-} AFR$ iff $\exists$ a structure M contained in T such that $M, s \models AFR$ iff $\exists j > 0 \, \exists$ a structure M contained in T such that $M, s \models q^j(false)$.

We will argue by induction on j that

$\exists$ a structure M contained in T such that $M, s \models q^j(false)$ iff $T, s \models (q')^j(false)$ (*), which will, by taking the disjunction over j, establish the lemma.

The base case $j = 1$ is immediate since R is a proposition. Therefore assume (*) holds for j and show that it holds for $j + 1$.

($\Rightarrow$) Assume $\exists$ a structure M contained in T such that $M, s \models q^{j+1}(false)$. Then $M, s \models R \vee AX_s q^j(false)$. If $M, s \models R$, we are done, since R is propositional. If $M, s \models AX_s q^j(false)$, then all successors t_i of s in M (there is at least one) are such that $M, t_i \models q^j(false)$. Hence, $T, t_i || - q^j(false)$ as M is contained in T, and $T, t_i \models (q')^j(false)$ by induction hypothesis. Also, there exists an AND-node c_s from s to the t_i's in T. So $T, s \models EX_s AX_s (q')^j(false)$ and $T, s \models (q')^{j+1}(false)$ as desired.

($\Leftarrow$) Suppose $T, s \models (q')^{j+1}(false)$. Then $T, s \models R \vee EX_s AX_s (q')^j(false)$. If $T, s \models R$, then because R is propositional, $T, s || - R$, and we are done. If $T, s \models EX_s AX_s (q')^j(false)$, then there is an AND-node successor c_s from s to OR-nodes t_0, t_1 such that each $T, t_i \models (q')^j(false)$. By induction hypothesis, for each t_i, $T, t_i || - q^j(false)$, and $\exists$ a structure M_i contained in T such that $M_i, t_i \models q^j(false)$.

We let M_0', M_1' be copies[12] of M_0, M_1, respectively, and graft them onto s by letting t_0', t_1' (the copies of t_0, t_1, respectively) be the successors of s. Then the resulting M' is a structure generated by T such that $M', s \models q^{j+1}(false)$. Now we can get a structure $\hat{M}$ contained in T such that $\hat{M}, s \models q^{j+1}(false)$, as follows.

Suppose nodes u, u' of M' are copies of the same OR-node of T, i.e., map to the same OR-node under the generation function. There is a smallest $k \leq j$ and a smallest $k' \leq j$ such that $M', u \models q^k(false)$ and $M', u' \models q^{k'}(false)$, respectively. If $k \leq k'$, then let u replace u' by redirecting all arcs going from predecessors of u' into u' so that they go from predecessors of u' into u instead. Thus u' is no longer accessible and may be deleted. Similarly, if $k > k'$, then let u' replace u. Call the resulting structure, which has one of u, u' "chopped out," M''. Note that for every node v common to M' and M'', if $M', v \models q^l(false)$, then $M'', v \models q^l(false)$. Accordingly, we have that the "q-rank" of nodes does not increase in going from M' to M''. Moreover, M'' is still a structure generated by T with one fewer pair of duplicates than M' such that $M'', s \models q^j(false)$. This process can be repeated until all duplicates are eliminated. Call the resulting final structure $\hat{M}$. Then $\hat{M}, s \models q^{j+1}(false)$ and is (a copy of) a structure contained in T. □

Proof of Lemma 4.8. The argument depends on the following.

Merging property. If for all $s \in \mathsf{Z}^k$ there exists M contained in Z^k such that $M, s \models g_\gamma \wedge AX_s Z^k \vee R$, then there exists a single M_0 contained in Z^k such that for all $s \in \mathsf{Z}^k$, $M_0, s \models g_\gamma \wedge AX_s Z^k \vee R$.

Proof of property. Let $s_1, \ldots, s_n$ be an enumeration without repetitions of $\mathsf{Z}^k \setminus R$. We will show that we can repeatedly apply the generalized linear size model theorem to get M contained in Z^k such that $M, s_1, \ldots, s_n \models g_\gamma \wedge AX_s Z^k$.

For s_1 there exists M_1 contained in Z^k such that $M_1, s_1 \models g_\gamma \wedge AX_s Z^k$. A similar M_2 for s_2 exists. Let M_1' be a copy of M_1 that is disjoint from M_1 except that the original node s_1 is retained. Similarly, let M_2' be a copy of M_2 that is also disjoint from M_1'. Then, defining the union of two disjoint structures in the obvious way, the structure $M_{12}' = M_1' \cup M_2'$ is generated by Z^k, and we have $M_{12}', s_1, s_2 \models g_\gamma \wedge AX_s Z^k$.

By the generalized linear size model theorem, we can collapse out duplicates yielding M_{12}'' contained in Z^k such that $M_{12}'', s_1'', s_2'' \models g_\gamma$, where M_{12}'' contains no duplicates and s_1'', s_2'' are (possibly copies of) s_1, s_2, respectively.

[12]We say M' is a copy of M if it is an isomorphic structure, i.e., labeled graph, with a fresh set of nodes disjoint from those of M.

Note that for any node t in M''_{12}, if t had successors in M'_{12}, it also has successors in M''_{12} by the nature of the method of chopping out duplicates. Hence, $M''_{12}, s''_1, s''_2 \models g_\gamma \wedge AX_s Z^k$. Of course, M''_{12} is isomorphic to a structure M_{12} contained in Z^k with s''_1, s''_2 corresponding to s_1, s_2, respectively, and $M_{12}, s_1, s_2 \models g_\gamma \wedge AX_s Z^k$.

We now continue with s_3 and use M_{12}, M_3 to get M_{123} contained in Z^k such that $M_{123}, s_1, s_2, s_3 \models g_\gamma \wedge AX_s Z^k$. Continue this process until $\mathsf{Z}^k \setminus R$ is exhausted, yielding $M_{1,\dots,k}$ contained in Z such that $M_{1,\dots,k}, s_1, \dots, s_k \models g_\gamma \wedge AX_s Z^k$. Now let $M_0 = M_{1,\dots,k} \cup N$, where N is the structure formed by just those single nodes t in Z^k satisfying R such that a copy of t does not appear in $M_{1\dots k}$. Then M_0 is contained in Z^k and for all $s \in \mathsf{Z}^k$ we have $M_0, s \models g_\gamma \wedge AX_s Z^k \vee R$.

This completes the proof of the merging property.

To establish soundness of the calculation, i.e., $\mathsf{Z}^k \subseteq A(g_\gamma U_w R)^{T,con}$, we first note that $\mathsf{Z}^k = (g \wedge AX_s \mathsf{Z}^k \vee R)^{\mathsf{Z}^k}$ implies, by definition of the f^{Z^k} notation, for each $s \in \mathsf{Z}^k$ that $\mathsf{Z}^k, s \Vdash_{con} g \wedge AX_s \mathsf{Z}^k \vee R$. By the merging property, this in turn implies that there exists a single M_0 contained in Z^k such that for every $s \in \mathsf{Z}^k \setminus R$, $M_0, s \models g$. It follows that for each $s_0 \in M_0$, $M_0, s_0 \models A(g_\gamma U_w R)$. To see this, let $x = s_0, s_1, s_2, \dots$ be a fullpath in M_0. For each s_i, $M_0, s_i \models g$ or $M_0, s_i \models R$. So $M_0, x \models (g_\gamma U_w R)$ as desired.

By virtue of M_0, for every $s_0 \in \mathsf{Z}^k \setminus R$, we now have that $\mathsf{Z}^k, s_0 \Vdash_{con} A(g_\gamma U_w R)$. Since M_0 is contained in Z^k and $\mathsf{Z}^k \subseteq T$, we also have that M_0 is contained in T, and for each $s_0 \in \mathsf{Z}^k \setminus R$, we get $T, s_0 \Vdash_{con} A(g_\gamma U_w R)$. Since for all $s_0 \in R^{T,con}$, we have $T, s_0 \Vdash_{con} A(g_\gamma U_w R)$, we get for all $s_0 \in \mathsf{Z}^k$ that $T, s_0 \Vdash_{con} A(g_\gamma U_w R)$. Thus, $\mathsf{Z}^k \subseteq A(g_\gamma U_w R)^{T,con}$.

In order to show completeness, i.e., $A(g_\gamma U_w R)^{T,con} \subseteq \mathsf{Z}^k$, we argue that for any i and for any state s_0

$$(*) \qquad \mathsf{Z}^i, s_0 \Vdash_{con} A(g_\gamma U_w R) \quad \text{implies} \quad \mathsf{Z}^{i+1}, s_0 \Vdash_{con} A(g_\gamma U_w R).$$

Assume that $\mathsf{Z}^i, s_0 \Vdash_{con} A(g_\gamma U_w R)$. Thus for some M contained in Z^i we have $M, s_0 \models A(g_\gamma U_w R)$. Without loss of generality, we may assume that every state t of M is reachable from s_0 and $M, t \models A(g_\gamma U_w R)$.

We claim that M is contained in Z^{i+1}. For all $t \in M$, since $M, t \models A(g_\gamma U_w R)$, it follows that $M, t \models (g \wedge AX_s Z^i) \vee R$, and thus $t \in \mathsf{Z}^{i+1} = \{u \in \mathsf{Z}^i : \mathsf{Z}^i, u \Vdash_{con} (g \wedge AX_s Z^i) \vee R\}$. Hence, $M \subseteq \mathsf{Z}^{i+1}$. Since M is contained in Z^i and $M \subseteq \mathsf{Z}^{i+1} \subseteq \mathsf{Z}^i$, it follows by the definition of containment that M is contained in Z^{i+1}.

Thus, $M, s_0 \models A(g_\gamma U_w R)$ and M is contained in Z^{i+1}, so $\mathsf{Z}^{i+1}, s_0 \Vdash_{con} A(g_\gamma U_w R)$ as desired. (Note that by definition of the $\Vdash_{con}$ notation, $\mathsf{Z}^{i+1}, s_0 \Vdash_{con} A(g_\gamma U_w R)$ implies $s_0 \in \mathsf{Z}^{i+1}$ and the above argument guarantees this.)

Since $\mathsf{Z}^0 = T$, for any $s \in A(g_\gamma U_w R)^{T,con}$, $\mathsf{Z}^0, s \Vdash_{con} A(g_\gamma U_w R)$ and by induction on i using $(*)$, we have that for all i, $\mathsf{Z}^i, s \Vdash_{con} A(g_\gamma U_w R)$. In particular, $\mathsf{Z}^k, s \Vdash_{con} A(g_\gamma U_w R)$. Since s was an arbitrary member of $A(g_\gamma U_w R)^{T,con}$, we conclude $A(g_\gamma U_w R)^{T,con} \subseteq \mathsf{Z}^k$, as desired.

To implement the calculation, we see by using disjunctivity of *con* that $((g_\gamma \wedge AX_s \mathsf{Z}^i) \vee R)^{\mathsf{Z}^i,con} = (g_\gamma \wedge AX_s \mathsf{Z}^i)^{\mathsf{Z}^i,con} \cup R^{\mathsf{Z}^i,con}$. Now

$$(g_\gamma \wedge AX_s \mathsf{Z}^i)^{\mathsf{Z}^i,con} = (g_\gamma \wedge AX_s true)^{\mathsf{Z}^i,con}$$

within the scope of $\mathsf{Z}^i, con = g_\gamma^{\mathsf{Z}^i,con}$, since g_γ already has $AX_s true$ as a conjunct. Now apply Lemma 4.9. $\square$

Proof of Lemma 4.9. The $\subseteq$ direction is immediate.

For the $\supseteq$ direction, pick an arbitrary element s of the right-hand side. Because s is in the first term, it must be that there exists an M contained in T such that $M, s \models A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma))$. Since s is also in the second term, it follows that $M, s \models A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma)) \wedge (P_\gamma \vee Y)$. If s has successors in M, then $M, s \models A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma)) \wedge (P_\gamma \vee Y) \wedge AX_s true$, and we are done. If s has no successors in M, then since we still have $M, s \models A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma))$ it must be that $M, s \models R \vee Q_\gamma$. Attach to s the successors t, u it must have by virtue of membership in the third term and call the resulting structure M'. We have $M', s \models A(\Phi_{-\gamma} \vee F(R \vee Q_\gamma)) \wedge (P_\gamma \vee Y) \wedge AX_s true$, the first conjunct holding because s satisfies $R \vee Q_\gamma$ in M and M', the second conjunct holding because s satisfies $P_\gamma \vee Y$ in M and M', and the third conjunct holding by virtue of t, u. Again, $s \in g_\gamma^{T,con}$ and we are done. ◻

REFERENCES

[1] J. R. Büchi, *Using determinacy to eliminate quantifiers*, in Fundamentals of Computation Theory, Lecture Notes in Comput. Sci. 56, Springer-Verlag, Berlin, 1977, pp. 367–378.

[2] B. Banieqbal and H. Barringer, *A Study of an Extended Temporal Language and a Temporal Fixed Point Calculus*, Tech. report UMCS-86-10-2, Computer Science Department, University of Manchester, Manchester, UK, 1986.

[3] J. R. Büchi and L. H. Landweber, *Solving sequential conditions by finite-state strategies*, Trans. Amer. Math. Soc., 138 (1969), pp. 295–311.

[4] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in Proceedings, IBM Workshop on Logics of Programs, Lecture Notes in Comput. Sci. 131, 1981, Springer-Verlag, Berlin, 1982, pp. 52–71.

[5] C. Courcoubetis, M. Y. Vardi, and P. Wolper, *Reasoning about fair concurrent programs*, in Proceedings, 16th ACM Symposium on the Theory of Computing, New York, 1986.

[6] E.A. Emerson, *Automata, tableaux, and temporal logics*, in Proceedings, Workshop on Logics of Programs, Brooklyn, New York, 1985.

[7] E. A. Emerson and E. M. Clarke, *Characterizing correctness properties of parallel programs using fixpoints*, in Seventh International Conference on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 85, Springer-Verlag, Berlin, 1980, pp. 169–182.

[8] E.A. Emerson and E. M. Clarke, *Using branching time logic to synthesize synchronization skeletons*, Sci. Comput. Programming, 2 (1982), pp. 241–266.

[9] E. A. Emerson and J. Y. Halpern, *"Sometimes" and "not never" revisited: On branching versus linear time temporal logic*, J. ACM, 33 (1986), pp. 151–178.

[10] E. A. Emerson and C. S. Jutla, *Complexity of tree automata and modal logics of programs*, in Proceedings, 29th IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1988.

[11] E. A. Emerson and C. S. Jutla, *Tree automata, mu-calculus, and determinacy*, in Proceedings, 33rd IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1991.

[11b] E. A. Emerson and C.-L. Lei, *Modalities for model checking: Branching time strikes back*, Principles of Programming Languages, 1985, pp. 84–96; journal version appears as *Modalities for model checking: Branching time logic strikes back*, Sci. Comput. Programming, 8 (1987), pp. 275–306.

[12] E. A. Emerson and C.-L. Lei, *Efficient model checking in fragments of the propositional mu-calculus*, in Proceedings, IEEE Symposium on Logics in Computer Sci., IEEE Press, Piscataway, NJ, 1986, pp. 267–278.

[13] E. A. Emerson and A. P. Sistla, *Deciding branching time logic*, Inform. and Control, 61 (1984), pp. 175–201.

[14] M. J. Fischer and R. E. Ladner, *Propositional dynamic logic of regular programs*, J. Comput. System Sci., 18 (1979), pp. 194–211.

[15] Y. Gurevich and L. Harrington, *Trees, automata, and games*, in Proceedings, 14th ACM Symposium on the Theory of Computing, New York, 1982.

[16] J. Y. Halpern, *Deterministic process logic is elementary*, in Proceedings, 23rd IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1982.

[17] D. Harel, D. Kozen, and R. Parikh, *Process logic: Expressiveness, decidability, complexity*, J. Comput. System Sci., 25 (1982), pp. 144–170.

[18] R. Hossley and C. Rackoff, *The emptiness problem for automata on infinite trees*, in Switch-

ing and Automata Theory Symposium, IEEE Press, Piscataway, NJ, 1972, pp. 121–124.

[19] D. Kozen, *Results on the propositional mu-calculus*, Theoret. Comput. Sci., 27 (1983), pp. 333–354.

[20] L. H. Landweber, *Decision problems for ω-automata*, Math. Systems Theory, 3 (1969), pp. 376–384.

[21] O. Lichtenstein, A. Pnueli, and L. Zuck, *The glory of the past*, in International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1985, pp. 196–218.

[22] D. A. Martin, *Borel determinacy*, Ann. Math., 102 (1975), pp. 363–371.

[23] R. McNaughton, *Testing and generating infinite sequences by a finite automaton*, Inform. and Control, 9 (1966), pp. 521–530.

[24] D. E. Muller and P. E. Schupp, *Alternating automata on infinite objects, determinacy and Rabin's theorem*, in Automata on Infinite Words, Lecture Notes in Comput. Sci. 192, Springer-Verlag, Berlin, 1985, pp. 100–107.

[25] Z. Manna and P. Wolper, *Synthesis of communicating processes from temporal logic specifications*, ACM TOPLAS, 6 (1984), pp. 68–93.

[26] R. Parikh, *Propositional game logic*, in Proceedings, 25th IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1983, pp. 195–200.

[27] A. Pnueli, *The temporal logic of programs*, in Proceedings, 19th IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1977.

[28] A. Pnueli and R. Rosner, *On the synthesis of a reactive module*, in Proceedings, 16th Annual ACM Symposium on Principles of Programing Languages, ACM Press, NY, 1989, pp. 179–190.

[29] A. Pnueli and R. Rosner, *On the synthesis of an asynchronous reactive module*, in Proceedings, 16th International Colloquium on Automata, Languages, and Programming, Stresa, Italy, Lecture Notes in Comput. Sci. 372, Springer-Verlag, Berlin, 1989, pp. 652–671,

[30] M. O. Rabin, *Decidability of second order theories and automata on infinite trees*, Trans. Amer. Math. Soc, 141 (1969), pp. 1–35.

[31] M. O. Rabin, *Automata on infinite objects and Church's problem*, CBMS Reg. Ser. in Math. 13, AMS, Providence, RI, 1972.

[32] S. Safra, *On Complexity of ω-automata*, in Proceedings, 29th IEEE Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1988.

[33] R. S. Streett, *A Propositional Dynamic Logic of Looping and Converse*, MIT LCS Tech. report TR-263, Massachusettes Institute of Technology, Cambridge, MA, 1981.

[34] A. P. Sistla and E. M. Clarke, *The complexity of propositional linear temporal logic*, J. Assoc. Comput. Mach., 32 (1985), pp. 733–749.

[35] R. S. Streett and E. A. Emerson, *An elementary decision procedure for the mu-calculus*, in Proceedings, 11th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1984,

[36] J. G. Thistle, *Control of Infinite Behavior of Discrete Event Systems*, Ph.D. dissertation, University of Toronto, Toronto, ON, Canada, 1991.

[37] J. Thistle and W. Wonham, *Control of ω-automata, Church's problem, and the emptiness problem for tree ω-automata*, in Computer Science Logic '91, Springer-Verlag, Berlin, 1992, pp. 367–381.

[38] W. Thomas, *Automata on infinite objects*, in Handbook of Theoretical Computer Science, vol. B, J. van Leeuwen, ed., Elsevier/MIT Press, New York, 1990, pp. 133–191.

[39] M. Y. Vardi, *Verification of concurrent programs: The automata-theoretic framework*, Ann. Pure Appl. Logic, 51 (1991), pp. 79–98.

[40] M. Y. Vardi, *A temporal fixpoint calculus*, in Proceedings, 15th Symposium on ACM Principles of Programming Languages, ACM Press, NY, 1988, pp. 250–259.

[41] M. Vardi and L. Stockmeyer, *Improved upper and lower bounds for modal logics of programs*, in Proceedings, 17th ACM Symposium on the Theory of Computing, New York, 1985, pp. 240–251.

[42] M. Y. Vardi and P. L. Wolper, *Yet another process logic*, in CMU Workshop on Logics of Programs, Lecture Notes in Comput. Sci. 164, Springer-Verlag, Berlin, 1983, pp. 501–512.

[43] M. Y. Vardi and P. L. Wolper, *Automata-theoretic techniques for modal logics of programs*, in Proceedings, 14th ACM Symposium on the Theory of Computing, Washington, DC, J. Comput. System Sci., 32 (1986), pp. 183–221.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 159–179

FINDING SEPARATOR CUTS IN PLANAR GRAPHS WITHIN TWICE THE OPTIMAL[*]

NAVEEN GARG[†], HUZUR SARAN[‡], AND VIJAY V. VAZIRANI[§]

Abstract. A factor 2 approximation algorithm for the problem of finding a minimum-cost b-balanced cut in planar graphs is presented, for $b \leq \frac{1}{3}$. We assume that the vertex weights are given in unary; for the case of binary vertex weights, a pseudoapproximation algorithm is presented. This problem is of considerable practical significance, especially in VLSI design.

The natural algorithm for this problem accumulates sparsest cuts iteratively. One of our main ideas is to give a definition of sparsity, called *net-sparsity*, that reflects precisely the cost of the cuts accumulated by this algorithm. However, this definition is too precise: we believe it is **NP**-hard to compute a minimum–net-sparsity cut, even in planar graphs. The rest of our machinery is built to work with this definition and still make it computationally feasible. Toward this end, we use several ideas from the works of Rao [*Proceedings,* 28th *Annual IEEE Symposium on Foundations of Computer Science*, 1987, pp. 225–237; *Proceedings,* 24th *Annual ACM Symposium on Theory of Computing*, 1992, pp. 229–240] and Park and Phillips [*Proceedings,* 25th *Annual ACM Symposium on Theory of Computing*, 1993, pp. 766–775].

Key words. separators, planar graphs, approximation algorithms

AMS subject classifications. 68Q25, 68Q35, 90C27, 05C85

PII. S0097539794271692

1. Introduction.

Given an undirected graph with edge costs and vertex weights, the *balance* of a cut is the ratio of the weight of vertices on the smaller side to the total weight in the graph. For $0 < b \leq \frac{1}{2}$, a cut having a balance of at least b is called a b-balanced cut; a $\frac{1}{3}$-balanced cut is given the special name *separator*. In this paper, we present a factor 2 approximation algorithm for finding a minimum-cost b-balanced cut in planar graphs, for $b \leq \frac{1}{3}$, assuming that vertex weights are given in unary. We also give examples to show that our analysis is tight. For the case of binary vertex weights, we use scaling to give a pseudoapproximation algorithm: for each $\alpha > 2/b$, it finds a $(b - 2/\alpha)$-balanced cut of cost within twice the cost of an optimal b-balanced cut, for $b \leq 1/3$, in time polynomial in n and α. The previous best approximation guarantee known for b-balanced cuts in planar graphs was $O(\log n)$, due to Rao [8, 9]; for general graphs, no approximation algorithms are known.

The problem of breaking a graph into "small"-sized pieces by removal of a "small" set of edges or vertices has attracted much attention since the seminal work of Lipton and Tarjan [5], because this opens up the possibility of a divide-and-conquer strategy for the solution of several problems on the graph. Small balanced cuts have numerous applications; see, for example, [1, 3, 4, 6]. Several of these applications pertain to planar graphs, the most important one being circuit partitioning in VLSI design.

The *sparsity* of a cut is defined to be the ratio of the cost of the cut and the weight on its smaller side, and a cut having minimum sparsity in the graph is called

[*]Received by the editors July 26, 1994; accepted for publication (in revised form) May 21, 1997; published electronically September 14, 1999.
http://www.siam.org/journals/sicomp/29-1/27169.html

[†]Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi 110016, India (naveen@cse.iitd.ernet.in).

[‡]Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi 110016, India (saran@iitd.ernet.in).

[§]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 (vazirani@cc.gatech.edu). The research of this author was supported by NSF grant CCR-9627308.

the *sparsest cut*. Rao [9] gave a $\frac{3}{2}$-approximation algorithm for the problem of finding a sparsest cut in planar graphs, and recently Park and Phillips [7] showed that this problem is polynomial time–solvable. A sparsest cut limits multicommodity flow in the same way that a min-cut limits max-flow. Leighton and Rao [3] derived an approximate max-flow min-cut theorem for uniform multicommodity flow and in the process gave an $O(\log n)$-approximation algorithm for finding a sparsest cut in general graphs. By finding and removing these cuts iteratively, one can show how to find in planar (general) graphs a b-balanced cut that is within an $O(1)$ factor ($O(\log n)$ factor) of the optimal b'-balanced cut for $b < b'$, and $b \leq \frac{1}{3}$ [8, 9, 3]. For instance, using the Park–Phillips algorithm for sparsest cut in planar graphs, this approach gives a $\frac{1}{3}$-balanced cut that is within 7.1 times the cost of the best $\frac{1}{2}$-balanced cut in planar graphs. Notice, however, that these are not true approximation algorithms, since the best $\frac{1}{2}$-balanced cut may have a much higher cost than the best $\frac{1}{3}$-balanced cut.

This iterative algorithm has shortcomings which prevent it from leading to a good true approximation algorithm; these are illustrated via an example in section 3. One of our main ideas is to give a definition of sparsity, called *net-sparsity*, that overcomes these shortcomings. The notion of *net-cost*, on which this definition of net-sparsity is based, reflects *precisely* the cost of the cuts accumulated iteratively. Indeed, it is too precise to be directly useful computationally—we believe that computing the sparsest cut under this definition is **NP**-hard even in planar graphs. The rest of our machinery is built to work with this definition and still make it computationally feasible, and we manage to scrape by narrowly!

Planarity is exploited in several ways: First, a cut in a planar graph corresponds to a set of cycles in the dual. Second, the notion of a *transfer function* turns out to be very useful. Given a planar graph with weights on faces, this notion can be used to define a function on the edges of the graph so that on any cycle it evaluates to the sum of the weights of the faces enclosed by the cycle. Such an idea has been used in the past by Kasteleyn [2] for computing the number of perfect matchings in a planar graph in polynomial time. Kasteleyn defined his function over $GF[2]$. Park and Phillips [7] first defined the function over reals, thereby demonstrating the full power of this notion.

Park and Phillips [7] have shown that the problems of finding a sparsest cut and a minimum b-balanced cut in planar graphs are weakly **NP**-hard; i.e., these problems are **NP**-hard if the vertex weights are given in binary. Indeed, the algorithm they give for finding the sparsest cut in planar graphs is a pseudopolynomial time algorithm. As a consequence of this algorithm, it follows that if $\mathbf{P} \neq \mathbf{NP}$, finding sparsest cuts in planar graphs is not strongly **NP**-hard. On the other hand it is not known if the b-balanced cut problem in planar graphs is strongly **NP**-hard or if there is a pseudopolynomial time algorithm for it. (The present paper gives only a pseudopolynomial approximation algorithm.) Park and Phillips leave open the question of finding a fully polynomial approximation scheme for sparsest cuts in planar graphs, i.e., if the vertex weights are given in binary. We give such an algorithm using a scaling technique.

2. Preliminaries.

Let $G = (V, E)$ be a connected undirected graph with an edge cost function $c : E \to \mathbf{R}^+$ and a vertex weight function $wt : V \to \mathbf{Z}^+$. Any function that we define on the elements of a universe extends to sets of elements in the obvious manner; the value of the function on a set is the sum of its values on the elements in the set. Let W be the sum of weights of all vertices in G. A partition $(S, \overline{S})$ of V defines a *cut* in G; the cut consists of all edges that have one end point in

S and the other in $\overline{S}$. A set of vertices, S, is said to be *connected* when the subgraph induced on it is connected. If either S or $\overline{S}$ is connected then cut $(S, \overline{S})$ will be called a *simple cut*, and when both S and $\overline{S}$ are connected then the cut $(S, \overline{S})$ is called a *bond*.

Given a set of vertices $S \subset V$, we define the cost of this set, $\text{cost}(S)$, as the sum of the costs of all edges in the cut $(S, \overline{S})$. The weight of the set S, $\text{wt}(S)$, is the sum of the weights of the vertices included in S.

A cut $(S, \overline{S})$ is a *separator* if $\frac{W}{3} \le \text{wt}(S), \text{wt}(\overline{S}) \le \frac{2W}{3}$. The cost of a separator is the sum of the costs of the edges in the separator.

LEMMA 2.1. *For any connected graph G there exists a minimum-cost separator, $(S, \overline{S})$, which is a simple cut. Further, if S is the side that is not connected, then each connected component of S has weight strictly less than $\frac{W}{3}$.*

Proof. Let $(S, \overline{S})$ be a minimum-cost separator in G. Consider the connected components obtained on removing the edges of this separator. Clearly, no component has weight strictly larger than $\frac{2W}{3}$. If all components have weight strictly less than $\frac{W}{3}$ then both S and $\overline{S}$ are not connected and we can arrive at a contradiction as follows. We first pick two components that have an edge between them and then pick the remaining, one by one, in an arbitrary order until we accumulate a weight of at least $\frac{W}{3}$. The accumulated weight cannot exceed $\frac{2W}{3}$ since each component has weight at most $\frac{W}{3}$. Thus we obtain a separator of cost strictly less than the cost of the separator $(S, \overline{S})$, a contradiction. Hence, at least one component has weight between $\frac{W}{3}$ and $\frac{2W}{3}$. If there are two such components then these are the only components since by switching the side of a third component we obtain a cheaper separator. If there is only one component of weight between $\frac{W}{3}$ and $\frac{2W}{3}$ then this separator is optimum iff this component forms one side of the cut and the remaining components the other side. Thus $(S, \overline{S})$ is a bond and if some side of the cut is not connected then all components on that side have weight strictly less than $\frac{W}{3}$. $\square$

Hence there always exists a minimum-cost separator that is a simple cut. Let OPT denote the set of vertices on the side of this separator that is not connected.

3. Overview of the algorithm.

Let S be a set of vertices such that $\text{wt}(S) \le \text{wt}(\overline{S})$. The *sparsity* of S is usually defined as the quotient of the cost and the weight of this set, i.e.,

$$\text{sparsity}(S) = \frac{\text{cost}(S)}{\text{wt}(S)}.$$

A natural approach to finding good separators is to repeatedly find a set of minimum sparsity and remove it from the graph, eventually reporting the union of the removed vertices. It is easy to concoct “bad” examples for this approach by ensuring that the picked vertices always come from the smaller side of the optimal separator, thereby ensuring that the minimum sparsity available in the remaining graph keeps increasing. This is illustrated in Figure 3.1; here the first cut picked has a sparsity of $\frac{1}{m}$, whereas the last cut has a sparsity of $\frac{1}{2}$.

This approach has two shortcomings: it removes the vertices picked in each iteration and deals only with the remaining graph in subsequent iterations, and it assumes that edges of the cuts found in each iteration are picked permanently, even though they may not be needed in the final cut. One of our main ideas is to give a definition of “sparsity” under which this algorithm does not suffer from either of these shortcomings.

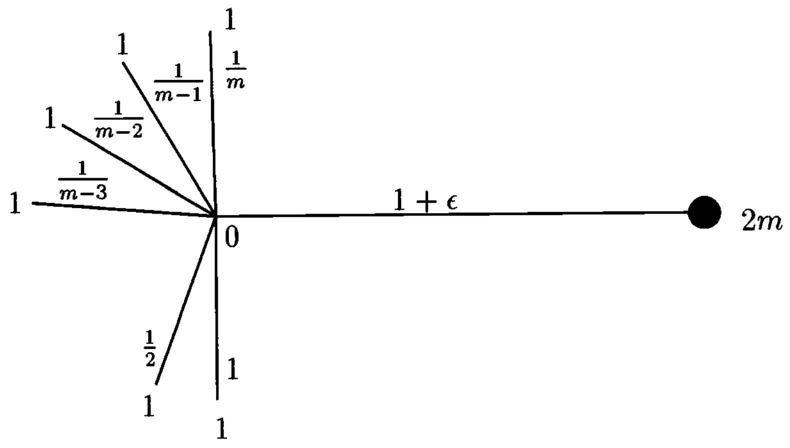


FIG. 3.1. *Graph with vertex weights and edge costs showing how minimum sparsity increases. Here* $m = \frac{n}{3}$.

Let $S, T \subset V$ be two sets of vertices. Define the *net-cost* of S with respect to T as

$$\text{net-cost}_T(S) = \text{cost}(T \cup S) - \text{cost}(T)$$

and the *net-weight* of S with respect to T as

$$\text{net-weight}_T(S) = \text{wt}(T \cup S) - \text{wt}(T).$$

Thus, if we have already picked the set of vertices T, then net-cost$_T(S)$ measures the extra cost incurred and net-weight$_T(S)$ the weight added in picking the set $S \cup T$. Finally, define the *net-sparsity* of S with respect to T as

$$\text{net-sparsity}_T(S) = \frac{\text{net-cost}_T(S)}{\text{net-weight}_T(S)}.$$

For any algorithm that picks a cut by accumulating sets of vertices, the notion of net-cost gives precisely the extra cost incurred in each iteration. But is it so precise that computing the sparsest cut under this definition turns out to be **NP**-hard even in planar graphs? Although we do not have an answer to this question, we believe that it is "yes"! Indeed, the rest of our machinery is built to work with this definition and still make it computationally feasible, and we manage to scrape by narrowly!

Let us first show that it is not sufficient to just keep picking sets of minimum net-sparsity. Consider the following example: Suppose OPT $= S_1 \cup S_2$, where S_1 is a very sparse set of weight $\frac{W}{3} - \epsilon$ and S_2 is a set of high sparsity and weight ϵ for a small ϵ. Having picked S_1, we might pick another set, S_3 of sparsity almost that of S_2, and weight $\frac{W}{3} - \epsilon$; hence, the cost incurred would be arbitrarily high compared to the optimum.

We get around this difficulty by ensuring that in each iteration the set of vertices we pick is such that the total weight accumulated is strictly under $\frac{W}{3}$. More formally, let T_{i-1} be the set of vertices picked by the end of the $(i-1)$th iteration ($T_0 = \phi$). In the ith iteration we pick a set D_i such that

[weight] $\mathrm{wt}(T_{i-1} \cup D_i) < \frac{W}{3}$.
[net-sparsity] $\forall S : \mathrm{wt}(T_{i-1} \cup S) < \frac{W}{3}$, $\text{net-sparsity}_{T_{i-1}}(D_i) \leq \text{net-sparsity}_{T_{i-1}}(S)$.
[minimality] D_i has minimum net-weight among all sets satisfying the above conditions.

We call set D_i a *dot* and denote it by $\bullet$. Thus, at the end of the ith iteration the set of vertices we have picked is given by $T_i = T_{i-1} \cup D_i$. This is how we augment the "partial solution" $(T_1, T_2, \ldots, T_i)$ in each iteration.

How do we ever obtain a "complete solution" (a separator)? In the ith iteration, besides augmenting the partial solution T_{i-1} to the partial solution T_i we also augment it to a complete solution, i.e., we pick a set of vertices B_i such that

[weight] $\frac{W}{3} \leq \mathrm{wt}(T_{i-1} \cup B_i) \leq \frac{2W}{3}$.
[cost] $\forall S : \frac{W}{3} \leq \mathrm{wt}(T_{i-1} \cup S) \leq \frac{2W}{3}$, $\mathrm{cost}(B_i) \leq \mathrm{cost}(S)$.

Since $T_0 = \phi$, finding the set B_1 corresponds to finding the minimum-cost separator. To avoid this circularity in the argument we restrict B_i to a smaller class of sets:

[cost] $\forall S : (S, \overline{S})$ is a bond and $\frac{W}{3} \leq \mathrm{wt}(T_{i-1} \cup S) \leq \frac{2W}{3}$, $\mathrm{cost}(B_i) \leq \mathrm{cost}(S)$.

We call the set B_i a *box* and denote it by $\square$. Notice that a box set need not be a bond and that we count a $\square$ at its cost rather than its net-cost. This is done only to simplify the algorithm and its analysis. The example which shows that the analysis of our algorithm is tight also shows that counting the $\square$ at its net-cost would not have led to any improvement in the approximation guarantee.

Thus, in each iteration we obtain a separator. The solution reported by the algorithm is the one of minimum cost from among all these separators. The algorithm, which we call the *dot-box algorithm*, is the following.

Dot-Box Algorithm.

```
    1. MINSOL ← $\infty$, $i \leftarrow 0$, $T_0 \leftarrow \phi$
    2. while $\mathrm{wt}(T_i) < \frac{W}{3}$ do
          2.1. $i \leftarrow i + 1$
          2.2. Find $\bullet$ and $\square$ sets, $D_i$ and $B_i$, respectively.
               If there is no $\bullet$ set, exit.
          2.3. MINSOL ← min(MINSOL, $\mathrm{cost}(T_{i-1}) + \mathrm{cost}(B_i)$)
          2.4. $T_i \leftarrow T_{i-1} \cup D_i$
end.
```

We make two remarks regarding step (2.2). First, we conjecture that finding the $\bullet$ set is **NP**-hard. Our procedure to find $\bullet$ sets may not always succeed; however, we will prove that if it fails, then the $\square$ set found in the current iteration gives a separator within twice OPT. Second, at some iteration it might be the case that no subset of vertices satisfies the weight criterion for a $\bullet$, since each set takes the total weight accumulated to $W/3$ or more. In this case, the dot-box algorithm halts and outputs the best separator found so far.

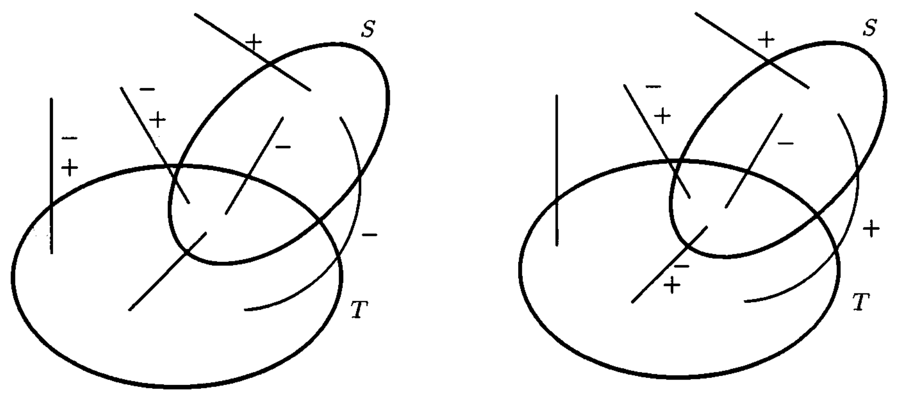


FIG. 4.1. *Computation of* net-cost$_T(S)$ *and* net-cost$_{S\cap T}(S)$. *A* $+/-$ *on an edge signifies that the edge is counted in the positive/negative term in the net-cost computation.*

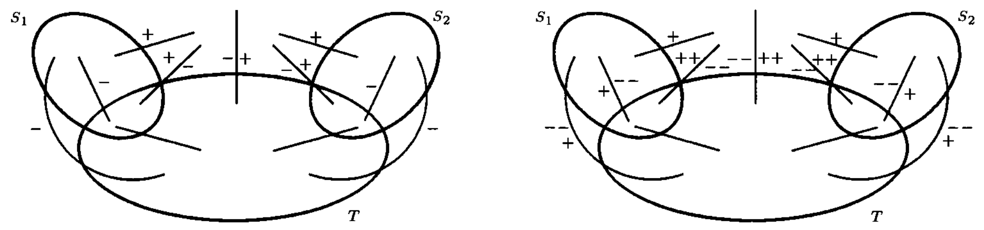


FIG. 4.2. *Computation of* net-cost$_T(S_1 \cup S_2)$ *and* net-cost$_T(S_1)$+net-cost$_T(S_2)$.

4. Analysis of the dot-box algorithm.

We first prove some properties of net-cost and net-weight which will be useful for the analysis. From the definition of net-cost and net-weight we have that net-cost$_T(S)$ = net-cost$_T(S-T)$ and net-weight$_T(S)$ = net-weight$_T(S-T)$ = wt$(S-T)$. The following property also follows from the definitions

PROPERTY 4.1. *Let S_1, S_2 be two sets of vertices not necessarily disjoint. Then*

$$\text{net-cost}_T(S_1 \cup S_2) = \text{net-cost}_T(S_1) + \text{net-cost}_{T\cup S_1}(S_2),$$
$$\text{net-weight}_T(S_1 \cup S_2) = \text{net-weight}_T(S_1) + \text{net-weight}_{T\cup S_1}(S_2).$$

PROPERTY 4.2. net-cost$_T(S) \leq$ net-cost$_{S\cap T}(S)$.

Proof. Figure 4.1 shows the edges between the sets $S\cap T$, $S-T$, $T-S$, and $\overline{S\cup T}$ from which the above property is immediate. The net-cost of S with respect to $S\cap T$ may be higher because it includes the cost of edges from $S-T$ to $T-S$. □

The following property is immediate from Figure 4.2.

PROPERTY 4.3. *Let S_1, S_2 be two disjoint sets of vertices with no edges between them. Then*

$$\text{net-cost}_T(S_1 \cup S_2) = \text{net-cost}_T(S_1) + \text{net-cost}_T(S_2).$$

REMARK 4.1. *For positive real numbers a, b, c, d,*

$$\min\left(\frac{a}{b}, \frac{c}{d}\right) \leq \frac{a+c}{b+d} \leq \max\left(\frac{a}{b}, \frac{c}{d}\right).$$

Further, let $a_i, b_i, 1 \leq i \leq k$, be positive real numbers. Then, $\exists i, 1 \leq i \leq k$, such that

$$\frac{a_i}{b_i} \leq \frac{\sum_{j=1}^{k} a_j}{\sum_{j=1}^{k} b_j}.$$

LEMMA 4.1. *The net-sparsity of the ●'s is increasing, i.e.,*

$$\forall i : i \geq 1, \quad \text{net-sparsity}_{T_{i-1}}(D_i) \leq \text{net-sparsity}_{T_i}(D_{i+1}).$$

Proof. Since the set $D_i \cup D_{i+1}$ satisfies the weight requirement for a ● at the ith iteration,

$$\text{net-sparsity}_{T_{i-1}}(D_i) \leq \text{net-sparsity}_{T_{i-1}}(D_i \cup D_{i+1}).$$

By Property 4.1,

$$\text{net-cost}_{T_{i-1}}(D_i \cup D_{i+1}) = \text{net-cost}_{T_{i-1}}(D_i) + \text{net-cost}_{T_i}(D_{i+1}),$$
$$\text{net-weight}_{T_{i-1}}(D_i \cup D_{i+1}) = \text{net-weight}_{T_{i-1}}(D_i) + \text{net-weight}_{T_i}(D_{i+1}),$$

which, using Remark 4.1, gives us

$$\begin{aligned}\min(\text{net-sparsity}_{T_{i-1}}(D_i), \text{net-sparsity}_{T_i}(D_{i+1})) &\leq \text{net-sparsity}_{T_{i-1}}(D_i \cup D_{i+1}) \\ &\leq \max(\text{net-sparsity}_{T_{i-1}}(D_i), \\ &\qquad \text{net-sparsity}_{T_i}(D_{i+1})).\end{aligned}$$

Now, by the first inequality, it must be the case that

$$\max(\text{net-sparsity}_{T_{i-1}}(D_i), \text{net-sparsity}_{T_i}(D_{i+1})) = \text{net-sparsity}_{T_i}(D_{i+1}).$$

The lemma follows. ▯

Let k be the first iteration at which some connected component of OPT meets the weight requirement of a □.

LEMMA 4.2. $\forall i : 1 \leq i \leq k-1$, $\text{net-sparsity}_{T_{i-1}}(D_i) \leq \text{net-sparsity}_{T_{i-1}}(\text{OPT})$.

Proof. Since OPT $\not\subseteq T_{i-1}$ there are connected components of OPT which are not completely contained in T_{i-1}. By assumption, none of these components satisfies the weight requirement for a □; hence each of these components meets the weight requirement for a ●. Hence the ● picked in this iteration should have net-sparsity at most that of any of these components.

By Property 4.3, the net-cost of OPT is the sum of the net-costs of these components of OPT. The same is true for net-weight; hence the component of OPT with minimum net-sparsity has net-sparsity less than that of OPT. The lemma follows. ▯

The above two lemmas imply that the net-sparsity at which the ●'s are picked is increasing and that for any iteration before the kth, this net-sparsity is less than the net-sparsity of OPT in that iteration.

LEMMA 4.3. $\forall i : 1 \leq i \leq k-1$, $\text{cost}(T_i) < \text{cost}(\text{OPT})$.

Proof. To establish this inequality for i we consider two processes.

1. The first process is our algorithm which picks the set of vertices D_j at the jth step, $1 \leq j \leq i$.
2. The second process picks the vertices $D_j \cap \text{OPT}$ at the jth step, $1 \leq j < i$. At the ith step it picks the remaining vertices of OPT.

Let P_j be the set of vertices picked by the second process in the first j steps. Then $P_j = \mathrm{OPT} \cap T_j, 1 \le j < i$ and $P_i = \mathrm{OPT}$. At the jth step the second process picks an additional weight of net-weight$_{P_{j-1}}(P_j)$ at a cost of net-cost$_{P_{j-1}}(P_j)$. By the fact that the second process picks a subset of what the first process picks at each step we have the following.

CLAIM 4.1. *For* $1 \le j \le i-1$, net-weight$_{T_{j-1}}(D_j) \ge$ net-weight$_{P_{j-1}}(P_j)$.

CLAIM 4.2. *For* $1 \le j \le i$, net-sparsity$_{T_{j-1}}(D_j) \le$ net-sparsity$_{P_{j-1}}(P_j)$.

Proof. Since $P_j \cap T_{j-1} = P_{j-1}$, by Property 4.2 we have

$$\text{net-cost}_{T_{j-1}}(P_j) \le \text{net-cost}_{P_{j-1}}(P_j).$$

Further,

$$\text{net-weight}_{T_{j-1}}(P_j) = \text{net-weight}_{P_{j-1}}(P_j)$$

and hence net-sparsity$_{T_{j-1}}(P_j) \le$ net-sparsity$_{P_{j-1}}(P_j)$.

For $j < i$, the claim follows since P_j satisfies the weight requirement for a $\bullet$ and D_j was picked as the $\bullet$. For $j = i$, $P_j = \mathrm{OPT}$ and the claim follows from Lemma 4.2. $\square$

The above claims imply that in each iteration (1 through i) the first process picks vertices at a lower net-sparsity than the second process. If both these processes were picking the same additional weight in each iteration then this fact alone would have implied that the cost of the vertices picked by the first process is less than the cost of the vertices picked by the second. But this is not the case. What is true, however, is the fact that in iterations 1 through $i-1$ the first process picks a larger additional weight than the second process. In the ith iteration, the second process picks enough additional weight so that it now has accumulated a total weight strictly larger than that picked by the first process (since $\mathrm{wt}(\mathrm{OPT}) \ge \frac{W}{3} > \mathrm{wt}(T_i)$). But the net-sparsity at which the second process picks vertices in the ith iteration is more than the maximum (over iterations 1 through i) net-sparsity at which the first process picks vertices. So it follows that the cost of the vertices picked by the first process is strictly less than the cost of the vertices picked by the second, i.e., $\mathrm{cost}(T_i) < \mathrm{cost}(\mathrm{OPT})$. $\square$

Consider the separator found in the kth iteration, i.e., the cut $(T_{k-1} \cup B_k, \overline{T_{k-1} \cup B_k})$. This solution is formed by picking $\bullet$'s in the first $k-1$ steps and a $\square$ in the kth step.

LEMMA 4.4. $\mathrm{cost}(T_{k-1} \cup B_k) \le 2 \cdot \mathrm{cost}(\mathrm{OPT})$.

Proof. Any connected component of OPT is a bond. In the kth iteration there exists a connected component of OPT, say OPT_j, such that $\frac{W}{3} \le \mathrm{wt}(T_{k-1} \cup \mathrm{OPT}_j) \le \frac{2W}{3}$. Hence the $\square$ at the kth step should have cost at most $\mathrm{cost}(\mathrm{OPT}_j)$, i.e., $\mathrm{cost}(B_k) \le \mathrm{cost}(\mathrm{OPT}_j) \le \mathrm{cost}(\mathrm{OPT})$.

From Lemma 4.3 we know that $\mathrm{cost}(T_{k-1}) < \mathrm{cost}(\mathrm{OPT})$. Hence

$$\mathrm{cost}(T_{k-1} \cup B_k) < \mathrm{cost}(T_{k-1}) + \mathrm{cost}(B_k) < 2 \cdot \mathrm{cost}(\mathrm{OPT}). \qquad \square$$

Since the dot-box algorithm outputs the best separator found, we have the following theorem.

THEOREM 4.5. *The cost of the separator found by the dot-box algorithm is at most twice the cost of* OPT.

Our analysis of the dot-box algorithm is tight; when run on the example in Figure 4.3, it picks a separator of cost almost twice the optimum. In this example,

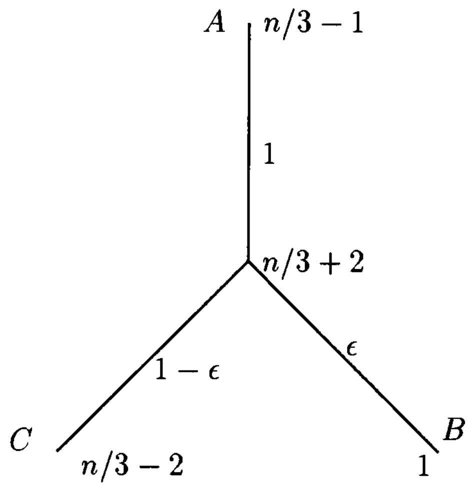


FIG. 4.3. *A tight example for our analysis. Vertex weights and edge costs are given and $\epsilon = 3/n$.*

OPT $= A \cup B$ and so cost(OPT) $= 1 + \epsilon$. For $\epsilon > 3/n$, the $\bullet$ in the first iteration is the set C and the $\Box$ is the set A. This separator, which is also the one returned by the dot-box algorithm, has cost $2 - 2\epsilon$ and hence the approximation ratio is $2(\frac{n-3}{n+3})$.

5. Structural properties of our solution, and a computationally easier definition of net-cost. In this section we prove some structural properties of the solution found by the dot-box algorithm. This allows us to redefine net-cost in such a manner that it becomes computationally easier and yet the analysis from the previous section continues to hold.

LEMMA 5.1. *For $1 \le i \le k-1$, $\overline{T_i}$ is connected.*

Proof. For contradiction assume that $\overline{T_i}$ is not connected. Let A be a connected component of $\overline{T_i}$. There are three cases.

$\text{wt}(T_i \cup A) < \mathbf{W/3}$: The set A satisfies the weight requirement for a $\bullet$ at the $(i+1)$th iteration. Since A has edges only to vertices in T_i, the net-cost of A with respect to T_i is negative. Hence net-sparsity$_{T_i}(A)$ is negative, contradicting Lemma 4.1.

$\mathbf{W/3} \le \text{wt}(T_i \cup A) \le \mathbf{2W/3}$: The cut $(T_i \cup A, \overline{T_i \cup A})$ is a separator of cost, $\text{cost}(T_i \cup A) < \text{cost}(T_i) < \text{cost(OPT)}$, a contradiction.

$\text{wt}(T_i \cup A) > \mathbf{2W/3}$: Since $\text{wt}(T_i) < W/3$, the condition of this case implies that $\text{wt}(A) > W/3$. If $\text{wt}(A) \le 2W/3$ then once again we have a contradiction since now the cut $(A, \overline{A})$ is a separator of cost, $\text{cost}(A) < \text{cost}(T_i) < \text{cost(OPT)}$. Thus it must be the case that $\text{wt}(A) > 2W/3$.

Since the above argument implies that each connected component of $\overline{T_i}$ has weight greater than $2W/3$, $\overline{T_i}$ must have only one connected component. ∎

LEMMA 5.2. *For $1 \le i \le k-1$, the set $T_i - T_{i-1}$ is connected.*

Proof. For contradiction assume that $T_i - T_{i-1}$ is not connected. Let A be a connected component of $T_i - T_{i-1}$ and B be the rest of $T_i - T_{i-1}$. (We also denote by A, B the corresponding sets of vertices.)

If D_i is the $\bullet$ at the ith iteration then $\text{net-cost}_{T_{i-1}}(D_i) = \text{net-cost}_{T_{i-1}}(A \cup B)$.

Since A and B are disjoint set of vertices with no edges between them, by Property 4.3

$$\text{net-cost}_{T_{i-1}}(D_i) = \text{net-cost}_{T_{i-1}}(A) + \text{net-cost}_{T_{i-1}}(B).$$

Further,

$$\text{net-weight}_{T_{i-1}}(D_i) = \text{net-weight}_{T_{i-1}}(A) + \text{net-weight}_{T_{i-1}}(B).$$

Thus it is the case that either A or B has smaller net-sparsity than D_i, which contradicts the assumption that D_i is a $\bullet$, or both A and B have the same net-sparsity as D_i, but this contradicts the minimality requirement on D_i. $\square$

LEMMA 5.3. *For every iteration $i : 1 \leq i \leq k-1$, there exists a $\bullet$, D_i, satisfying the following:*

1. *$(D_i, \overline{D_i})$ is a bond.*
2. *Each connected component of T_{i-1} is contained in D_i or $\overline{D_i}$ and there is no edge between D_i and components of T_{i-1} in $\overline{D_i}$.*

Proof. The set $T_i - T_{i-1}$ together with any subset of T_{i-1} is also a $\bullet$ for the ith iteration. We form a new $\bullet$, D_i, by merging with $T_i - T_{i-1}$ all connected components of T_{i-1} which have an edge to $T_i - T_{i-1}$. Since $T_i - T_{i-1}$ is connected, so is D_i. Further, since the graph is connected, every remaining component of T_{i-1} has an edge to $\overline{T_i}$ so that $\overline{D_i}$ is also connected. Thus $(D_i, \overline{D_i})$ is a bond. It follows from the definition of D_i that there is no edge between D_i and components of T_{i-1} in $\overline{D_i}$. $\square$

Since every $\bullet$ in iterations 1 through $k-1$ satisfies the conditions in Lemma 5.3, we can restrict our search for the $\bullet$ at the ith iteration to sets that satisfy these conditions as additional requirements.

$(D_i, \overline{D_i})$ is a bond.
Each connected component of T_{i-1} is contained in D_i or $\overline{D_i}$ and there is no edge between D_i and components of T_{i-1} in $\overline{D_i}$.

Let $G_i = (V_i, E_i)$ be the graph obtained by shrinking each connected component of T_{i-1} into a single vertex, removing the self-loops formed and replacing each set of parallel edges by one edge of cost equal to the sum of the cost of the edges in the set. For finding a $\bullet$ at the ith iteration we consider only such sets, S, such that no connected component of T_{i-1} is split across $(S, \overline{S})$ and $(S, \overline{S})$ is a bond. Therefore, we need to look only at subsets of V_i that correspond to bonds in G_i.

Let S be a subset of vertices in G_i. The *trapped cost of S with respect to T_{i-1}*, denoted by $\text{trapped-cost}_{T_{i-1}}(S)$, is the sum of the costs of the components of T_{i-1} that are contained in S. We now redefine the *net-cost of S with respect to T_{i-1}* as

$$\text{net-cost}_{T_{i-1}}(S) = \text{cost}(S) - \text{trapped-cost}_{T_{i-1}}(S).$$

Note that for any subset of vertices in G_i, the net-cost under this new definition is at least as large as that under the previous definition. However, and this is crucial, the net-cost of the $\bullet$ set D_i remains unchanged. This is so because by Lemma 5.3 there are no edges between D_i and the components of T_{i-1} not in D_i. Therefore, a $\bullet$ under this new definition of net-cost will also be a $\bullet$ under the previous definition, and so our analysis of the dot-box algorithm continues to hold.

6. Onto planar graphs.

We do not know the complexity of computing $\bullet$ sets; we suspect that it is **NP**-hard even in planar graphs. Yet, we can implement the dot-box algorithm for planar graphs—using properties of cuts in planar graphs, and by showing that if in any iteration, the algorithm does not find the • set, then in fact, the separator found (using the □ set found in this iteration) is within twice the optimal. (This is proven in Theorem 7.1.)

6.1. Associating cycles with sets. We let G^D be the planar dual of G and fix an embedding of G^D.

PROPOSITION 6.1. *There is a one-to-one correspondence between bonds in* G *and simple cycles in* G^D.

Proof. Let $(S, \overline{S})$ be a bond in G. Since S is connected, the faces corresponding to S in G^D are adjacent, so the edges of G^D corresponding to $(S, \overline{S})$ form a simple cycle.

For the converse, let C be a simple cycle in G^D which corresponds to the cut $(S, \overline{S})$ in G. Let u, v be two vertices in G that are on the same side of the cut $(S, \overline{S})$. To prove that $(S, \overline{S})$ is a bond it suffices to show a path between u and v in G that does not use any edge of $(S, \overline{S})$.

Embed G and G^D in $\mathbf{R} \times \mathbf{R}$, and consider the two faces of G^D corresponding to vertices u and v. Pick an arbitrary point in each face, for instance, the points corresponding to u and v. Since C is a simple cycle in G^D (and hence in $\mathbf{R} \times \mathbf{R}$) there is a continuous curve (in $\mathbf{R} \times \mathbf{R}$) that connects the two points without intersecting C. By considering the faces of G^D that this curve visits, and the edges of G^D that the curve intersects, we obtain a path in G that connects vertices u, v without using any edge of $(S, \overline{S})$. □

Since for finding a • and □ we need to consider only sets, S, such that $(S, \overline{S})$ is a bond, we can restrict ourselves to simple cycles in G^D. Furthermore, the two orientations of a simple cycle can be used to distinguish between the two sides of the cut to which this cycle corresponds. The notation we adopt is the following: with a cycle C directed clockwise we associate the set of faces in G^D (and hence vertices in G) *enclosed* by C. (The side that does not include the infinite face is said to be enclosed by C and the side containing the infinite face is said to be *outside* C.)

Let $\vec{G^D}$ be the graph obtained from G^D by replacing each undirected edge (u, v) by two directed edges $(u \to v)$ and $(v \to u)$. By the preceding discussion, there exists a correspondence between sets of vertices, S, in G such that $(S, \overline{S})$ is a bond and directed simple cycles in $\vec{G^D}$.

6.2. Transfer function. We associate a cost function, c, with the edges of $\vec{G^D}$ in the obvious manner; an edge in $\vec{G^D}$ is assigned the same cost as the corresponding dual edge in G. Thus, for a directed cycle, C, $c(C)$, denotes the sum of the costs of the edges along the cycle. We would also like to associate functions, t_i, w_i with the edges of $\vec{G^D}$ so that if S is the set corresponding to a directed simple cycle C, then $t_i(C) = \text{trapped-cost}_{T_{i-1}}(S)$ and $w_i(C) = \text{net-weight}_{T_{i-1}}(S)$. We achieve this by means of a *transfer function*.

The notion of a transfer function was introduced by Park and Phillips [7] and can be viewed as an extension of a function given by Kasteleyn [2]. A function g defined on the edges of $\vec{G^D}$ is *antisymmetric* if $g(u \to v) = -g(v \to u)$. (Notice that the function c defined above is *symmetric*.) Let $f : V \to \mathbf{R}$ be a function on the vertices of G. The transfer function corresponding to f is an antisymmetric function, f_t, on the edges of $\vec{G^D}$ such that the sum of the values that f_t takes on the edges of any clockwise (anticlockwise) simple cycle in $\vec{G^D}$ is equal to the (negative of the) sum of the values that f takes on the vertices corresponding to the faces enclosed by this cycle.

That a transfer function exists for every function defined on the vertices of G and that it can be computed efficiently from the following simple argument. Pick a spanning tree in G^D, and set f_t to zero for the corresponding edges in $\vec{G^D}$. Now, add the remaining edges of G^D in an order so that with each edge added, one face of this graph is completed. Note that before the edge e is added, all other edges of the face that e completes have been assigned a value under f_t. One of the two directed edges corresponding to e is used in the clockwise traversal of this face, and the other in the anticlockwise traversal. Since the value of f for this face is known and since f_t should sum to this value (the negative of this value) in a clockwise (anticlockwise) traversal of this face, the value of f_t for the two directed edges corresponding to e can be determined. Note that the function f_t obtained in this manner is antisymmetric, and this together with the fact that the edges of any simple cycle in G^D can be written as a $GF[2]$ sum of the edges belonging to the faces contained in the cycle implies that f_t has the desired property.

7. Finding • sets.

Recall that a • at the ith iteration is a bond in the graph $G_i = (V_i, E_i)$; hence we can restrict our search for a • at the ith iteration to directed simple cycles in $\vec{G_i^D}$.

7.1. Obtaining net-weight and net-cost from transfer functions.

Let $\tilde{w}_i : V_i \to \mathbf{Z}^+$, $\tilde{t}_i : V_i \to \mathbf{R}^+$ be two functions defined on the vertices of G_i as follows. The vertices in V_i obtained by shrinking connected components of T_{i-1} have $\tilde{w}_i = 0$ and $\tilde{t}_i$ equal to the cost of the corresponding component of T_{i-1}. The remaining vertices have $\tilde{w}_i = wt$ and $\tilde{t}_i = 0$. Let w_i, t_i denote the transfer functions corresponding to functions $\tilde{w}_i, \tilde{t}_i$. We now relate the values of the functions c, w_i, t_i on a directed simple cycle to the net-cost, net-weight, and net-sparsity of the set corresponding to the cycle.

Let C be a directed simple cycle in $\vec{G_i^D}$ and $S \subset V$ the set corresponding to it. If C is clockwise then the net-weight and trapped cost of S are given by the values of the transfer functions on C, i.e.,

$$\text{net-weight}_{T_{i-1}}(S) = w_i(C),$$
$$\text{trapped-cost}_{T_{i-1}}(S) = t_i(C).$$

If C is anticlockwise then the values of the transfer functions w_i, t_i on C equal the negative of the net-weight and the trapped cost of the set enclosed by C (which is $\overline{S}$ in our notation). Hence

$$\text{net-weight}_{T_{i-1}}(S) = W - \text{wt}(T_{i-1}) - \text{net-weight}_{T_{i-1}}(\overline{S}) = W - \text{wt}(T_{i-1}) + w_i(C),$$

$$\text{trapped-cost}_{T_{i-1}}(S) = \text{cost}(T_{i-1}) - \text{trapped-cost}_{T_{i-1}}(\overline{S}) = \text{cost}(T_{i-1}) + t_i(C).$$

Recalling our new definition of net-cost,

$$\text{net-cost}_{T_{i-1}}(S) = \text{cost}(S) - \text{trapped-cost}_{T_{i-1}}(S).$$

We conclude that if C is clockwise

$$\text{net-sparsity}_{T_{i-1}}(S) = \frac{c(C) - t_i(C)}{w_i(C)},$$

and for anticlockwise C,

$$\text{net-sparsity}_{T_{i-1}}(S) = \frac{c(C) - (t_i(C) + \text{cost}(T_{i-1}))}{w_i(C) + W - \text{wt}(T_{i-1})}.$$

Hence, for a simple directed cycle C, once we know the values of the transfer functions w_i, t_i it is easy to determine the net-weight and net-sparsity of the corresponding set S. Note that the orientation of C can be determined by the sign of $w_i(C)$ since $w_i(C) > 0$ ($w_i(C) < 0$) implies that C is clockwise (anticlockwise).

7.2. The approach to finding a •. For a fixed value of $w_i(C)$, net-sparsity$_{T_{i-1}}(S)$ is minimized when $c(C) - t_i(C)$ is minimum. This suggests the following approach for finding a •: For each w in the range $(0 \le w \le \frac{W}{3} - \text{wt}(T_{i-1})) \cup (-W + \text{wt}(T_{i-1}) \le w \le \frac{-2W}{3})$, compute **min-cycle($w$)**, a directed simple cycle with minimum $c(C) - t_i(C)$ among all directed cycles C with $w_i(C) = w$. Find the net-sparsity of the set corresponding to each of these cycles. The set with the minimum net-sparsity is the • for this iteration.

However, we can implement only a weaker version of procedure **min-cycle**. Following [7], we construct a graph H_i whose vertices are 2-tuples of the kind (v, j) where v is a vertex in $\vec{G_i^D}$ and j is an integer between $-nW$ and nW. For an edge $e = u \to v$ in $\vec{G_i^D}$ we have, for all possible choices of j, edge $(u, j) \to (v, j + w_i(e))$ of length $c(e) - t_i(e)$. The shortest path between $(v, 0)$ and (v, w) in H_i gives the shortest cycle among all directed cycles in $\vec{G_i^D}$ which contain v and for which $w_i(C) = w$. By doing this computation for all choices of v, we can find the shortest cycle with $w_i(C) = w$.

Two questions arise.

1. Is H_i free of negative cycles? This is essential for computing shortest paths efficiently.
2. Is the cycle obtained in $\vec{G_i^D}$ simple?

The answer to both questions is "no." Interestingly enough, things still work out. We will tackle first the second question (in Theorem 7.1) and then the first (in Lemma 7.2).

7.3. Overcoming nonsimple cycles. Before we discuss how to get over this problem, we need to have a better understanding of the structure of a nonsimple cycle, C. If C is not a simple cycle in $\vec{G_i^D}$, decompose it arbitrarily into a collection of edge-disjoint directed simple cycles, $\mathcal{C}$. Let $(S_j, \overline{S_j})$ be the cut (in G_i) corresponding to a cycle $C_j \in \mathcal{C}$ and S_j be the side of the cut that has smaller net-weight. Further, let $\mathcal{S}$ be the collection of sets S_j, one for each $C_j \in \mathcal{C}$.

The value of the transfer functions w_i, t_i over C is the sum of their values over the cycles C_j in the collection $\mathcal{C}$. Also,

$$c(C) = \sum_{S_j \in \mathcal{S}} \text{cost}(S_j).$$

For each cycle $C_j \in \mathcal{C}$ we need to relate the net-weight, trapped cost of S_j to the value of the transfer functions w_i, t_i on C_j. C_j might be either clockwise or anticlockwise. Further, S_j might be either inside C_j or outside C_j. This gives us a total of four different cases. The relationship between net-weight$_{T_{i-1}}(S_j)$ and $w_i(C_j)$, and trapped-cost$_{T_{i-1}}(S_j)$ and $t_i(C_j)$ is given in Figure 7.1 and can be captured succinctly

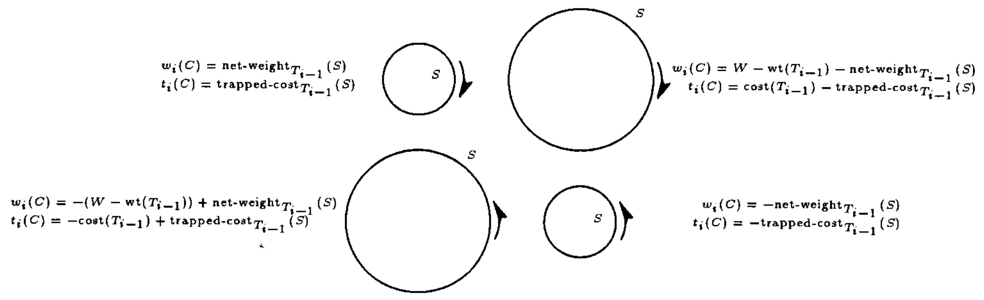

$w_i(C) = \text{net-weight}_{T_{i-1}}(S)$
$t_i(C) = \text{trapped-cost}_{T_{i-1}}(S)$
S
S
$w_i(C) = W - \text{wt}(T_{i-1}) - \text{net-weight}_{T_{i-1}}(S)$
$t_i(C) = \text{cost}(T_{i-1}) - \text{trapped-cost}_{T_{i-1}}(S)$
S
$w_i(C) = -(W - \text{wt}(T_{i-1})) + \text{net-weight}_{T_{i-1}}(S)$
$t_i(C) = -\text{cost}(T_{i-1}) + \text{trapped-cost}_{T_{i-1}}(S)$
S
$w_i(C) = -\text{net-weight}_{T_{i-1}}(S)$
$t_i(C) = -\text{trapped-cost}_{T_{i-1}}(S)$

FIG. 7.1. *Relationship between* net-weight$_{T_{i-1}}(S)$ *and* $w_i(C)$ *for the four cases.*

as

$$w_i(C_j) = x_j(W - \text{wt}(T_{i-1})) + y_j \cdot \text{net-weight}_{T_{i-1}}(S_j),$$
$$t_i(C_j) = x_j \cdot \text{cost}(T_{i-1}) + y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j),$$

where $x_j \in \{+1, 0, -1\}$ and $y_j \in \{+1, -1\}$.

Hence we get a *decomposition rule* relating the value of the functions w_i, t_i on a nonsimple cycle C to the net-weight and trapped cost of the sets induced by this cycle:

$$w_i(C) = x(W - \text{wt}(T_{i-1})) + \sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j),$$
$$t_i(C) = x \cdot \text{cost}(T_{i-1}) + \sum_{S_j \in \mathcal{S}} y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j),$$

where $x = \sum_j x_j$ is an integer.

7.4. A key theorem. Let D_i be a $\bullet$ at the ith iteration ($i \le k-1$) and C^* be the directed simple cycle in $\vec{G}_i^D$ corresponding to it. Further, let C be the directed cycle reported by **min-cycle**$(w_i(C^*))$.

THEOREM 7.1. *If C is not simple then the separator found in this iteration has cost at most* $2 \cdot \text{cost(OPT)}$*, i.e.,*

$$\text{cost}(T_{i-1} \cup B_i) \le 2 \cdot \text{cost(OPT)}.$$

Proof. Since C is the directed cycle for which $c(C) - t_i(C)$ is minimum among all cycles with $w_i(C) = w_i(C^*)$ we can claim the following.

If C^* is clockwise, i.e., $w_i(C^*) > 0$, then

$$w_i(C) = \text{net-weight}_{T_{i-1}}(D_i),$$
$$c(C) - t_i(C) \le \text{net-cost}_{T_{i-1}}(D_i),$$

and if C^* is anticlockwise, i.e., $w_i(C^*) < 0$, then

$$W - \text{wt}(T_{i-1}) + w_i(C) = \text{net-weight}_{T_{i-1}}(D_i),$$
$$c(C) - t_i(C) - \text{cost}(T_{i-1}) \le \text{net-cost}_{T_{i-1}}(D_i).$$

Substituting for $w_i(C)$ and $t_i(C)$ by the decomposition rule we get

$$z(W - \mathrm{wt}(T_{i-1})) + \sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j) = \text{net-weight}_{T_{i-1}}(D_i), \tag{7.1}$$

$$-z \cdot \mathrm{cost}(T_{i-1}) + \sum_{S_j \in \mathcal{S}} (\mathrm{cost}(S_j) - y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j)) \leq \text{net-cost}_{T_{i-1}}(D_i), \tag{7.2}$$

where z is x if C^* is clockwise and $x + 1$ if C^* is anticlockwise.

We now prove that there exists $S_j \in \mathcal{S}$ which meets the weight requirement for a $\Box$ and has cost no more than the cost of T_i, i.e.,

1. $\frac{W}{3} \leq \mathrm{wt}(T_{i-1} \cup S_j) \leq \frac{2W}{3}$,
2. $\mathrm{cost}(S_j) \leq \mathrm{cost}(T_{i-1}) + \text{net-cost}_{T_{i-1}}(D_i)$.

Assume for contradiction that no such S_j exists. The following observations about the cost/net-cost of a set $S_j \in \mathcal{S}$ are immediate.

OBSERVATION 7.1. *If* $\text{net-weight}_{T_{i-1}}(S_j) \geq W/3 - \mathrm{wt}(T_{i-1})$ *then*

$$\mathrm{cost}(S_j) > \mathrm{cost}(T_{i-1}) + \text{net-cost}_{T_{i-1}}(D_i),$$

which implies

$$\text{net-cost}_{T_{i-1}}(S_j) > \text{net-cost}_{T_{i-1}}(D_i).$$

OBSERVATION 7.2. *If* $\text{net-weight}_{T_{i-1}}(S_j) < W/3 - \mathrm{wt}(T_{i-1})$ *then*

$$\text{net-sparsity}_{T_{i-1}}(S_j) \geq \text{net-sparsity}_{T_{i-1}}(D_i)$$

and hence

$$\text{net-cost}_{T_{i-1}}(S_j) \geq \text{net-sparsity}_{T_{i-1}}(D_i) \cdot \text{net-weight}_{T_{i-1}}(S_j).$$

The next observation follows from the above two observations.

OBSERVATION 7.3. *All sets* $S_j \in \mathcal{S}$ *have nonnegative net-cost, i.e.,* $\text{net-cost}_{T_{i-1}}(S_j) \geq 0$.

The idea behind obtaining a contradiction is as follows. For every integral choice of z we use (7.1) to provide a lower bound on the total net-weight of the sets $S_j \in \mathcal{S}$ and (7.2) to provide an upper bound on the total net-cost of the sets $S_j \in \mathcal{S}$. We then use the above observations on the cost/net-cost of sets $S_j \in S$ to argue that there is no way of having sets with so large a total net-weight at so little a total net-cost.

We shall consider three cases depending upon whether z is positive, negative, or zero.

$z = 0$. Equation (7.2) implies

$$\begin{aligned}\text{net-cost}_{T_{i-1}}(D_i) &\geq \sum_{S_j \in \mathcal{S}} (\mathrm{cost}(S_j) - y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j)) \\ &\geq \sum_{S_j \in \mathcal{S}} \text{net-cost}_{T_{i-1}}(S_j),\end{aligned}$$

and from (7.1) we have

$$\begin{aligned}\sum_{S_j \in \mathcal{S}} \text{net-weight}_{T_{i-1}}(S_j) &\geq \sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j) \\ &= \text{net-weight}_{T_{i-1}}(D_i).\end{aligned}$$

Since the net-cost of each set is nonnegative (Observation 7.3) each set in $\mathcal{S}$ has net-cost no more than net-cost$_{T_{i-1}}(D_i)$. This in turn implies that every set in $\mathcal{S}$ has net-weight strictly less than $W/3 - \text{wt}(T_{i-1})$ (Observation 7.1). Thus every set in $\mathcal{S}$ meets the weight requirement for a $\bullet$. Since the net-cost of every set in $\mathcal{S}$ is nonnegative, Remark 4.1 applied to the above two inequalities implies that either there exists $S_j \in \mathcal{S}$ of lower net-sparsity than D_i or every set in $\mathcal{S}$ has the same net-sparsity as D_i and the sum of the net-weight of the sets in $\mathcal{S}$ is equal to the net-weight of D_i. The first setting leads to a contradiction since every set in $\mathcal{S}$ satisfies the weight requirement for a $\bullet$ and D_i is the $\bullet$ at this iteration. The second setting in turn contradicts the minimality requirement on D_i.

$z > 0$. Let $\mathcal{S}^-$ denote the collection of sets $S_j \in \mathcal{S}$ with $y_j = -1$. Equation (7.2) now yields

$$\begin{aligned}\text{net-cost}_{T_{i-1}}(D_i) + z \cdot \text{cost}(T_{i-1}) &\geq \sum_{S_j \in \mathcal{S}} (\text{cost}(S_j) - y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j)) \\ &\geq \sum_{S_j \in \mathcal{S}^-} (\text{cost}(S_j) + \text{trapped-cost}_{T_{i-1}}(S_j)) \\ &\geq \sum_{S_j \in \mathcal{S}^-} \text{cost}(S_j),\end{aligned}$$

where the second inequality follows from the fact that all sets in $\mathcal{S} - \mathcal{S}^-$ have nonnegative net-cost. We shall develop a contradiction by showing that the costs of the sets in $\mathcal{S}^-$ is more than the left-hand side of the above inequality. A lower bound on the total net-weight of the sets in $\mathcal{S}^-$ can be obtained from (7.1) as follows:

$$\begin{aligned}z(W - \text{wt}(T_{i-1})) - \text{net-weight}_{T_{i-1}}(D_i) &= -\sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j) \\ &\leq \sum_{S_j \in \mathcal{S}^-} \text{net-weight}_{T_{i-1}}(S_j).\end{aligned}$$

What is the cheapest way of picking sets so that their net-weight is at least $z(W - \text{wt}(T_{i-1})) - \text{net-weight}_{T_{i-1}}(D_i)$? By Observation 7.2 a set S_j such that net-weight$_{T_{i-1}}(S_j) < W/3 - \text{wt}(T_{i-1})$ can be picked only at a net-sparsity of at least net-sparsity$_{T_{i-1}}(D_i)$. On the other hand Observation 7.1 says that we could be picking sets with large net-weight for cost little more than $\text{cost}(T_{i-1}) + \text{net-cost}_{T_{i-1}}(D_i)$. Since any set in $\mathcal{S}$ has net-weight at most $\frac{W - \text{wt}(T_{i-1})}{2}$ the unit cost of picking these large sets could be as small as

$$\begin{aligned}\frac{\text{cost}(T_{i-1}) + \text{net-cost}_{T_{i-1}}(D_i)}{\frac{W - \text{wt}(T_{i-1})}{2}} &< \frac{\text{cost}(T_{i-1}) + \text{net-cost}_{T_{i-1}}(D_i)}{\text{wt}(T_{i-1}) + \text{net-weight}_{T_{i-1}}(D_i)} \\ &\leq \max\left\{\frac{\text{cost}(T_{i-1})}{\text{wt}(T_{i-1})}, \frac{\text{net-cost}_{T_{i-1}}(D_i)}{\text{net-weight}_{T_{i-1}}(D_i)}\right\} \\ &\leq \text{net-sparsity}_{T_{i-1}}(D_i),\end{aligned}$$

where the last inequality follows from the fact that $\text{sparsity}(T_{i-1}) \leq$ net-sparsity$_{T_{i-1}}(D_i)$ which in turn is a consequence of Lemma 4.1.

Thus the cheapest possible way of picking sets is to pick sets of net-weight $\frac{W-\mathrm{wt}(T_{i-1})}{2}$, incurring a cost of little more than $\mathrm{cost}(T_{i-1})+\text{net-cost}_{T_{i-1}}(D_i)$ for each set picked. Since we need to pick a net-weight of at least $z(W-\mathrm{wt}(T_{i-1}))-\text{net-weight}_{T_{i-1}}(D_i)$, we would have to pick at least $2z-1$ such sets and so the cost incurred is at least

$$\begin{aligned}\sum_{S_j\in\mathcal{S}^-}\mathrm{cost}(S_j) &> (2z-1)(\mathrm{cost}(T_{i-1})+\text{net-cost}_{T_{i-1}}(D_i))\\ &\geq z\cdot\mathrm{cost}(T_{i-1})+\text{net-cost}_{T_{i-1}}(D_i),\end{aligned}$$

where the last inequality follows from the fact that $z\geq 1$ (hence $2z-1\geq z$ and $2z-1\geq 1$). This, however, contradicts the upper bound on the sum of the costs of the sets in $\mathcal{S}^-$, which we derived at the beginning of this case.

$z<0$. Let $\mathcal{S}^+$ denote the collection of sets $S_j\in\mathcal{S}$ with $y_j=1$. Equation (7.2) now yields

$$\begin{aligned}\text{net-cost}_{T_{i-1}}(D_i) &\geq \sum_{S_j\in\mathcal{S}}(\mathrm{cost}(S_j)-y_j\cdot\text{trapped-cost}_{T_{i-1}}(S_j))\\ &\geq \sum_{S_j\in\mathcal{S}^+}\text{net-cost}_{T_{i-1}}(S_j).\end{aligned}$$

The total net-weight of the sets in $\mathcal{S}^+$ can be bounded using (7.1) as follows:

$$\begin{aligned}\sum_{S_j\in\mathcal{S}^+}\text{net-weight}_{T_{i-1}}(S_j) &\geq \sum_{S_j\in\mathcal{S}} y_j\cdot\text{net-weight}_{T_{i-1}}(S_j)\\ &= \text{net-weight}_{T_{i-1}}(D_i)-z(W-\mathrm{wt}(T_{i-1}))\\ &\geq -z(W-\mathrm{wt}(T_{i-1})),\end{aligned}$$

where the last inequality follows from the fact that the net-weight of D_i is nonnegative.

What is the cheapest way of picking sets so that their net-weight is at least $-z(W-\mathrm{wt}(T_{i-1}))$? Once again by Observation 7.2 a set S_j of net-weight less than $\frac{W}{3}-\mathrm{wt}(T_{i-1})$ can be picked only at a net-sparsity of at least $\text{net-sparsity}_{T_{i-1}}(D_i)$. On the other hand Observation 7.1 says that we could be picking a set of net-weight as large as $(W-\mathrm{wt}(T_{i-1}))/2$ for a net-cost that is only strictly larger than $\text{net-cost}_{T_{i-1}}(D_i)$. Since

$$\begin{aligned}\frac{\text{net-cost}_{T_{i-1}}(D_i)}{(W-\mathrm{wt}(T_{i-1}))/2} &< \frac{\text{net-cost}_{T_{i-1}}(D_i)}{\text{net-weight}_{T_{i-1}}(D_i)}\\ &= \text{net-sparsity}_{T_{i-1}}(D_i),\end{aligned}$$

the cheapest possible way of picking sets is to pick sets of net-weight $\frac{W-\mathrm{wt}(T_{i-1})}{2}$ and incur a net-cost strictly larger than $\text{net-cost}_{T_{i-1}}(D_i)$ for each set picked. Since we need to pick a net-weight of at least $-z(W-\mathrm{wt}(T_{i-1}))$, we should pick at least $-2z$ such sets. Since $z\leq -1$, the total net-cost of these sets is strictly larger than $\text{net-cost}_{T_{i-1}}(D_i)$ contradicting the upper bound derived at the beginning of this case.

We have thus established that there exists a set $S_j\in\mathcal{S}$ which meets the weight requirement for a $\square$ and has cost no more than $\mathrm{cost}(T_i)$. Further, S_j corresponds to a directed simple cycle in $\vec{G_i^D}$. Our procedure for finding a □ returns a set of cost less than the cost of any set that meets the weight requirement for a □ and corresponds to a directed simple cycle in $\vec{G^D}$. Hence

$$\text{cost}(B_i) \leq \text{cost}(S_j) \leq \text{cost}(T_i).$$

Therefore,

$$\text{cost}(T_{i-1} \cup B_i) \leq \text{cost}(T_{i-1}) + \text{cost}(B_i) \leq \text{cost}(T_{i-1}) + \text{cost}(T_i) \leq 2 \cdot \text{cost(OPT)},$$

where the last inequality follows from Lemma 4.3 and the fact that $i \leq k-1$. ∎

For each w in the range $[0..\frac{W}{3} - \text{wt}(T_{i-1})] \cup [-W + \text{wt}(T_{i-1})..\frac{-2W}{3}]$, it suffices to find in G_i a directed cycle (not necessarily simple) with minimum $c(C) - t_i(C)$ among all directed cycles C with $w_i(C) = w$. If for some w the shortest cycle is not simple we discard the cycle and do not consider that w for the purpose of computing the •. If in the process we discard the cycle with $w_i(C) = w_i(C^*)$ then by the above theorem the separator found in this iteration is within twice the optimum. Otherwise, we obtain a simple cycle C with $w_i(C) = w_i(C^*)$ and the set corresponding to this cycle is a •.

Finally, we have to deal with the case that there are negative cycles in H_i. A negative cycle in H_i corresponds to a cycle C in $\vec{G_i^D}$ such that $w_i(C) = 0$ and $c(C) - t_i(C) < 0$.

LEMMA 7.2. *If C is a cycle in $\vec{G_i^D}$ such that $w_i(C) = 0$ and $c(C) - t_i(C) < 0$ then the separator found in this iteration has cost at most $2 \cdot \text{cost(OPT)}$.*

Proof. The proof of this lemma is along the lines of Theorem 7.1. We decompose C into a collection $\mathcal{C}$ of directed simple cycles. For $C_j \in \mathcal{C}$ let S_j be the side of the cycle with smaller net-weight and let $\mathcal{S}$ be the collection of sets S_j, one for each $C_j \in \mathcal{C}$. Using the decomposition rule, we have

$$z(W - \text{wt}(T_{i-1})) + \sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j) = 0, \tag{7.3}$$

$$-z \cdot \text{cost}(T_{i-1}) + \sum_{S_j \in \mathcal{S}} (\text{cost}(S_j) - y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j)) < 0. \tag{7.4}$$

For contradiction we assume that every $S_j \in \mathcal{S}$ which satisfies the weight requirement for a □ has cost more than $\text{cost}(T_i)$. By Observation 7.3 every set $S_j \in \mathcal{S}$ has nonnegative net-cost. Hence (7.4) yields

$$\begin{aligned} z \cdot \text{cost}(T_{i-1}) &> \sum_{S_j \in \mathcal{S}} (\text{cost}(S_j) - y_j \cdot \text{trapped-cost}_{T_{i-1}}(S_j)) \\ &\geq \sum_{S_j \in \mathcal{S}^-} (\text{cost}(S_j) + \text{trapped-cost}_{T_{i-1}}(S_j)) \\ &\geq \sum_{S_j \in \mathcal{S}^-} \text{cost}(S_j), \end{aligned}$$

which implies that $z > 0$.

A lower bound on the total net-weight of sets in $\mathcal{S}^-$ can be obtained using (7.3):

$$\begin{aligned} z(W - \text{wt}(T_{i-1})) &= -\sum_{S_j \in \mathcal{S}} y_j \cdot \text{net-weight}_{T_{i-1}}(S_j) \\ &\leq \sum_{S_j \in \mathcal{S}^-} \text{net-weight}_{T_{i-1}}(S_j). \end{aligned}$$

Beyond this point the argument is almost identical to that for the case when $z > 0$ in the proof of Theorem 7.1. This contradicts our assumption that every set $S_j \in \mathcal{S}$ which meets the weight requirement of a $\square$ has cost more than $\text{cost}(T_i)$. As in the proof of Theorem 7.1, the $\square$ picked in this iteration has cost at most $\text{cost}(T_i)$ and hence the cost of the separator output is at most $2 \cdot \text{cost}(\text{OPT})$. $\square$

By Lemma 7.2, we need to compute shortest paths in graph H_i only if it has no negative cycles.

8. Finding $\square$ sets. We will use Rao's algorithm [8, 9] to find a $\square$ set. Let $w : V \to Z^+$ be a weight function on the vertices of G such that $w(v) = 0$ if $v \in T_{i-1}$ and $wt(v)$ otherwise. If B_i is a $\square$ in the ith iteration, then $(B_i, \overline{B_i})$ is a b-balanced bond in G when the weights on the vertices are given by w and $b = \frac{W/3 - \text{wt}(T_{i-1})}{W - \text{wt}(T_{i-1})}$. Thus, to find the $\square$ we need to find the minimum-cost simple cycle in G^D which corresponds to a b-balanced bond in G.

Rao [8, 9] gives an algorithm for finding a minimum-cost b-balanced *connected circuit* in G^D. A connected circuit in G^D is a set of cycles in G^D connected by an acyclic set of paths. Intuitively, a connected circuit can be viewed as a simple cycle with "pinched" portions corresponding to the paths. The cost of a connected circuit is defined to be the cost of the closed walk that goes through each pinched portion twice and each cycle once. A connected circuit in G^D defines a simple cut in G; the vertices corresponding to faces included in the cycles of the connected circuit form one side of the cut. A connected circuit is b-balanced if the cut corresponding to it is b-balanced. Note that the cost of the cut defined by a connected circuit is just the sum of the costs of the cycles in it. Hence, the definition of cost of a connected circuit is an upper bound on the cost of the underlying cut; the two are equal if the connected circuit is a simple cycle.

Notice that for a $\square$ we do not really need to find a minimum-cost b-balanced bond in G; any cut that is b-balanced and has cost no more than the minimum-cost b-balanced bond will serve our purpose. Hence we can use Rao's algorithm to find a $\square$. The total time taken by Rao's algorithm to obtain an optimal b-balanced connected circuit cut is $O(n^2 W)$.

9. Running time. Clearly, the algorithm terminates in at most n iterations. The running time of each iteration is dominated by the time to find a $\bullet$. In each iteration, computing a $\bullet$ involves $O(n)$ single source shortest path computations in a graph with $O(n^2 W)$ vertices and $O(n^2 W)$ edges; the edge-lengths may be negative. This can be done in $O(n^4 W^2)$ time by preprocessing the network so that all edge-lengths are negative and then applying Dijkstra's algorithm n times. Hence, the total running time of the dot-box algorithm is $O(n^5 W^2)$. This is polynomial if W is polynomially bounded.

THEOREM 9.1. *The dot-box algorithm finds an edge-separator in a planar graph of cost within twice the optimum and runs in time $O(n^5 W^2)$, where W is the sum of weights of the vertices.*

10. Dealing with binary weights. The size of the graph in which we compute shortest paths (and hence the running time of the dot-box algorithm) depends on the sum of the vertex weights. Using scaling we can make our algorithm strongly polynomial; however, the resulting algorithm is a pseudoapproximation algorithm in the sense that it compares the cut obtained with an optimal cut having a better balance. Finally, we use our scaling ideas to extend the algorithm of Park and Phillips into a fully polynomial approximation scheme for finding sparsest cuts in planar graphs with vertex weights given in binary, thereby settling their open problem.

10.1. *b*-balanced cut. Let us scale the vertex weights so that the sum of the weights is no more than αn $(\alpha > 1)$. This can be done by defining a new weight function $\hat{wt} : V \rightarrow \mathbf{Z}^+$ as

$$\hat{wt}(v) = \left\lfloor \frac{\mathrm{wt}(v)}{W} \cdot \alpha n \right\rfloor .$$

The process of obtaining the new weights can be viewed as a two-step process: first we scale the weights by a constant factor $\frac{\alpha n}{W}$ and then we truncate. The first step does not affect the balance of any cut since all vertex weights are scaled by the same factor. However, the second step could affect the balance of a cut. Thus a cut $(S, \overline{S})$ with balance b under the weight function wt might have a worse balance under $\hat{wt}$ since all vertices on the side with smaller weight might have their weights truncated. However, the total loss in weight due to truncations is at most n (1 for each vertex). The balance would be worst when the total weight stays at αn (not drop by the truncations) and then the loss of weight of the smaller side is a $1/\alpha$ fraction of the total weight. Thus the balance of the cut $(S, \overline{S})$ under $\hat{wt}$ might be $b - 1/\alpha$ but no worse.

Similarly, a cut $(S, \overline{S})$ with balance $\hat{b}$ under $\hat{wt}$ might have a worse balance under wt. It is easy to show by a similar argument that under wt, $(S, \overline{S})$ has a balance no worse than $\hat{b} - 1/\alpha$.

Let OPT denote the cost of the optimum b-balanced cut under the weight assignment wt. Since this cut might be $(b - 1/\alpha)$-balanced under $\hat{wt}$, we use the dot-box algorithm to find a $(b - 1/\alpha)$-balanced cut of cost within 2OPT. The cut returned by our algorithm, while being $(b - 1/\alpha)$-balanced under $\hat{wt}$, might only be $(b - 2/\alpha)$-balanced under wt. Thus we obtain a $(b - 2/\alpha)$-balanced cut of cost within twice the optimum b-balanced cut.

THEOREM 10.1. *For $\alpha > 2/b$, the dot-box algorithm, with weight scaling, finds a $(b - 2/\alpha)$-balanced cut in a planar graph of cost within twice the cost of an optimum b-balanced cut for $b \leq \frac{1}{3}$ in $O(\alpha^2 n^7)$ time.*

10.2. Sparsest cut. Assume that vertex weights in planar graph G are given in binary. Let 2^p be the least power of 2 that bounds the weight of each vertex and W be the sum of weights of all vertices. We will construct $p + 1$ copies of G, $G_i, 0 \leq i \leq p$, each having the same edge costs as G. In G_i, vertex weights are assigned as follows: Let α be a positive integer; α determines the approximation guarantee as described below. Vertices having weights in the range $[2^i, 2^{i+2\log n+\alpha+2}]$ are assigned their original weight; those having weight $< 2^i$ are assigned weight 0, i.e., they can be deleted from the graph; and those having weight $> 2^{i+2\log n+\alpha+2}$ are assigned weight $2^{i+2\log n+\alpha+2}$. The sparsest cut is computed in each of these graphs using the algorithm of Park and Phillips. For the purpose of this computation, the weights of all vertices in G_i are divided by 2^i; notice that this leaves the weights integral, and the total weight of vertices is at most $O(2^\alpha n^3)$. The running time of [7] is $O(n^2 w \log nw)$, where w is the total weight of vertices in the graph. Thus, this computation takes time $O(\alpha 2^\alpha n^5 \log W \log n)$, which is polynomial in the size of the input, for fixed α. The sparsity of the $p + 1$ cuts so obtained is computed in the original graph, and the sparsest one is chosen.

Let $(S, \overline{S})$ be an optimal sparsest cut in G, and let S be its lighter side. Let the weight of S be t, and let q be the weight of the heaviest vertex in S. Pick the smallest integer i such that $2^{i+\log n+\alpha+1} \geq q$.

LEMMA 10.2. *The cut found in G_i has cost at most that of $(S, \overline{S})$, and weight at least $(1-\frac{1}{2^\alpha})t$. Therefore, this cut has sparsity within a factor of $\frac{1}{1-\frac{1}{2^\alpha}}$ of the sparsity of $(S, \overline{S})$.*

Proof. The algorithm of Park and Phillips searches for the cheapest cut of each weight, for all choices of weight between 0 and half the weight of the given graph. It then outputs the sparsest of these cuts.

First notice that for the choice of i given above, the weight of S in G_i, say t', satisfies $(1-\frac{1}{2^\alpha})t \leq t' \leq t$. Indeed, any set of vertices whose weights are at most $2^{i+2\log n+\alpha+2}$ in G satisfies that its weight drops by a factor of at most $(1-\frac{1}{2^\alpha})$ in G_i. On the other hand, any set of vertices containing a vertex having weight $> 2^{i+2\log n+\alpha+2}$ in G has weight exceeding t in G_i. Therefore, the cut found in G_i for the target weight of t' satisfies the conditions of the lemma. $\square$

For a given choice of $\delta > 0$, pick the smallest positive integer α so that $1+\delta \geq \frac{1}{(1-\frac{1}{2^\alpha})}$. Then we get the following.

THEOREM 10.3. *The above algorithm gives a fully polynomial time approximation scheme for the minimum-sparsity cut problem in planar graphs. For each $\delta > 0$, this algorithm finds a cut of sparsity within a factor of $(1+\delta)$ of the optimal in $O(\frac{1}{\delta}\log(\frac{1}{\delta})n^5 \log W \log n)$ time.*

11. Open problems.

Several open problems remain.

1. Is the problem of finding the cheapest b-balanced cut in planar graphs strongly **NP**-hard, or is there a pseudo–polynomial time algorithm for it?
2. What is the complexity of finding a minimum net-sparsity cut in planar graphs, assuming that the vertex weights are given in unary?
3. What is the complexity of finding $\bullet$ sets in planar graphs, assuming that the vertex weights are given in unary?
4. Can the algorithm given in this paper be extended to other submodular functions?
5. Can it be extended to other classes of graphs? In particular, can the notion of transfer function be extended to other classes of graphs?

REFERENCES

[1] S. N. BHATT AND F. T. LEIGHTON, *A framework for solving VLSI graph layout problems*, J. Comput. System Sci., 28 (1984), pp. 300–343.

[2] P. W. KASTELEYN, *Dimer statistics and phase transitions*, J. Math. Phys., 4 (1963), pp. 287–293.

[3] F. T. LEIGHTON AND S. RAO *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms*, in Proceedings, 29th Annual IEEE Symposium on Foundations of Computer Science, 1988, pp. 422–431.

[4] C. E. LEISERSON, *Area-efficient layouts (for VLSI)*, in Proceedings, 21st Annual IEEE Symposium on Foundations of Computer Science, 1980.

[5] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[6] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 615–627.

[7] J. K. PARK AND C. A. PHILLIPS, *Finding minimum-quotient cuts in planar graphs*, in Proceedings, 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 766–775.

[8] S. B. RAO, *Finding near optimal separators in planar graphs*, in Proceedings, 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 225–237.

[9] S. B. RAO, *Faster algorithms for finding small edge cuts in planar graphs*, in Proceedings, 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 229–240.

SIAM J. Comput.
Vol. 29, No. 1, pp. 180–200

BALANCED ALLOCATIONS[*]

YOSSI AZAR[†], ANDREI Z. BRODER[‡], ANNA R. KARLIN[§], AND ELI UPFAL[¶]

Abstract. Suppose that we sequentially place n balls into n boxes by putting each ball into a randomly chosen box. It is well known that when we are done, the fullest box has with high probability $(1+o(1))\ln n/\ln\ln n$ balls in it. Suppose instead that for each ball we choose two boxes at random and place the ball into the one which is less full at the time of placement. We show that with high probability, the fullest box contains only $\ln\ln n/\ln 2+O(1)$ balls—exponentially less than before. Furthermore, we show that a similar gap exists in the infinite process, where at each step one ball, chosen uniformly at random, is deleted, and one ball is added in the manner above. We discuss consequences of this and related theorems for dynamic resource allocation, hashing, and on-line load balancing.

Key words. urn models, occupancy problems, on-line algorithms, resource allocation, hashing, load balancing

AMS subject classifications. 68Q25, 68M20, 68P10, 60G99, 60G10, 90C40

PII. S0097539795288490

1. Introduction.

Suppose that we sequentially place n balls into n boxes by putting each ball into a randomly chosen box. Properties of this random allocation process have been extensively studied in the probability and statistics literature. (See, e.g., [20, 17].) One of the classical results in this area is that, asymptotically, when the process has terminated, with high probability (that is, with probability $1-o(1)$) the fullest box contains $(1+o(1))\ln n/\ln\ln n$ balls. (G. Gonnet [16] has proven a more accurate result, $\Gamma^{-1}(n)-3/2+o(1)$.)

Consider a variant of the process above whereby each ball comes with d possible destinations, chosen independently and uniformly at random. (Hence the d destinations are not necessarily distinct.) The ball is placed in the least full box among the d possible locations. Surprisingly, even for $d=2$, when the process terminates the fullest box has only $\ln\ln n/\ln 2+O(1)$ balls in it. Thus, this apparently minor change in the random allocation process results in an exponential decrease in the maximum occupancy per location. The analysis of this process is summarized as follows

THEOREM 1.1. *Suppose that m balls are sequentially placed into n boxes. Each ball is placed in the least full box, at the time of the placement, among d boxes, $d\geq 2$, chosen independently and uniformly at random. Then after all the balls are placed,*

- *with high probability, as $n\to\infty$ and $m\geq n$, the number of balls in the fullest box is $(1+o(1))\ln\ln n/\ln d+\Theta(m/n)$;*

[*]Received by the editors July 5, 1995; accepted for publication (in revised form) August 4, 1997; published electronically September 14, 1999.
http://www.siam.org/journals/sicomp/29-1/28849.html

[†]Department of Computer Science, Tel Aviv University, Tel-Aviv 69978, Israel (azar@math.tau.ac.il). This research was supported by the Israel Academy of Sciences.

[‡]Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301 (broder@src.dec.com).

[§]Department of Computer Science, University of Washington, Seattle, WA 98195 (karlin@cs.washington.edu).

[¶]IBM Almaden Research Center, San Jose, CA 95120 and Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel (eli@wisdom.weizmann.ac.il). Work at the Weizmann Institute was supported in part by the Norman D. Cohen Professorial Chair of Computer Science.

- *in particular, with high probability, as $n \to \infty$ and $m = n$, the number of balls in the fullest box is* $\ln \ln n / \ln d + \Theta(1)$;
- *any other on-line strategy that places each ball into one of d randomly chosen boxes results in stochastically more balls*[1] *in the fullest box.*

It is also interesting to study the infinite version of the random allocation process. There, at each step a ball is chosen uniformly at random and removed from the system, and a new ball appears. The new ball comes with d possible destinations, chosen independently at random, and it is placed into the least full box among these d possible destinations.

The analysis of the case $d = 1$ in this infinite stochastic process is simple since the location of any ball does not depend on the locations of other balls in the system. Thus, for $d = 1$, in the stationary distribution, with high probability the fullest box has $\Theta(\log n / \log \log n)$ balls. The analysis of the case $d \geq 2$ is significantly harder, since the locations of the current n balls might depend on the locations of balls that are no longer in the system. We prove that when $d \geq 2$, in the stationary distribution, the fullest box has $\ln \ln n / \ln d + O(1)$ balls, with high probability. Thus, the same exponential gap holds in the infinite process. Theorem 1.2 is proven in section 4.

THEOREM 1.2. *Consider the infinite process with $d \geq 2$, starting at time 0 in an arbitrary state. There is a constant c such that for any fixed $T > cn^2 \log \log n$, the fullest box at time T contains, with high probability, less than $\ln \ln n / \ln d + O(1)$ balls. Thus, in the stationary distribution, with high probability, no box contains more than $\ln \ln n / \ln d + O(1)$ balls.*

Karp, Luby, and Meyer auf der Heide [18] were the first to notice a dramatic improvement when switching from one hash function to two in the context of PRAM simulations. In fact, it is possible to use a result from [18] to derive a weaker form of our static upper bound. (For details see [7].)

A preliminary version of this paper has appeared in [7]. Subsequently, Adler et al. [1] analyzed parallel implementation of the balanced allocation mechanism and obtained interesting communication vs. load tradeoffs.

A related question was considered by Broder et al. [10]. In their model the set of choices is such that there is a placement that results in maximum load equal to one. The question they analyze is, what is the expected maximum load under a random order of insertion under the greedy strategy?

More recent results, based on the balanced allocation paradigm, have appeared in [23, 24, 25, 12].

1.1. Applications.

Our results have a number of interesting applications to computing problems. We elaborate here on three of them.

1.1.1. Dynamic resource allocation.

Consider a scenario in which a user or a process has to choose between a number of identical resources on-line (choosing a server to use among the servers in a network, choosing a disk to store a directory, etc.). To find the least loaded resource, users may check the load on all resources before placing their requests. This process is expensive, since it requires sending an interrupt to each of the resources. A second approach is to send the task to a random resource. This approach has minimum overhead, but if all users follow it, the difference in load between different servers will vary by up to a logarithmic factor. Our analysis suggests

[1]By this we mean that for any other strategy and any k the probability that the number of balls in the fullest box is greater than k is at least the probability that the number of balls in the fullest box is greater than k under the greedy strategy. See Corollary 3.6.

a more efficient solution. If each user samples the load of two resources and sends his request to the least loaded, the total overhead is small, and the load on the n resources varies by only a $O(\log\log n)$ factor.

1.1.2. Hashing. The efficiency of a hashing technique is measured by two parameters: the expected and the maximum access time. Our approach suggests a simple hashing technique, similar to hashing with chaining. We call it 2-*way chaining.* It has $O(1)$ expected and $O(\log\log n)$ maximum access time. We use two random hash functions. The two hash functions define two possible entries in the table for each key. The key is inserted to the least full location, at the time of the insertion. Keys in each entry of the table are stored in a linked list. Assume that n keys are sequentially inserted by this process into a table of size n. As shown in section 5, the expected insertion and look-up time is $O(1)$, and our analysis summarized above immediately implies that with high probability the maximum access time is $\ln\ln n/\ln 2+O(1)$, vs. the $\Theta(\log n/\log\log n)$ time when only one random hash function is used.

An advantage of our scheme over some other known techniques for reducing worst-case behavior of hashing (e.g., [14, 13, 11]) is, that it uses only two hash functions, it is easy to parallelize, and it does not involve rehashing of data. Other commonly used schemes partition the available memory into multiple tables, and use a different hash function in each table. For example, the Fredman, Komlos, Szemeredi scheme for perfect hashing [14] uses up to n different hash functions to get $O(1)$ worst-case access time (not on-line however), and the algorithm of Broder and Karlin [11] uses $O(\log\log n)$ hash functions to achieve $O(\log\log n)$ maximum access time on-line but using rehashings.

Karp, Luby, and Meyer auf der Heide [18] studied the use of two hash functions in the context of PRAM simulations. Other PRAM simulations using multiple hash functions were developed and analyzed in [21].

1.1.3. Competitive on-line load balancing. Consider the following on-line load balancing problem: We are given a set of n servers and a sequence of arrivals and departures of tasks. Each task comes with a list of servers on which it can be executed. The load balancing algorithm has to assign each task to a server on-line, with no information on future arrivals and departures of tasks. The goal of the algorithm is to minimize the maximum load on any server. The quality of an on-line algorithm is measured by the *competitive ratio*: the ratio between the maximum load it achieves and the maximum load achieved by the optimal off-line algorithm that knows the whole sequence in advance. This load balancing problem models, for example, communication in heterogeneous networks containing workstations, I/O devices, etc. Servers correspond to communication channels and tasks correspond to requests for communication links between devices. A network controller must coordinate the channels so that no channel is too heavily loaded.

On-line load balancing has been studied extensively against worst-case adversaries [9, 6, 5, 3, 8, 4]. For permanent tasks (tasks that arrive but never depart), Azar, Naor and Rom [9] showed that the competitive ratio of the greedy algorithm is $\log n$ and that no algorithm can do better. For temporary tasks (tasks that depart at unpredictable times), the works of Azar, Broder, and Karlin [6] and Azar et al. [8] show that there is an algorithm with competitive ratio $\Theta(\sqrt{n})$ and that no algorithm can do better.

It is interesting to compare these high competitive ratios, obtained from inputs generated by an adversary, to the competitive ratio against randomly generated inputs. Our results show that under reasonable probabilistic assumptions the competitive ratios for both permanent and temporary tasks are significantly better. In the case of permanent tasks, if the set of servers on which a task can be executed is a small set (that is, constant size ≥ 2) chosen at random, the competitive ratio decreases from $\Theta(\log n)$ to $\Theta(\log\log n)$. In the case of temporary tasks, if we further assume that at each time step a randomly chosen existent task is replaced by a new task, then at any fixed time the ratio between the maximum on-line load and the maximum off-line load is $\Theta(\log\log n)$ with high probability. Further details are presented in section 6.

2. Definitions and notation.

We consider two stochastic processes: the *finite process* and the *infinite process.*

The finite process. There are n boxes, initially empty, and m balls. Each ball is allowed to go into $d \geq 1$ boxes chosen independently and uniformly at random. The balls arrive one by one, and a *placement algorithm* must decide on-line (that is, without knowing what choices are available to future balls) in which box to put each ball as it comes. Decisions are irrevocable. We will subsequently refer to this setup as a (m, n, d)-problem.

The infinite process. There are n boxes, initially containing n balls in an arbitrary state. (For example, all the balls could be in one box.) At each step, one random ball is removed, and one new ball is added; the new ball is allowed to go into $d \geq 1$ boxes chosen independently and uniformly at random. Once again, a placement algorithm must decide on-line (that is, without knowing what choices are available to future balls and without knowing which ball will be removed at any future time) in which box to put each arriving ball. Decisions are irrevocable.

We use the following notations for the random variables associated with a placement algorithm A. Note that the state at time t refers to the state immediately after the placement of the tth ball.

$\lambda_j^A(t)$	called the *load* of box j, is the number of balls in box j at time t, resulting from algorithm A.
$\nu_k^A(t)$	is the number of boxes that have load k at time t.
$\nu_{\geq k}^A(t)$	is the number of boxes that have load $\geq k$ at time t, that is, $\nu_{\geq k}^A(t) = \sum_{i\geq k} \nu_i^A(t)$.
h_t^A	called the *height* of ball t (= the ball that arrives at time t), is the number of balls at time t in the box where ball t is placed. In other words, the first ball to be placed in a particular box has height 1, the second ball has height 2, etc.
$\mu_k^A(t)$	is the number of balls that have height k at time t.
$\mu_{\geq k}^A(t)$	is the number of balls that have height $\geq k$ at time t, that is, $\mu_{\geq k}^A(t) = \sum_{i\geq k} \mu_i^A(t)$.

We omit the superscript A when it is clear which algorithm we are considering. Constants were chosen for convenience, and we made no attempts to optimize them.

Algorithm GREEDY assigns ball j to the box that has the lowest load among the d random choices that j has. We use the superscript G for GREEDY.

The basic intuition behind the proofs that follow is simple: Let $p_i = \mu_{\geq i}/n$. Since the available choices for each ball are independent and $\nu_{\geq i} \leq \mu_{\geq i}$, we roughly have ("on average" and disregarding conditioning) $p_{i+1} \leq p_i^d$, which implies a doubly exponential decrease in p_i, once $\mu_{\geq i} < n/2$. Of course the truth is that $\mu_{\geq i+1}$ is strongly dependent on $\mu_{\geq i}$ and a rather complex machinery is required to construct a correct proof.

3. The finite process. We use the notation $B(n,p)$ to denote a binomially distributed random variable with parameters n and p, and start with the following standard lemma, whose proof is omitted.

LEMMA 3.1. *Let $X_1, X_2, \ldots, X_n$ be a sequence of random variables with values in an arbitrary domain, and let $Y_1, Y_2, \ldots, Y_n$ be a sequence of binary random variables, with the property that $Y_i = Y_i(X_1, \ldots, X_i)$. If*

$$\mathbf{Pr}(Y_i = 1 \mid X_1, \ldots, X_{i-1}) \le p,$$

then

$$\mathbf{Pr}\left(\sum Y_i \ge k\right) \le \mathbf{Pr}(B(n,p) \ge k),$$

and similarly if

$$\mathbf{Pr}(Y_i = 1 \mid X_1, \ldots, X_{i-1}) \ge p,$$

then

$$\mathbf{Pr}\left(\sum Y_i \le k\right) \le \mathbf{Pr}(B(n,p) \le k). \qquad \square$$

We now turn to the analysis of the finite process. In what follows, we omit the argument t when $t = m$, that is, when the process terminates. In the interest of a clearer exposition, we start with the case $m = n$, although the general case (Theorem 3.7) subsumes it.

THEOREM 3.2. *The maximum load achieved by the* GREEDY *algorithm on a random (n, n, d)-problem is less than $\ln \ln n / \ln d + O(1)$ with high probability.*

Proof. Since the d choices for a ball are independent, we have

$$\mathbf{Pr}(h_t \ge i+1 \mid \nu_{\ge i}(t-1)) = \frac{\left(\nu_{\ge i}(t-1)\right)^d}{n^d}.$$

Let $\mathcal{E}_i$ be the event that $\nu_{\ge i}(n) \le \beta_i$ where β_i will be exposed later. (Clearly, $\mathcal{E}_i$ implies that $\nu_{\ge i}(t) \le \beta_i$ for $t = 1, \ldots, n$.) Now fix $i \ge 1$ and consider a series of binary random variables Y_t for $t = 2, \ldots, n$, where

$$Y_t = 1 \text{ iff } h_t \ge i+1 \text{ and } \nu_{\ge i}(t-1) \le \beta_i.$$

(Y_t is 1 if the height of the ball t is $\ge i+1$ despite the fact that the number of boxes that have load $\ge i$ is less than β_i.)

Let ω_j represent the choices available to the jth ball. Clearly,

$$\mathbf{Pr}(Y_t = 1 \mid \omega_1, \ldots, \omega_{t-1}) \le \frac{\beta_i^d}{n^d} \overset{def}{=} p_i.$$

Thus we can apply Lemma 3.1 to conclude that

$$\mathbf{Pr}\left(\sum Y_t \ge k\right) \le \mathbf{Pr}(B(n, p_i) \ge k). \tag{3.1}$$

Observe that conditioned on $\mathcal{E}_i$, we have $\mu_{\ge i+1} = \sum Y_t$. Therefore

$$\mathbf{Pr}(\mu_{\ge i+1} \ge k \mid \mathcal{E}_i) = \mathbf{Pr}\left(\sum Y_t \ge k \mid \mathcal{E}_i\right) \le \frac{\mathbf{Pr}\left(\sum Y_t \ge k\right)}{\mathbf{Pr}(\mathcal{E}_i)}. \tag{3.2}$$

Combining (3.1) and (3.2) we obtain that

$$\mathbf{Pr}(\nu_{\geq i+1} \geq k \mid \mathcal{E}_i) \leq \mathbf{Pr}(\mu_{\geq i+1} \geq k \mid \mathcal{E}_i) \leq \frac{\mathbf{Pr}(B(n,p_i) \geq k)}{\mathbf{Pr}(\mathcal{E}_i)}. \tag{3.3}$$

We can bound large deviations in the binomial distribution with the formula (see, for instance, [2, Appendix A])

$$\mathbf{Pr}(B(n,p_i) \geq ep_i n) \leq e^{-p_i n}, \tag{3.4}$$

which inspires us to set

$$\beta_i = \begin{cases} n, & i = 1,2,\ldots,5; \\ \dfrac{n}{2e}, & i = 6; \\ \dfrac{e\beta_{i-1}^d}{n^{d-1}}, & i > 6. \end{cases}$$

With these choices $\mathcal{E}_{\geq 6} = \{\nu_6 \leq n/(2e)\}$ holds with certainty, and from (3.3) and (3.4), for $i \geq 6$

$$\mathbf{Pr}(\neg\mathcal{E}_{i+1} \mid \mathcal{E}_i) \leq \frac{1}{n^2 \mathbf{Pr}(\mathcal{E}_i)},$$

provided that $p_i n \geq 2\ln n$. Since

$$\mathbf{Pr}(\neg\mathcal{E}_{i+1}) \leq \mathbf{Pr}(\neg\mathcal{E}_{i+1} \mid \mathcal{E}_i)\mathbf{Pr}(\mathcal{E}_i) + \mathbf{Pr}(\neg\mathcal{E}_i),$$

it follows that for $p_i n \geq 2\ln n$

$$\mathbf{Pr}(\neg\mathcal{E}_{i+1}) \leq \frac{1}{n^2} + \mathbf{Pr}(\neg\mathcal{E}_i). \tag{3.5}$$

To finish the proof let i^* be the smallest i such that $\beta_{i^*}^d/n^d \leq 2\ln n/n$. Notice that $i^* \leq \ln\ln n/\ln d + O(1)$ since

$$\beta_{i+6} = \frac{ne^{(d^i-1)/(d-1)}}{(2e)^{d^i}} \leq \frac{n}{2^{d^i}}.$$

As before,

$$\mathbf{Pr}(\nu_{\geq i^*+1} \geq 6\ln n \mid \mathcal{E}_{i^*}) \leq \frac{\mathbf{Pr}(B(n, 2\ln n/n) \geq 6\ln n)}{\mathbf{Pr}(\mathcal{E}_{i^*})} \leq \frac{1}{n^2\mathbf{Pr}(\mathcal{E}_{i^*})},$$

and thus

$$\mathbf{Pr}(\nu_{\geq i^*+1} \geq 6\ln n) \leq \frac{1}{n^2} + \mathbf{Pr}(\neg\mathcal{E}_{i^*}). \tag{3.6}$$

Finally,

$$\begin{aligned} \mathbf{Pr}(\mu_{\geq i^*+2} \geq 1 \mid \ \nu_{\geq i^*+1} \leq 6\ln n) &\leq \frac{\mathbf{Pr}(B(n, (6\ln n/n)^d) \geq 1)}{\mathbf{Pr}(\nu_{\geq i^*+1} \leq 6\ln n)} \\ &\leq \frac{n(6\ln n/n)^d}{\mathbf{Pr}(\nu_{\geq i^*+1} \leq 6\ln n)} \end{aligned}$$

by the Markov inequality, and thus

$$\mathbf{Pr}(\mu_{\geq i^*+2} \geq 1) \leq \frac{(6\ln n)^d}{n^{d-1}} + \mathbf{Pr}(\nu_{\geq i^*+1} \geq 6\ln n). \tag{3.7}$$

Combining (3.7), (3.6), and (3.5), we obtain that

$$\mathbf{Pr}(\nu_{\geq i^*+2} \geq 1) \leq \frac{(6\ln n)^d}{n^{d-1}} + \frac{i^*+1}{n^2} = o(1),$$

which implies that with high probability the maximum load achieved by GREEDY is less than $i^*+2 = \ln\ln n/\ln d + O(1)$. □

We now prove a matching lower bound.

THEOREM 3.3. *The maximum load achieved by the* GREEDY *algorithm on a random* (n,n,d)*-problem is at least* $\ln\ln n/\ln d - O(1)$ *with high probability.*

Proof. Let $\mathcal{F}_i$ be the event that $\nu_{\geq i}(n(1-1/2^i)) \geq \gamma_i$ where γ_i will be exposed later. For the time being, it suffices to say that $\gamma_{i+1} < \gamma_i/2$. We want to compute $\mathbf{Pr}(\neg\mathcal{F}_{i+1} \mid \mathcal{F}_i)$. To this aim, for t in the range $R = \{n(1-1/2^i)+1, \ldots, n(1-1/2^{i+1})\}$, let Z_t be defined by

$$Z_t = 1 \text{ iff } h_t = i+1 \text{ or } \nu_{\geq i+1}(t-1) \geq \gamma_{i+1},$$

and observe that while $\nu_{\geq i+1}(t-1) < \gamma_{i+1}$, if $Z_t = 1$, then the box where the tth ball is placed had load exactly i at time $t-1$. This means that all the d choices that ball t had pointed to boxes with load $\geq i$ and at least one choice pointed to a box with load exactly i.

Now let ω_j represent the choices available to the jth ball. Using

$$\begin{aligned}\mathbf{Pr}(A \vee B \mid C) &= \mathbf{Pr}(A \wedge \bar{B} \mid C) + \mathbf{Pr}(B \mid C)\\ &= \mathbf{Pr}(A \mid \bar{B} \wedge C)\mathbf{Pr}(\bar{B} \mid C) + \mathbf{Pr}(B \mid C) \geq \mathbf{Pr}(A \mid \bar{B} \wedge C),\end{aligned}$$

and in view of the observation above, we derive that

$$\mathbf{Pr}(Z_t = 1 \mid \omega_1, \ldots, \omega_{t-1}, \mathcal{F}_i) \geq \left(\frac{\gamma_i}{n}\right)^d - \left(\frac{\gamma_{i+1}}{n}\right)^d \geq \frac{1}{2}\left(\frac{\gamma_i}{n}\right)^d \overset{def}{=} p_i. \tag{3.8}$$

Applying Lemma 3.1 we get

$$\mathbf{Pr}\left(\sum_{t\in R} Z_t \leq k \;\middle|\; \mathcal{F}_i\right) \leq \mathbf{Pr}(B(n/2^{i+1}, p_i) \leq k).$$

We now choose

$$\begin{aligned}\gamma_0 &= n;\\ \gamma_{i+1} &= \frac{\gamma_i^d}{2^{i+3}n^{d-1}} = \frac{n}{2^{i+3}}\left(\frac{\gamma_i}{n}\right)^d = \frac{1}{2}\frac{n}{2^{i+1}}p_i.\end{aligned}$$

Since $\mathbf{Pr}(B(N,p) < Np/2) < e^{-Np/8}$ (see, for instance, [2, Appendix A]), it follows that

$$\mathbf{Pr}(B(n/2^{i+1}, p_i) \leq \gamma_{i+1}) = o(1/n^2), \tag{3.9}$$

provided that $p_i n/2^{i+1} \geq 17\ln n$. Let i^* be the largest integer for which this holds. Clearly $i^* = \ln\ln n/\ln d - O(1)$.

Now observe that by the definition of $\mathcal{F}$ and Z_t, the event $\{\sum_{t\in R} Z_t \geq \gamma_{i+1}\}$ implies $\mathcal{F}_{i+1}$. Thus in view of (3.8) and (3.9)

$$\mathbf{Pr}(\neg\mathcal{F}_{i+1} \mid \mathcal{F}_i) \leq \mathbf{Pr}\left(\sum_{t\in R} Z_t \leq \gamma_{i+1} \,\middle|\, \mathcal{F}_i\right) = o(1/n^2),$$

and therefore

$$\begin{aligned}\mathbf{Pr}(\mathcal{F}_{i^*}) &\geq \mathbf{Pr}(\mathcal{F}_{i^*} \mid \mathcal{F}_{i^*-1}) \times \mathbf{Pr}(\mathcal{F}_{i^*-1} \mid \mathcal{F}_{i^*-2}) \times \cdots \times \mathbf{Pr}(\mathcal{F}_1 \mid \mathcal{F}_0) \times \mathbf{Pr}(\mathcal{F}_0)\\ &\geq (1-1/n^2)^{i^*} = 1 - o(1/n),\end{aligned}$$

which completes the proof. ∎

We now turn to showing that the GREEDY algorithm is stochastically optimal under our model, that is, we assume that each ball has d destinations chosen uniformly at random and that all balls have equal weight. (The optimality is not preserved if either condition is violated.) It suffices to consider only deterministic algorithms since randomized algorithms can be considered as a distribution over deterministic algorithms.

We say that a vector $\bar{v} = (v_1, v_2, \ldots, v_n)$ *majorizes* a vector $\bar{u}$, written $\bar{v} \succeq \bar{u}$, if for $1 \leq i \leq n$, we have $\sum_{1\leq j\leq i} v_{\pi(j)} \geq \sum_{1\leq j\leq i} u_{\sigma(j)}$, where π and σ are permutations of $1, \ldots, n$ such that $v_{\pi(1)} \geq v_{\pi(2)} \geq \cdots \geq v_{\pi(n)}$ and $u_{\sigma(1)} \geq u_{\sigma(2)} \geq \cdots \geq u_{\sigma(n)}$.

LEMMA 3.4. *Let $\bar{v}$ and $\bar{u}$ be two positive integer vectors such that $v_1 \geq v_2 \geq \cdots \geq v_n$ and $u_1 \geq u_2 \geq \cdots \geq u_n$. If $\bar{v} \succeq \bar{u}$ then also $\bar{v} + \bar{e}_i \succeq \bar{u} + \bar{e}_i$, where $\bar{e}_i$ is the ith unit vector, that is $\bar{e}_{i,j} = \delta_{i,j}$.*

Proof. Let $S_j(\bar{x})$ be the sum of the j largest components of the vector $\bar{x}$. Notice first that for all j

$$S_j(\bar{x}) \leq S_j(\bar{x} + \bar{e}_i) \leq S_j(\bar{x}) + 1. \tag{3.10}$$

By hypothesis, for all j, we have $S_j(\bar{v}) \geq S_j(\bar{u})$. To prove the lemma we show that for all j, we also have $S_j(\bar{v} + \bar{e}_i) \geq S_j(\bar{u} + \bar{e}_i)$. Fix j. By (3.10) if $S_j(\bar{v}) > S_j(\bar{u})$, then $S_j(\bar{v} + \bar{e}_i) \geq S_j(\bar{u} + \bar{e}_i)$. Now assume $S_j(\bar{v}) = S_j(\bar{u})$. There are three cases to consider:

Case 1. $i \leq j$. Then

$$S_j(\bar{v} + \bar{e}_i) = S_j(\bar{v}) + 1 = S_j(\bar{u}) + 1 = S_j(\bar{u} + \bar{e}_i).$$

Case 2. $i > j$ and $u_j > u_i$. Since $u_j \geq u_i + 1$, it follows that $S_j(\bar{u}) = S_j(\bar{u} + \bar{e}_i)$ and therefore

$$S_j(\bar{v} + \bar{e}_i) \geq S_j(\bar{v}) = S_j(\bar{u}) = S_j(\bar{u} + \bar{e}_i).$$

Case 3. $i > j$ and $u_j = u_{j+1} = \cdots = u_i$. Observe first that since $S_{j-1}(\bar{v}) \geq S_{j-1}(\bar{u})$, $S_j(\bar{v}) = S_j(\bar{u})$, and $S_{j+1}(\bar{v}) \geq S_{j+1}(\bar{u})$, we have

$$v_j \leq u_j \qquad \text{and} \qquad v_{j+1} \geq u_{j+1}.$$

Hence

$$v_j \geq v_{j+1} \geq u_{j+1} = u_j \geq v_j.$$

We conclude that $v_j = u_j = v_{j+1} = u_{j+1}$, and thus $S_{j+1}(\bar{v}) = S_{j+1}(\bar{u})$. Repeating the argument, we obtain that

$$v_j = u_j = v_{j+1} = u_{j+1} = \cdots = v_i = u_i,$$

and therefore

$$S_j(\bar{v} + \bar{e}_i) = S_j(\bar{v}) + 1 = S_j(\bar{u}) + 1 = S_j(\bar{u} + \bar{e}_i). \qquad \Box$$

Let Ω be the set of all possible n^d choices for each ball and Ω^t be the set of sequences of choices for the first t balls.

THEOREM 3.5. *For any on-line deterministic algorithm A, and $t \geq 0$, there is* 1-1 *correspondence $f : \Omega^t \to \Omega^t$ such that for any $\omega_t \in \Omega^t$ the vector of box loads associated with* GREEDY *acting on ω_t, written*

$$\bar{\lambda}^G(\omega_t) = (\lambda_1^G(\omega_t), \lambda_2^G(\omega_t), \ldots, \lambda_n^G(\omega_t)),$$

is majorized by the vector of box loads associated with A acting on $f(\omega_t)$, that is

$$\bar{\lambda}^G(\omega_t) \preceq \bar{\lambda}^A(f(\omega_t)).$$

Proof. To simplify notation we assume $d = 2$. The proof for larger d is analogous. The proof proceeds by induction on t, the length of the sequence. The base case $(t = 0)$ is obvious. Assume the theorem valid for t and let f_t be the mapping on Ω^t. Fix a sequence $\omega_t \in \Omega^t$. It suffices to show that we can refine f_t to obtain a 1-1 correspondence for all possible 1-step extensions of ω_t. Without loss of generality, renumber the boxes such that

$$\lambda_1^G(\omega_t) \geq \lambda_2^G(\omega_t) \geq \cdots \geq \lambda_n^G(\omega_t),$$

and let π be a permutation of $1, \ldots, n$ such that

$$\lambda_{\pi(1)}^A(f_t(\omega_t)) \geq \lambda_{\pi(2)}^A(f_t(\omega_t)) \geq \cdots \geq \lambda_{\pi(n)}^A(f_t(\omega_t)).$$

Let (i, j) be two choices for the $t + 1$ ball. For every i, j we define

$$f_{t+1}(\omega_t \diamond (i, j)) = f_t(\omega_t) \diamond (\pi(i), \pi(j)),$$

where "$\diamond$" represents extension of sequences.

Clearly f_{t+1} is 1-1. We need to show that

$$\bar{\lambda}^G(\omega_t \diamond (i, j)) \preceq \bar{\lambda}^A\bigl(f_t(\omega_t) \diamond (\pi(i), \pi(j))\bigr).$$

Notice that when the sequence ω_t is extended by the step (i, j) for any algorithm, exactly one component of the vector $\bar{\lambda}(\omega_t)$ changes, namely either $\lambda_i(\omega_t)$ or $\lambda_j(\omega_t)$ increases by one. Assume that $i \geq j$; then

$$\bar{\lambda}^G(\omega_t \diamond (i, j)) = \bar{\lambda}^G(\omega_t) + \bar{e}_i \preceq \bar{\lambda}^A(f_t(\omega_t)) + \bar{e}_{\pi(i)} \preceq \bar{\lambda}^A\bigl(f_t(\omega_t) \diamond (\pi(i), \pi(j))\bigr),$$

where the first inequality follows from Lemma 3.4 and the second is due to the fact that

$$\bar{\lambda}^A(f_t(\omega_t)) + \bar{e}_{\pi(i)} \preceq \bar{\lambda}^A(f_t(\omega_t)) + \bar{e}_{\pi(j)}. \qquad \Box$$

COROLLARY 3.6. *For any fixed k and any t*

$$\mathbf{Pr}(\max_i \lambda_i^A(t) > k) \geq \mathbf{Pr}(\max_i \lambda_i^G(t) > k).$$

We are now ready to discuss the general case of the finite process.

THEOREM 3.7. *The maximum load achieved by the* GREEDY *algorithm on a random (m,n,d)-problem, with $d \geq 2$ and $m \geq n$, is, with high probability, less than $(1+o(1))\ln\ln n/\ln d + O(m/n)$.*

Proof. We start by replaying the proof of Theorem 3.2, taking into account the fact that there are now m balls. So let $\mathcal{E}_i$ be the event that $\nu_{\geq i}(m) \leq \beta_i$, and define $p_i = \beta_i^d/n^d$. Following the proof of Theorem 3.2 we derive that

$$\mathbf{Pr}(\nu_{\geq i+1} \geq k \mid \mathcal{E}_i) \leq \frac{\mathbf{Pr}(B(m,p_i) \geq k)}{\mathbf{Pr}(\mathcal{E}_i)}.$$

Suppose that for some value x we set $\beta_x = n^2/(2em)$ and show that $\mathcal{E}_x$ holds with high probability, that is,

$$\mathbf{Pr}\left(\nu_x \geq \frac{n^2}{2em}\right) = o(1). \tag{3.11}$$

Then

$$\beta_{i+x} = \frac{n}{2^{d^i}}\left(\frac{me}{n}\right)^{(d^i-1)/(d-1)-d^i} \leq \frac{n}{2^{d^i}},$$

and continuing as before, we obtain that

$$\mathbf{Pr}(\mu \geq x + \ln\ln n/\ln d + 2) = o(1).$$

It remains to be shown that x can be taken to be $O(m/n) + o(\ln\ln n/\ln d)$. First assume that $m/n \geq w(n)$ where $w(n)$ is an increasing function of n, but $w(n) = o(\ln\ln n/\ln d)$. Then we claim that we can take $x = \lceil em/n \rceil$.

Consider a placement algorithm, denoted R, that always puts a ball in the box corresponding to the first choice offered. This is entirely equivalent with the case $d = 1$, the classical occupancy problem. The load within a box under this process is a binomial random variable $B(m, 1/n)$, and therefore (via (3.4)), the probability that the load within a box exceeds em/n is bounded by $e^{-m/n}$. Now consider the height of the tth ball, denoted h_t^R. The probability that the box into which the tth ball is placed has load greater than em/n is less than $e^{-m/n}$, and therefore the expected number of balls of height $\geq em/n$ satisfies

$$\mathbf{E}(\mu^R_{\geq em/n}) \leq me^{-m/n}.$$

Hence by Markov's inequality

$$\mathbf{Pr}\left(\mu^R_{\geq em/n} \geq \frac{n^2}{2em}\right) \leq \frac{2em^2}{n^2}e^{-m/n} = o(1),$$

since $m/n \to \infty$.

We claim that Theorem 3.5 implies

$$\mathbf{Pr}\left(\mu^G_{\geq k} \geq r\right) \leq \mathbf{Pr}\left(\mu^R_{\geq k} \geq r\right). \tag{3.12}$$

Indeed, suppose that there is an outcome ω_t for which GREEDY has exactly i boxes with load greater than or equal to k. As in the proof of Theorem 3.5, renumber the boxes such that

$$\lambda_1^G(\omega_t) \geq \lambda_2^G(\omega_t) \geq \cdots \geq \lambda_n^G(\omega_t).$$

Let $f_t(\omega_t)$ be the corresponding outcome for algorithm R and let π be a permutation of $1, \ldots, n$ such that

$$\lambda_{\pi(1)}^R(f_t(\omega_t)) \geq \lambda_{\pi(2)}^R(f_t(\omega_t)) \geq \cdots \geq \lambda_{\pi(n)}^R(f_t(\omega_t)).$$

Then

$$\mu_{\geq k}^G(\omega_t) = \sum_{1 \leq j \leq i} (\lambda_j^G(\omega_t) - (k-1))$$

and

$$\mu_{\geq k}^R(f_t(\omega_t)) \geq \sum_{1 \leq j \leq i} (\lambda_{\pi(j)}^R(f_t(\omega_t)) - (k-1)).$$

But Theorem 3.5 implies that

$$\sum_{1 \leq j \leq i} \lambda_j^R(f(\omega_t)) \geq \sum_{1 \leq j \leq i} \lambda_j^G(\omega_t),$$

and by considering all outcomes we obtain (3.12). Therefore,

$$\mathbf{Pr}\left(\mu_{\geq em/n}^G \geq \frac{n^2}{2em}\right) \leq \mathbf{Pr}\left(\mu_{\geq em/n}^R \geq \frac{n^2}{2em}\right),$$

and since

$$\mathbf{Pr}\left(\nu_{\geq em/n}^G \geq \frac{n^2}{2em}\right) \leq \mathbf{Pr}\left(\mu_{\geq em/n}^G \geq \frac{n^2}{2em}\right),$$

we have that for $x = \lceil em/n \rceil$ (3.11) is satisfied.

To remove the assumption $m/n \geq w(n)$, we can simply imagine that the number of balls is increased to $\max(m, nw(n))$. Then the corresponding value of x becomes $O(\max(m, nw(n))/n) = O(m/n) + o(\ln \ln n / \ln d)$. ◻

4. The infinite process. In this section we consider the infinite process. Analogously to Theorem 3.5 it is possible to show that the GREEDY algorithm minimizes the expected maximum load on any box. We analyze its performance below. The main theorem of this section is the following.

THEOREM 4.1. *Assume that the infinite process starts in an arbitrary state. Under* GREEDY*, with $d \geq 2$, there is a constant c such that for any fixed $T \geq cn^2 \log \log n$,*

$$\mathbf{Pr}(\exists j, \lambda_j(T) \geq \ln \ln n / \ln d + O(1)) = o(1).$$

Thus in the stationary distribution the maximum load is $\ln \ln n / \ln d + O(1)$ with high probability.

Proof. For simplicity of presentation we state and prove the results only for $d = 2$. The proof assumes that at time $T - cn^2 \log \log n$ the process is in an arbitrary state and therefore we can let $T = cn^2 \log \log n$ with no loss of generality.

By the definition of the process, the number of balls of height at least i cannot change by more than 1 in a time step, that is $|\mu_{\geq i}(t+1) - \mu_{\geq i}(t)| \leq 1$. The random variable $\mu_{\geq i}(t)$ can be viewed as a random walk on the integers l, $0 \leq l \leq n$. The proof is based on bounding the maximum values taken by the variables $\mu_{\geq i}(t)$ by studying the underlying process.

We define an integer i^* and a decreasing sequence α_i, for $200 \leq i \leq i^* + 1$ as follows:

$$\begin{aligned}
\alpha_{200} &= \frac{n}{200}, \\
\alpha_i &= \frac{100\alpha_{i-1}^2}{n} && \text{for } i > 200 \text{ and } \alpha_{i-1} \geq \sqrt{n \log_2 n}, \\
\alpha_{i^*} &= 100 \log_2 n, && i^* = \text{the smallest } i \text{ for which} \\
& && \quad \alpha_{i-1} < \sqrt{n \log_2 n}, \\
\alpha_{i^*+1} &= 100.
\end{aligned}$$

Clearly $i^* \leq \ln \ln n / \ln 2 + O(1)$. For future reference, observe also that for $200 < i \leq i^* + 1$

$$\alpha_i \geq \frac{100\alpha_{i-1}^2}{n}. \tag{4.1}$$

We also define an increasing sequence of times: $t_{200} = 0$ and $t_i = t_{i-1} + n^2$ for $i > 200$. Thus $t_{i^*+1} = O(n^2 \log \log n) = O(T)$.

Let $\{\mu_{\geq i}[t^-, t^+] \leq \alpha\}$ denote the event that $\mu_{\geq i}(t) \leq \alpha$ for all t, such that $t^- < t \leq t^+$, and similarly, let $\{\mu_{\geq i}[t^-, t^+] > \alpha\}$ denote the event that $\mu_{\geq i}(t) > \alpha$ for all t, such that $t^- < t \leq t^+$. We define the events C_i as follows:

$$\begin{aligned}
C_{200} &= \{\nu_{\geq 200}[t_{200}, T] \leq 2\alpha_{200}\} \equiv \{\nu_{\geq 200}[0, T] \leq n/100\}; \\
C_i &= \{\mu_{\geq i}[t_i, T] \leq 2\alpha_i\} \qquad \text{for } i > 200.
\end{aligned}$$

Note that C_{200} always holds, and for $i > 200$, the event $C - i$ implies that $\nu_{\geq i}[t_i, T] \leq 2\alpha_i$.

We shall prove inductively that for all $i = 200, \ldots, i^* + 1$

$$\mathbf{Pr}(\neg C_i) \leq \frac{2i}{n^2}. \tag{4.2}$$

This implies that the event $\{\mu_{\geq i^*+1}[t_{i^*+1}, T] \leq 200\}$ occurs with probability $1 - o(1)$, and therefore with high probability, for every j, $\lambda_j(T) \leq i^* + 201 = \ln \ln n / \ln 2 + O(1)$, which completes the proof of the main part of the theorem.

Finally, we show that in the stationary distribution

$$\mathbf{Pr}(\forall j, \lambda_j \leq \log \log n + O(1)) = 1 - o(1).$$

Indeed, let S be the set of states such that for all $j, \lambda_j(t) \leq \log \log n + O(1)$. Let $s(t)$ be the state of the chain at time t. Then the previous observation implies that

$$\mathbf{Pr}(s(t+T) \notin S \mid s(t)) = o(1).$$

Let π be the stationary distribution; then

$$\sum_{i \notin S} \pi_i = \sum_j \mathbf{Pr}(s(t+T) \notin S \mid s(t) = j) \cdot \pi_j = \sum_j \pi_j o(1) = o(1),$$

which completes the proof of the theorem assuming (4.2). To prove it, we show that conditioned on C_{i-1}:

a. With high probability $\mu_{\geq i}(t)$ becomes less than α_i before time t_i. (This is shown in Lemma 4.2.)
b. If $\mu_{\geq i}(t)$ becomes less than α_i at any time before T, from then until T, with high probability, it does not become larger than $2\alpha_i$. (This is shown in Lemma 4.3.)

The two facts above imply that if C_{i-1} holds, then with high probability $\mu_{\geq i}[t_i, T] \leq 2\alpha_i$, that is, C_i holds as well.

Base case. The base case is straightforward since $\mathbf{Pr}(\neg C_{200}) = \mathbf{Pr}(\neg\{\nu_{\geq 200}[0,T] \leq n/100\}) = 0$.

Induction. Suppose that

$$\mathbf{Pr}(\neg C_{i-1}) \leq \frac{2(i-1)}{n^2}, \tag{4.3}$$

where $200 < i \leq i^* + 1$.

Let $s(t)$ be the state at time t. It is easy to verify the following bounds on the underlying transition probabilities. For any t,

$$\mathbf{Pr}(\mu_{\geq i}(t+1) > \mu_{\geq i}(t) \mid s(t)) \leq \left(\frac{\nu_{\geq(i-1)}(t)}{n}\right)^2 \leq \left(\frac{\mu_{\geq(i-1)}(t)}{n}\right)^2 \tag{4.4}$$

and

$$\mathbf{Pr}(\mu_{\geq i}(t+1) < \mu_{\geq i}(t) \mid s(t)) \geq \frac{\mu_{\geq i}(t)}{n}\left(1 - \left(\frac{\nu_{\geq(i-1)}(t)}{n}\right)^2\right) \geq \frac{\mu_{\geq i}(t)}{2n}. \tag{4.5}$$

From (4.4) and (4.5) we obtain that the transition probabilities satisfy

$$\mathbf{Pr}(\mu_{\geq i}(t+1) > \mu_{\geq i}(t) \mid \mu_{\geq i-1}(t) \leq 2\alpha_{i-1}) \leq \left(\frac{2\alpha_{i-1}}{n}\right)^2 \overset{def}{=} q_i^+$$

and

$$\mathbf{Pr}(\mu_{\geq i}(t+1) < \mu_{\geq i}(t) \mid \mu_{\geq i}(t) \geq \alpha_i) \geq \frac{\alpha_i}{2n} \overset{def}{=} q_i^-.$$

Thus in view of (4.1)

$$q_i^+ \leq \frac{\alpha_i}{25n}.$$

We define two new binary random variables for $0 < t \leq T$ as follows:

$$X_t = 1 \text{ iff } \mu_{\geq i}(t) > \mu_{\geq i}(t-1) \text{ and } \mu_{\geq i-1}(t-1) \leq 2\alpha_{i-1},$$

and

$$Y_t = 1 \text{ iff } \mu_{\geq i}(t) < \mu_{\geq i}(t-1) \text{ or } \mu_{\geq i}(t-1) \leq \alpha_i.$$

Clearly

$$\mathbf{Pr}(X_t = 1) \leq q_i^+ \qquad \text{and} \qquad \mathbf{Pr}(Y_t = 1) \geq q_i^-. \tag{4.6}$$

We also define F_i to be the event

$$F_i \overset{def}{=} \{\exists t^* \in [t_{i-1}, t_i] \text{ s.t. } \mu_{\geq i}(t^*) \leq \alpha_i\};$$

thus $\neg F_i$ is the event

$$\neg F_i = \{\mu_{\geq i}[t_{i-1}, t_i] > \alpha_i\}.$$

Two lemmas are necessary in order to conclude that $\mathbf{Pr}(\neg C_i) \leq 2i/n^2$.

LEMMA 4.2. *Under the inductive hypothesis*

$$\mathbf{Pr}(\neg F_i \mid C_{i-1}) \leq \frac{1}{n^2}.$$

Proof. Notice that conditioned on C_{i-1}, the sum $\sum_{t\in[t_{i-1},t_i]} X_t$ is the number of times $\mu_{\geq i}(t)$ increased in the interval $[t_{i-1}, t_i]$; similarly, if within this interval $\mu_{\geq i}$ did not become less than α_i, then $\sum_{t\in[t_{i-1},t_i]} Y_t$ equals the number of times $\mu_{\geq i}(t)$ decreased in this interval. We conclude that

$$\begin{aligned}\mathbf{Pr}(\neg F_i \mid C_{i-1}) &\leq \mathbf{Pr}\left(\sum_{t\in[t_{i-1},t_i]} Y_t - \sum_{t\in[t_{i-1},t_i]} X_t \leq n \;\middle|\; C_{i-1}\right) \\ &\leq \frac{1}{\mathbf{Pr}(C_{i-1})}\mathbf{Pr}\left(\sum_{t\in[t_{i-1},t_i]} Y_t - \sum_{t\in[t_{i-1},t_i]} X_t \leq n\right).\end{aligned}$$

In view of (4.6) and Lemma 3.1, Chernoff-type bounds imply that for every $i \leq i^* + 1$

$$\mathbf{Pr}\left(\sum_{t\in[t_{i-1},t_i]} X_t > 2n^2 q_i^+ \right) \leq \mathbf{Pr}\left(B(n^2, q_i^+) \geq 2n^2 q_i^+\right) \leq e^{-\Omega(n^2 q_i^+)} = o(1/n^c)$$

and

$$\mathbf{Pr}\left(\sum_{t\in[t_{i-1},t_i]} Y_t < \tfrac{1}{2} n^2 q_i^- \right) \leq \mathbf{Pr}\left(B(n^2, q_i^-) \leq \tfrac{1}{2} n^2 q_i^-\right) \leq e^{-\Omega(n^2 q_i^-)} = o(1/n^c)$$

for any constant c. On the other hand, in view of (4.1),

$$\frac{1}{2} n^2 q_i^- - 2n^2 q_i^+ \geq \frac{1}{4} n\alpha_i - \frac{2}{25} n\alpha_i \geq \frac{n\alpha_i}{10} \geq n,$$

and therefore we conclude that

$$\mathbf{Pr}(\neg F_i \mid C_{i-1}) \leq \frac{1}{n^c \mathbf{Pr}(C_{i-1})}$$

for any constant c. Taking $c = 3$ and using the inductive hypothesis on C_{i-1} (4.3) completes the proof. ∎

LEMMA 4.3. *Under the inductive hypothesis*

$$\mathbf{Pr}(\neg C_i \mid C_{i-1}, F_i) \leq \frac{1}{n^2}.$$

Proof. Since $\mathbf{Pr}(A \mid B \wedge C) \le \mathbf{Pr}(A \wedge B|C)$ we get that

$$\begin{aligned}
&\mathbf{Pr}(\neg C_i \mid C_{i-1}, F_i)\\
&\qquad \le \mathbf{Pr}(\neg C_i \wedge F_i \mid C_{i-1})\\
&\qquad \le \mathbf{Pr}(\exists t_1, t_2 \in [t_{i-1}, T]\\
&\qquad\qquad\qquad \text{s.t. } \mu_{\ge i}(t_1) = \alpha_i, \mu_{\ge i}(t_2) = 2\alpha_i, \mu_{\ge i}[t_1, t_2] \ge \alpha_i \mid C_{i-1})\\
&\qquad \le \sum_{t_{i-1} \le t_1 < t_2 \le T} \mathbf{Pr}(\mu_{\ge i}(t_1) = \alpha_i, \mu_{\ge i}(t_2) = 2\alpha_i, \mu_{\ge i}[t_1, t_2] \ge \alpha_i \mid C_{i-1})\\
&\qquad \le \sum_{t_{i-1} \le t_1 < t_2 \le T} \mathbf{Pr}\left(\sum_{t \in [t_1, t_2]} X_t - \sum_{t \in [t_1, t_2]} Y_t \ge \alpha_i \mid C_{i-1} \right)\\
&\qquad \le \frac{1}{\mathbf{Pr}(C_{i-1})} \sum_{t_{i-1} \le t_1 < t_2 \le T} \mathbf{Pr}(\sum_{t \in [t_1, t_2]} X_t - \sum_{t \in [t_1, t_2]} Y_t \ge \alpha_i).
\end{aligned}$$

Fix t_1 and t_2 and let $\Delta = t_2 - t_1$. We now consider four cases.

A. $\Delta \le n$ and $i \le i^*$:

$$\mathbf{Pr}\Big(\sum_{t \in [t_1, t_2]} X_t \ge \alpha_i \Big) \le \binom{\Delta}{\alpha_i} (q^+)^{\alpha_i} \le \left(\frac{e\Delta}{\alpha_i} \cdot \frac{\alpha_i}{25n} \right)^{\alpha_i} \le n^{-100}.$$

B. $\Delta \le n \log n$ and $i = i^* + 1$:

$$\begin{aligned}
\mathbf{Pr}\Big(\sum_{t \in [t_1, t_2]} X_t \ge \alpha_{i^*+1} \Big) &\le \binom{\Delta}{\alpha_{i^*+1}} (q^+)^{\alpha_{i^*+1}} \le \left(\frac{e\Delta}{\alpha_{i^*+1}} \cdot \frac{4\alpha_{i^*}^2}{n^2} \right)^{\alpha_{i^*+1}}\\
&\le \left(\frac{en \log n}{100} \cdot \frac{4 \cdot 100^2 \log^2 n}{n^2} \right)^{100} \le n^{-100}.
\end{aligned}$$

C. $\Delta \ge n$ and $i \le i^*$: Again using large deviation bounds and the fact that $\alpha_i \Delta \ge 100 \log n$ we obtain that

$$\mathbf{Pr}\left(\sum_{t \in [t_1, t_2]} Y_t \le \frac{1}{2} q^- \Delta \right) \le e^{-q^- \Delta/8} = e^{-\alpha_i \Delta/(16n)} \le n^{-6.1}$$

and that

$$\mathbf{Pr}\left(\sum_{t \in [t_1, t_2]} X_t \ge \frac{1}{2} q^- \Delta \right) \le \left(\frac{2eq^+ \Delta}{q^- \Delta} \right)^{q^- \Delta/2} \le \left(\frac{4e}{25} \right)^{\alpha_i \Delta/(4n)} \le n^{-25}.$$

D. $\Delta \ge n \log n$ and $i = i^* + 1$: We use the same proof as case **C** using the fact that $\alpha_{i^*+1} \Delta \ge 100 \log n$.

Thus in all four cases,

$$\mathbf{Pr}\left(\sum_{t \in [t_1, t_2]} X_t - \sum_{t \in [t_1, t_2]} Y_t \ge \alpha_i \right) \le \frac{1}{n^{6.1}},$$

therefore, under the induction hypothesis,

$$\frac{1}{\mathbf{Pr}(C_{i-1})} \sum_{t_{i-1} \le t_1 < t_2 \le T} \mathbf{Pr}\Big(\sum_{t \in [t_1, t_2]} X_t - \sum_{t \in [t_1, t_2]} Y_t \ge \alpha_i \Big) \le \frac{2T^2}{n^{6.1}} \le \frac{1}{n^2}. \qquad \square$$

Returning to the proof of (4.2), by using the induction hypothesis, Lemmas 4.2 and 4.3, and the law of total probability, we can complete the induction as follows:

$$\begin{aligned}
\mathbf{Pr}(\neg C_i) &= \mathbf{Pr}(\neg C_i \mid C_{i-1}) \cdot \mathbf{Pr}(C_{i-1}) \\
&\quad + \mathbf{Pr}(\neg C_i \mid \neg C_{i-1}) \cdot \mathbf{Pr}(\neg C_{i-1}) && \text{Now apply IH} \\
&\le \mathbf{Pr}(\neg C_i \mid C_{i-1}) + 2(i-1)/n^2 \\
&= \mathbf{Pr}(\neg C_i \mid C_{i-1}, F_i) \cdot \mathbf{Pr}(F_i \mid C_{i-1}) && \text{Now apply Lemma 4.3} \\
&\quad + \mathbf{Pr}(\neg C_i \mid C_{i-1}, \neg F_i) \cdot \mathbf{Pr}(\neg F_i \mid C_{i-1}) && \text{Now apply Lemma 4.2} \\
&\quad + 2(i-1)/n^2 \\
&\le 1/n^2 + 1/n^2 + 2(i-1)/n^2 = 2i/n^2. \qquad \square
\end{aligned}$$

5. Hashing. We define a simple hashing algorithm, called 2-*way chaining*, by analogy with the popular direct chaining method. We use two random hash functions. For each key, the two hash functions define two indices in a table. Each table location contains a pointer to a linked list. When a new key arrives, we compare the current length of the two lists associated to the key, and the key is inserted at the end of the shortest list. (The direct chaining method corresponds to having only one associated random index.)

For searching, the two hash values are computed, and two linked lists are searched in alternate order. (That is, after checking the ith element of the first list, we check the ith element of the second list, then element $i+1$ of the first list, and so on.) When the shorter list is exhausted, we continue searching the longer list until it is exhausted as well. (In fact, if no deletions are allowed, we can stop after checking only one more element in the longer list. For the analysis below, this is immaterial.)

Assume that n keys are sequentially inserted by this process to a table of size n. Theorem 1.1 analysis implies that with high probability the maximum access time, which is bounded by twice the length of the longest list, is $2 \ln n \ln n / \ln 2 + O(1)$, versus the $\Theta(\log n / \log\log n)$ time when one random hash function is used. More generally, if m keys are stored in the table with d hash functions, then the maximum access time under this scheme is $2(1 + o(1)) \ln\ln n / \ln d + \Theta(m/n)$.

Next we show that the average access time of 2-way chaining is no more than twice the average access time of the standard direct chaining method. As customary, we discuss the average access time separately for successful searches and unsuccessful searches. The latter, denoted C'_G, is bounded by twice the expected cost of checking a list chosen uniformly at random. Therefore

$$C'_G(m,n) \le 2 + \frac{2m}{n}.$$

For successful searches, the cost C_G, is given by

$$C_G(m,n) \le \frac{2}{m} \sum_{1 \le i \le m} h_i = \frac{2}{m} \sum_{1 \le j \le n} \binom{\lambda_j + 1}{2},$$

where all the notations are as in section 2. Since we know that ν_k eventually decreases doubly exponentially, we can bound C_G via the inequality

$$C_G(m,n) \le \frac{2}{m} \sum_{k>0} k\nu_{\ge k}.$$

However, we can achieve better bounds, using the majorization Theorem 3.5. We start from the following.

LEMMA 5.1. *Let $\bar{v} = (v_1, v_2, \ldots, v_n)$ and $\bar{u} = (u_1, v_2, \ldots, u_n)$ be two positive integer vectors. If $\bar{v} \succeq \bar{u}$, then*

$$\sum_{1\le i\le n} v_i^2 \ge \sum_{1\le i\le n} u_i^2.$$

This lemma is a special case of a well-known theorem from majorization (see, e.g., [22]), but for completeness we present a proof.

Proof. Let $\bar{x}$ be a n-vector and let $(\bar{x}, \bar{u})$ denote the inner product of $\bar{x}$ and $\bar{u}$. Consider the linear program

$$\text{Maximize } (\bar{x}, \bar{u}) \text{ subject to } \bar{x} \preceq \bar{v} \text{ and } \bar{x} \ge 0.$$

It is easy to check that $\bar{x} = \bar{u}$ is a feasible point and that the optimal solution is $\bar{x} = \bar{v}$. Hence $(\bar{u}, \bar{u}) \le (\bar{v}, \bar{u})$. Now consider the same program with the objective function $(\bar{x}, \bar{v})$. Then again $\bar{x} = \bar{u}$ is a feasible point and the optimal solution is $\bar{x} = \bar{v}$. Hence $(\bar{u}, \bar{u}) \le (\bar{u}, \bar{v}) \le (\bar{v}, \bar{v})$. ◻

Consider now the standard direct chaining method. In our terminology it corresponds to the random placement algorithm R and it therefore majorizes G. It is well known that the cost for successful search for direct chaining is [19, Ex. 6.4.34]

$$C_R(m,n) = \frac{1}{m} \sum_{1\le j\le n} \binom{\lambda_j^R + 1}{2} = 1 + \frac{m-1}{2n}.$$

Applying the lemma above we obtain that the cost of successful search in 2-way chaining satisfies

$$C_G(m,n) \le 2 + \frac{m-1}{n}.$$

6. Competitive on-line load balancing.

6.1. Preliminaries.

6.1. Preliminaries. The on-line load balancing problem is defined as follows. Let M be a set of servers (or machines) that is supposed to run a set of tasks that arrive and depart in time. Each task j has associated with it a weight, or load, $w(j) \ge 0$, an arrival time $\tau(j)$, and a set $M(j) \subset M$ of servers capable of running it. We distinguish among two variants of this problem: the case of *permanent tasks*, tasks that arrive but never depart, and the case of *temporary tasks*, tasks that depart the system at a time unknown in advance.

As soon as each task arrives, it must be assigned to exactly one of the servers capable of running it, and once assigned, it can not be transferred to a different server. The assigned server starts to run the task immediately, and continues to run it until the task departs.

When task j arrives, an *assignment algorithm* must select a server $i \in M(j)$, and assign task j to it.

The *load* on server i at time t, denoted $L_i^A(t)$, is the sum of the weights of all the tasks running on server i at time t under assignment algorithm A.

Let σ be a sequence of task arrivals and departures, and let $|\sigma|$ be the time of the last arrival. Then the cost, $C_A(\sigma)$, of an assignment algorithm A on the sequence σ is defined as

$$C_A(\sigma) = \max_{0 \leq t \leq |\sigma|; i \in M} L_i^A(t).$$

An *on-line assignment algorithm* must assign an arriving task j at time $\tau(j)$ to a server in $M(j)$ knowing only $w(j)$, $M(j)$, and the past and current state of the servers—the decision is made without any knowledge about future arrivals or departures. The *optimal off-line assignment algorithm*, denoted OPT, assigns arriving tasks knowing the entire sequence of task arrivals and departures and does so in a way that minimizes cost.

The worst-case behavior of an on-line algorithm A is characterized by the *competitive ratio,* defined as the supremum over all sequences σ of the ratio $C_A(\sigma)/C_{\text{OPT}}(\sigma)$.

To characterize the average behavior of A, let $C_A(\mathcal{P})$ (resp., $C_{\text{OPT}}(\mathcal{P})$) be the expected cost of algorithm A (resp., OPT) on sequences σ generated by the distribution $\mathcal{P}$. The competitive ratio of an on-line algorithm, *A against distribution $\mathcal{P}$*, is defined as the ratio $C_A(\mathcal{P})/C_{\text{OPT}}(\mathcal{P})$.

Finally, the GREEDY algorithm is formally defined as follows.

ALGORITHM GREEDY. Upon arrival of a task j, assign it to the server in $M(j)$ with the current minimum load (ties are broken arbitrarily).

6.1.1. Permanent tasks. For permanent tasks, Azar, Naor, and Rom [9] have shown that the competitive ratio of the GREEDY algorithm is $\Theta(\log n)$ and that no algorithm can do better.

To bring this problem into our framework, we present our results for the case where for each task j the set of servers that can run it, $M(j)$, consists of $d \geq 2$ servers chosen uniformly at random (with replacement), the number of requests $|\sigma|$ equals n, and all weights are equal. Let $\mathcal{P}_d$ be the associated probability distribution on request sequences.

LEMMA 6.1. *With probability* $1 - O(1/n)$, $C_{\text{OPT}}(\mathcal{P}_d) = O(1)$.

Proof. We show that with high probability there is an assignment with cost 10 for the case $d = 2$. A fortiori the result is true for $d > 2$.

The problem can be reduced to showing that in a random n-by-n bipartite graph (U, V, E) where each node in U has two random edges to V, there is an assignment of value 10. Arbitrarily break U into 10 pieces of size $n/10$ each. We show that each of these pieces contains a perfect matching. By Hall's theorem, the probability that there is no such assignment is bounded by the probability that there is a set of size k in one of the pieces of U whose neighborhood has size less than k. Ipso facto, this probability is at most

$$10 \sum_{k \leq n/10} \binom{n}{k-1} \binom{n/10}{k} \left(\left(\frac{k-1}{n} \right)^2 \right)^k .$$

Using standard approximations to the binomial coefficients, this is at most

$$10 \sum_{k \le n/10} \left(\frac{en}{k-1}\right)^{k-1} \left(\frac{en}{10k}\right)^k \left(\left(\frac{k-1}{n}\right)^2\right)^k .$$

Finally, rewriting and simplifying yield

$$\frac{10}{n} \sum_{k \le n/10} \frac{k-1}{e} \left(\frac{e^2 n^2 (k-1)^2}{10k(k-1)n^2}\right)^k = \frac{10}{n} \sum_{k \le n/10} \frac{k-1}{e} \left(\frac{e^2}{10}\right)^k = O\left(\frac{1}{n}\right). \qquad \square$$

(A more delicate analysis [15] shows that the maximum load achieved by the off-line case is 2 with high probability, for $d \ge 2$ and $m \le 1.6n$.)

LEMMA 6.2. *With high probability,* $C_{\text{GREEDY}}(\mathcal{P}_d) = O(\log\log n/\log d)$

Proof. The proof follows immediately from Theorem 3.2. $\square$

Thus, we obtain the following theorem.

THEOREM 6.3. *The competitive ratio of the* GREEDY *algorithm against the distribution* $\mathcal{P}_d$ *is* $O(\log\log n/\log d)$*, and no algorithm can do better.*

Proof. The proof follows from Lemmas 6.1 and 6.2 and Corollary 3.6. $\square$

6.1.2. Temporary tasks. For temporary tasks, the results of Azar, Broder, and Karlin [6] and Azar et al. [8] showed that there is an algorithm with competitive ratio $\Theta(\sqrt{n})$ and that no algorithm can do better.

It is difficult to construct a natural distribution of task arrivals and departures. As an approximation, we consider the following stochastic process $\mathcal{S}$: First, n tasks arrive; for each task, the set of servers that can run it consists of $d \ge 2$ servers chosen uniformly at random (with replacement). Then the following repeats forever: a random task among those present departs and a random task arrives, which again may be served by any one of d random servers. Clearly, in such an infinite sequence, eventually there will be n tasks which can only be served by one server, and so for trivial reasons the competitive ratio for long enough sequences is 1. However, we can state a competitiveness result in the following way:

THEOREM 6.4. *Let* $L_A[t]$ *be the maximum load on any server at time* t*, for tasks arriving according to the stochastic process* $\mathcal{S}$*, and assigned using algorithm* A*, that is,* $L_A[t] = \max_{i \in M} L_i^A(t)$*. Then for any fixed* $t > 0$*, with high probability,*

$$\frac{L_{\text{GREEDY}}[t]}{L_{\text{OPT}}[t]} = O(\log\log n).$$

Proof. The proof follows from Lemma 6.1 and Theorem 4.1. $\square$

7. Experimental results. The bound proven in Theorem 3.2 for the $O(1)$ term in the formula for the upper bound on the maximum load is rather weak (≈ 8), so it might be the case that for practical values of n, the constant term dominates the $\ln\ln n/\ln d$ term. However, experiments seem to indicate that this is not the case, and in fact even for small values of n, the maximum load achieved with $d = 2$ is substantially smaller than the maximum load achieved with $d = 1$. For the values we considered, $256 \le n \le 16777216$, increasing d beyond 2 has only limited further effect. For each value of n and d we ran 100 experiments. The results are summarized in Table 7.1.

Table 7.1
Experimental maximum load ($m = n$).

n	$d = 1$	$d = 2$	$d = 3$	$d = 4$
2^8	**3** 1%	**2** 10%	**2** 84%	**2** 99%
	4 ...... 40%	**3** 90%	**3** 16%	**3** 1%
	5 ...... 41%			
	6 ...... 15%			
	7 ...... 3%			
2^{12}	**5** 12%	**3** 99%	**2** 12%	**2** 91%
	6 ...... 66%	**4** 1%	**3** 88%	**3** 9%
	7 ...... 17%			
	8 ...... 4%			
	9 ...... 1%			
2^{16}	**7** 48%	**3** 64%	**3** 100%	**2** 23%
	8 ...... 43%	**4** 36%		**3** 77%
	9 ...... 9%			
2^{20}	**8** 28%	**4** 100%	**3** 100%	**3** 100%
	9 ...... 61%			
	10 ...... 10%			
	13 ...... 1%			
2^{24}	**9** 12%	**4** 100%	**3** 100%	**3** 100%
	10 ...... 73%			
	11 ...... 13%			
	12 ...... 2%			

Acknowledgments. We wish to thank Martin Dyer, Alan Frieze, and Greg Nelson for several very useful discussions.

REFERENCES

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, *Parallel randomized load balancing*, in Proc. 27th Annual ACM Symposium on Theory of Computing, 1995, pp. 238–247.

[2] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley & Sons, New York, 1992.

[3] B. Awerbuch, Y. Azar, and S. Plotkin, *Throughput-competitive on-line routing*, in 34th Annual Symposium on Foundations of Computer Science, Palo Alto, CA, 1993, IEEE, Piscataway, NJ, pp. 32–40.

[4] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts, *Competitive routing of virtual circuits with unknown duration*, in Proc. ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, PA, 1994, pp. 321–327.

[5] Y. Azar, J. Aspnes, A. Fiat, S. Plotkin, and O. Waarts, *On-line routing of virtual circuits with applications to load balancing and machine scheduling*, J. Assoc. Comput. Mach., 44 (1997), pp. 486–504.

[6] Y. Azar, A. Z. Broder, and A. R. Karlin, *On-line load balancing*, Theoret. Comput. Sci., 130 (1994), pp. 73–84.

[7] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, *Balanced allocations*, in Proc. 26th Annual ACM Symposium on the Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 593–602.

[8] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts, *On-line load balancing of temporary tasks*, J. Algorithms, 22 (1997), pp. 93–110.

[9] Y. Azar, J. Naor, and R. Rom, *The competitiveness of on-line assignments*, J. Algorithms, 18 (1995), pp. 221–237.

[10] A. Z. Broder, A. M. Frieze, C. Lund, S. Phillips, and N. Reingold, *Balanced allocations*

for tree-like inputs, Inform. Process. Lett., 55 (1995), pp. 329–332.

[11] A. Z. Broder and A. R. Karlin, *Multilevel adaptive hashing*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, PA, 1990, pp. 43–53.

[12] A. Czumaj and V. Stemann, *Randomized allocation processes*, in Proc. 38th Annual IEEE Symposium on Foundations of Computer Science, 1997, Miami Beach, FL, 1997, pp. 194–203.

[13] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with O(1) worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538–544.

[15] A. M. Frieze, *Personal communication.* 1994.

[16] G. H. Gonnet, *Expected length of the longest probe sequence in hash code searching*, J. Assoc. Comput. Mach., 28 (1981), pp. 289–304.

[17] N. L. Johnson and S. Kotz, *Urn Models and Their Application*, John Wiley & Sons, New York, 1977.

[18] R. M. Karp, M. Luby, and F. Meyer auf der Heide, *Efficient PRAM simulation on a distributed memory machine*, in Proc. 24th Annual ACM Symposium on the Theory of Computing, Victoria, British Columbia, Canada, 1992, pp. 318–326.

[19] D. E. Knuth, *The Art of Computer Programming, Vol.* III*: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[20] V. F. Kolchin, B. A. Sevastyanov, and V. P. Chistyakov, *Random Allocations*, John Wiley & Sons, New York, 1978.

[21] P. D. Mackenzie, C. G. Plaxton, and R. Rajaraman, *On contention resolution protocols and associated probabilistic phenomena*, in Proc. 26th Annual ACM Symposium on the Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 153–162.

[22] A. W. Marshall and I. Olkin, *Inequalities: Theory of Majorization and Its Applications*, Academic Press, New York, 1979.

[23] M. Mitzenmacher, *Load balancing and density dependent jump Markov processes*, in Proc. 37th Annual Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 213–222.

[24] M. Mitzenmacher, *The Power of Two Choices in Randomized Load Balancing*, Ph.D. thesis, University of California, Berkeley, 1996.

[25] V. Stemann, *Parallel balanced allocations*, in Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures, Padua, Italy, 1996, pp. 261–269.

SIAM J. Comput. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 201–230

THE ALGORITHMIC ASPECTS OF UNCROWDED HYPERGRAPHS[*]

CLAUDIA BERTRAM-KRETZBERG[†] AND HANNO LEFMANN[†]

Abstract. We consider the problem of finding deterministically a large independent set of guaranteed size in a hypergraph on n vertices and with m edges. With respect to the Turán bound, the quality of our solutions is better for hypergraphs with not too many small cycles by a logarithmic factor in the input size. The algorithms are fast; they often have a running time of $O(m) + o(n^3)$. Indeed, the denser the hypergraphs are the closer the running times are to the linear times. For the first time, this gives for some combinatorial problems algorithmic solutions with state-of-the-art quality, solutions of which only the existence was known to date. In some cases, the corresponding upper bounds match the lower bounds up to constant factors. The involved concepts are uncrowded hypergraphs.

Key words. independence number, approximation algorithm

AMS subject classifications. 68Q25, 68R05, 68R10

PII. S0097539797323716

1. Introduction.

A fundamental problem in computer science and mathematics is to find a large independent set in an arbitrary graph [6], [16], [22]. Recall that for a graph $G = (V, E)$ with vertex set V and edge set $E \subseteq [V]^2$, a subset $I \subseteq V$ of the vertex set is called *independent* if the subgraph induced on I contains no edges $e \in E$, i.e., $E \cap [I]^2 = \emptyset$. The maximum cardinality of an independent set I is called the *independence number* $\alpha(G)$ of G. It is well known that finding an independent set of size $\alpha(G)$ in a graph G is an NP-hard problem, even for graphs with bounded maximum degree.

This suggests that we should look for approximation algorithms with guaranteed performance ratio, which is the quotient of the sizes of the optimal and the found solution in the worst case. The results of Arora et al. [10] on interactive proof systems show that with respect to polynomial time algorithms there is no constant performance ratio for the independent set problem for graphs on n vertices, indeed no ratio of $n^{1/4}$ unless $P = NP$; cf. the work of Bellare, Goldreich, and Sudan [13]. Recently, Håstad [28] showed that there is no performance ratio of $n^{1/2-\epsilon}$ unless $NP = P$ and no such ratio of $n^{1-\epsilon}$ unless $NP = coR$. With respect to polynomial time algorithms for triangle-free graphs with maximum degree Δ, a performance ratio of $O(\Delta/\ln\Delta)$ was given in [25] and [29] (see [45], [46]), and moreover, if they contain no complete subgraph K_l, $l \geq 4$, then a performance ratio of $O(\Delta/\ln\ln\Delta)$ is known; cf. [25], [26], [1]. Recently it was shown by Brandt [17] that the independence number of triangle-free graphs with minimum degree $\delta > n/3$ can be computed as fast as matrix multiplication in time $O(n^{2.376})$, while within the class of graphs with minimum degree $\delta > (1-\epsilon)n/4$, where $\epsilon > 0$, the problem is NP-hard.

For hypergraphs, the corresponding problem has been studied less; some papers are concerned with finding in parallel a maximal independent set; cf. [30] and [39].

[*]Received by the editors June 27, 1997; accepted for publication (in revised form) February 23, 1998; published electronically September 14, 1999. An extended abstract of this work appeared in Proc. 8th ACM-SIAM Symposium on Discrete Algorithms SODA'97. This research was supported by the Deutsche Forschungsgemeinschaft as part of the Collaborative Research Center "Computational Intelligence" (SFB 531).

http://www.siam.org/journals/sicomp/29-1/32371.html

[†]Lehrstuhl Informatik II, Universität Dortmund, Dortmund, D-44221, Germany (bertram@Ls2.informatik.uni-dortmund.de, lefmann@Ls2.informatik.uni-dortmund.de).

However, the size of a maximal independent set might be far off the size of a maximum independent set. Here we consider the problem of finding a large independent set in a hypergraph. Hypergraphs are important because many problems can be formulated in terms of them. A *hypergraph* $\mathcal{G} = (V, \mathcal{E})$ is given by a set V of vertices and a set $\mathcal{E}$ of edges (hyperedges), where each edge $E \in \mathcal{E}$ is a nonempty subset of V. A hypergraph $\mathcal{G} = (V, \mathcal{E})$ is called $(k+1)$-*uniform* if $\mathcal{E} \subseteq [V]^{k+1}$, i.e., each edge $E \in \mathcal{E}$ contains exactly $(k+1)$ vertices. If the hypergraph $\mathcal{G} = (V, \mathcal{E})$ is $(k+1)$-uniform, then

$$t(\mathcal{G}) = t = \left(\frac{(k+1) \cdot |\mathcal{E}|}{|V|}\right)^{1/k}$$

is the kth root of the *average degree* of $\mathcal{G}$. Similarly to the graph case, the *independence number* $\alpha(\mathcal{G})$ of a hypergraph $\mathcal{G} = (V, \mathcal{E})$ is defined as the maximum size of a subset $I \subseteq V$ which contains no edges $E \in \mathcal{E}$, i.e., there is no edge $E \in \mathcal{E}$ with $E \subseteq I$.

We give a general approximation strategy which has a performance ratio of $O(t/(\ln t)^{1/k})$ for $(k+1)$-uniform hypergraphs with average degree t^k and not too many small cycles. For random hypergraphs, no algorithm has a better performance ratio. The idea for doing this originates in a powerful result of Ajtai et al. [2] on *uncrowded hypergraphs.* These have the property that they contain no small cycles. We remark that derandomizing a probabilistic argument of Spencer [48] yields a performance ratio of $O(t)$ for arbitrary hypergraphs. We apply our approximation strategy to some combinatorial and graph problems for which only the existence of solutions of a certain quality was known to date. Our algorithms match these qualities and yield for these problems algorithmic solutions of state-of-the-art quality.

2. Uncrowded hypergraphs.

Turán's theorem for hypergraphs $\mathcal{G}$ gives a lower bound for the independence number $\alpha(\mathcal{G})$; cf. Spencer [48].

THEOREM 2.1 (see [48]). *Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph, $k \geq 1$, with average degree $t^k \geq 1$ and $|V| = n$. Then,*

$$\alpha(\mathcal{G}) \geq c_k \cdot \frac{n}{t}. \tag{2.1}$$

Moreover, an independent set of size at least $c_k \cdot n/t$ can be found in time $O(|\mathcal{E}|)$.

We remark that for $(k+1)$-uniform hypergraphs on n vertices with average degree $t^k < 1$, one can simply use a Greedy strategy to obtain in time $O(n)$ an independent set of size at least $c'_k \cdot n$; more simply, in an arbitrary way we add a few edges to the hypergraph to achieve in the resulting hypergraph that $1 \leq t^k < (k+1)$, and then we apply Theorem 2.1.

For completeness we give the algorithmic proof of (2.1).

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of vertices of $\mathcal{G}$. The existence of an independent set of size as guaranteed in (2.1) can be shown by picking vertices of V independently of each other at random with probability $p = 1/t$. The set of picked vertices will yield the independent set. For the algorithm we imitate this approach by using the method of conditional probabilities; cf. [9], [42]. To give a deterministic algorithm, we start by generalizing the probabilistic experiment. For $i = 1, 2, \ldots, n$ vertex v_i will be assigned a weight (probability) $p_i \in [0, 1]$. Define a potential $V(p_1, p_2, \ldots, p_n)$ by

$$V(p_1, p_2, \ldots, p_n) = \sum_{i=1}^{n} p_i - \sum_{E \in \mathcal{E}} \prod_{v_i \in E} p_i.$$

Note that in the corresponding random experiment, $\sum_{i=1}^{n} p_i$ is the expected number of picked vertices and $\sum_{E\in\mathcal{E}} \prod_{v_i\in E} p_i$ is the expected number of edges in the induced random hypergraph.

Now, for $i = 1, 2, \ldots, n$ in each step i we choose either $p_i = 0$ or $p_i = 1$ in order to maximize the value of the potential $V(p_1, p_2, \ldots, p_n)$. As $V(p_1, p_2, \ldots, p_n)$ is linear in each p_i, i.e., for $i = 1$

$$V(p_1, p_2, \ldots, p_n) = p_1 \cdot V(1, p_2, \ldots, p_n) + (1 - p_1) \cdot V(0, p_2, \ldots, p_n),$$

either $V(1, p_2, \ldots, p_n) \geq V(p_1, p_2, \ldots, p_n)$ or $V(0, p_2, \ldots, p_n) \geq V(p_1, p_2, \ldots, p_n)$. We take vertex v_1 for the independent set if $V(1, p_2, \ldots, p_n) > V(p_1, p_2, \ldots, p_n)$; otherwise we discard it. Doing this one after the other for $i = 1, 2, \ldots, n$, finally, each vertex v_i is assigned a weight $p_i \in \{0, 1\}$, $i = 1, 2, \ldots, n$.

Choosing in the beginning $p_1 = p_2 = \cdots = p_n = p$, we have

$$V(p, \ldots, p) = p \cdot n - p^{k+1} \cdot \frac{n \cdot t^k}{k+1},$$

which is maximal (taking the derivative) for $p = 1/t$, i.e., with this choice of p we have

$$V(p, \ldots, p) = \frac{k}{k+1} \cdot \frac{n}{t}. \tag{2.2}$$

By our strategy we finally obtain $V(p_1, p_2, \ldots, p_n) \geq V(p, \ldots, p)$. Let $V' = \{v_i \in V \mid p_i = 1\}$. By (2.2), and using that $p_i \geq 0$ for $i = 1, 2, \ldots, n$, we infer

$$|V'| = \sum_{i=1}^{n} p_i = V(p_1, \ldots, p_n) + \sum_{E\in\mathcal{E}} \prod_{v_i\in E} p_i \geq V(p, \ldots, p) + \sum_{E\in\mathcal{E}} \prod_{v_i\in E} p_i \geq \frac{k}{k+1} \cdot \frac{n}{t}.$$

We claim that V' is an independent set in $\mathcal{G}$. Suppose for contradiction that this is not the case, and let $E = \{v_{i_1}, v_{i_2}, \ldots, v_{i_{k+1}}\} \in \mathcal{E} \cap [V']^{k+1}$, where $i_1 < i_2 < \cdots < i_{k+1} = s$. Consider step s. Because we had chosen $p_s = 1$ in this step, we have

$$V(p_1, \ldots, p_{s-1}, 1, p_{s+1}, \ldots, p_n) > V(p_1, \ldots, p_n)\ . \tag{2.3}$$

However, as edge E came in in step s, we infer

$$V(p_1, \ldots, p_{s-1}, 1, p_{s+1}, \ldots, p_n) - V(p_1, \ldots, p_n) \leq (1 - p_s) - (1 - p_s) = 0,$$

which contradicts (2.3). Thus V' is an independent set in $\mathcal{G}$ of size at least $\frac{k}{k+1} \cdot \frac{n}{t}$, i.e., $\alpha(\mathcal{G}) \geq \frac{k}{k+1} \cdot \frac{n}{t}$.

During the algorithm each vertex and each edge is considered only a constant number of times; hence, with $t^k \geq 1$ the running time is $O(|\mathcal{E}|)$. □

For fixed integers $k \geq 1$, there are examples of $(k+1)$-uniform hypergraphs whose independence numbers match the lower bound (2.1) up to constant factors. Simply take n/x vertex disjoint copies of complete $(k+1)$-uniform hypergraphs on x vertices each where $t^k = \binom{x-1}{k}$. However, quite often the hypergraphs under consideration are in some sense *sparse*, and for these the general lower bound (2.1) can be improved, as can be seen in the following.

A *cycle* of length i in a $(k+1)$-uniform hypergraph $\mathcal{G} = (V, \mathcal{E})$ is a set of i pairwise distinct edges from $\mathcal{E}$ whose union contains at most $i \cdot k$ vertices. An *i-cycle* is a cycle of length i which does not contain any cycles of length $2, 3, \dots, i-1$. To specify the notion of "sparse" mentioned above, we define a hypergraph $\mathcal{G}$ to be *uncrowded* if it does not contain any 2-, 3-, or 4-cycles. For a vertex $v \in V$ let $d(v)$ denote its *degree*, i.e., the number of edges $E \in \mathcal{E}$ which contain v. Let $\Delta(\mathcal{G}) = \max_{v\in V} \{d(v)\}$ be the *maximum degree* of $\mathcal{G}$. For a subset $V' \subseteq V$ let $\mathcal{G}' = (V', \mathcal{E}')$ with $\mathcal{E}' = \mathcal{E} \cap [V']^{k+1}$ be on the V' *induced subhypergraph* of $\mathcal{G}$.

For uncrowded hypergraphs, the lower bound (2.1) was improved by the following powerful result of Ajtai et al. [2].

THEOREM 2.2 (see [2]). *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G}$ be a $(k+1)$-uniform hypergraph on n vertices. Assume that*

(i) *$\mathcal{G}$ is uncrowded, i.e., contains no 2-, 3-, or 4-cycles,*

(ii) *the maximum degree $\Delta(\mathcal{G})$ satisfies $\Delta(\mathcal{G}) \leq t^k$ where $t \geq t_0(k)$, and*

(iii) *$n \geq n_0(k, t)$.*

Then the independence number $\alpha(\mathcal{G})$ satisfies

$$\alpha(\mathcal{G}) \geq \frac{0.98}{e} \cdot 10^{-5/k} \cdot \frac{n}{t} \cdot (\ln t)^{1/k}. \tag{2.4}$$

Various applications of Theorem 2.2 have been found, including the disproof of Heilbronn's conjecture [32], results on Sidon sets [4], Steiner systems [44], [18], complexity theory [41], Ramsey numbers [3], geometric selection problems [37], Turán numbers for random graphs [31], and graph coloring problems [8], [36].

Indeed, for a certain range of the involved parameters k, n, t, inequality (2.4) is best possible up to constant factors, as a random hypergraph argument shows. Namely, for a fixed integer $k \geq 2$ consider a random $(k+1)$-uniform hypergraph on n vertices, where the edges are chosen independently at random with probability $p = (t/n)^k$, where $n \gg t \gg k$. Let $l = C \cdot n/t \cdot (\ln t)^{1/k}$, where $C > 0$ is a large enough constant. For a fixed l-element subset $I \subseteq V$ of the vertex set, the probability that I is an independent set is equal to

$$(1-p)^{\binom{l}{k+1}} \leq \exp\left\{-p \cdot \binom{l}{k+1}\right\} \leq \exp\left\{-\frac{p \cdot l^{k+1} \cdot (1-o(1))}{(k+1)!}\right\},$$

where we used the inequality $1 - x \leq \exp\{-x\} := e^{-x}$. Using the inequality $\binom{n}{l} \leq (e \cdot n/l)^l$, the expected number of independent sets of size l is at most

$$\begin{aligned}
&\binom{n}{l} \cdot \exp\left\{-\frac{p \cdot l^{k+1} \cdot (1-o(1))}{(k+1)!}\right\} \\
&\leq \left(\frac{e \cdot n}{l} \cdot \exp\left\{-\frac{p \cdot l^k \cdot (1-o(1))}{(k+1)!}\right\}\right)^l \\
&\leq \left(\frac{e}{C} \cdot \frac{t}{(\ln t)^{1/k}} \cdot \exp\left\{-\frac{C^k \cdot (1-o(1)) \cdot \ln t}{(k+1)!}\right\}\right)^l \\
&\leq \left(\frac{e}{C} \cdot \exp\left\{\ln t - \frac{\ln \ln t}{k} - \frac{C^k \cdot (1-o(1)) \cdot \ln t}{(k+1)!}\right\}\right)^l \\
&< \frac{1}{5}
\end{aligned}$$

for $C^k \geq 2 \cdot (k+1)!$ and t large enough. However, the expected number of cycles of length $i \leq 4$ in the random hypergraph is at most

$$n^{ik} \cdot p^i = t^{ik} \leq t^{4k}.$$

By Markov's inequality there exists a hypergraph $\mathcal{G}$ on n vertices with at most

$$5 \cdot p \cdot \binom{n}{k+1} \le 5 \cdot n \cdot \frac{t^k}{(k+1)!}$$

edges which contains no independent set of size $5 \cdot C \cdot n/t \cdot (\ln t)^{1/k}$ and with at most $15 \cdot t^{4k}$ cycles of length at most 4. Note that for each induced subhypergraph $\mathcal{G}'$ of $\mathcal{G}$ we have $\alpha(\mathcal{G}') \le \alpha(\mathcal{G})$. Now, omitting one vertex from each cycle of length at most 4 gives a $(k+1)$-uniform uncrowded hypergraph on $n - 15 \cdot t^{4k} = (1 - o(1)) \cdot n$ vertices with average degree at most $5/k! \cdot t^k$ and with independence number at most $5 \cdot C \cdot n/t \cdot (\ln t)^{1/k}$, provided $t^{4k} = o(n)$. Hence, for many hypergraphs, inequality (2.4) is best possible up to constant factors.

Notice that in (2.4) we gain a logarithmic factor if we compare it to Turán's lower bound (2.1). This additional logarithmic factor is of interest because in some applications one can use Theorem 2.2 to prove lower bounds which, with respect to the corresponding applications, asymptotically match the upper bounds.

We remark that recently Łuczak and Szymańska [39] gave a randomized parallel algorithm with polylogarithmic running time to find a maximal (not necessarily maximum) independent set in hypergraphs without 2-cycles.

Fundia [21] gave a polynomial time algorithm for uncrowded hypergraphs, which finds an independent set of size asymptotically at least as guaranteed by inequality (2.4).

THEOREM 2.3 (see [21]). *Let $k \ge 2$ be a fixed integer. Let $\mathcal{G}$ be an uncrowded $(k+1)$-uniform hypergraph on n vertices with average degree t^k, where $k^4 \cdot t^{4k} < n$ and $k < t$. Then, one can find in time $O\left(n^3 \cdot t^{6k} \cdot \ln t\right)$ an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.*

Notice that in uncrowded $(k+1)$-uniform hypergraphs $\mathcal{G} = (V, \mathcal{E})$ with $|V| = n$, two distinct edges have at most one vertex in common; hence $|\mathcal{E}| \le \binom{n}{2}/\binom{k+1}{2}$ and $t = O(n^{1/k})$.

We remark that in Theorem 2.3 the condition $k^4 \cdot t^{4k} < n$ is not that important (cf. [8]) if we ignore for the moment the algorithmic aspects. Namely, if $k^4 \cdot t^{4k} \ge n$, then we set

$$N = \left\lceil \frac{k^4 \cdot t^{4k} + 1}{n} \right\rceil$$

and we form a new hypergraph $\mathcal{H} = (V', \mathcal{E}')$ by taking N vertex disjoint copies of $\mathcal{G}$. Clearly, $|V'| = N \cdot n$ and $|\mathcal{E}'| = N \cdot |\mathcal{E}|$; thus $\mathcal{G}$ and $\mathcal{H}$ have the same average degree. However, $\mathcal{H}$ fulfills the assumptions of Theorem 2.3, and we obtain in time $O((N \cdot n)^3 \cdot t^{6k} \cdot \ln t)$ an independent set of size $\Omega(N \cdot n/t \cdot (\ln t)^{1/k})$. Then, restricting the independent set to one copy of $\mathcal{G}$ yields the desired result, i.e., an independent set of size $\Omega(n/t \cdot (\ln t)^{1/k})$. We have in uncrowded hypergraphs $t = O(n^{1/k})$; hence, we infer for the running time $O((N \cdot n)^3 \cdot t^{6k} \cdot \ln t) = O(t^{18k} \cdot \ln t) = O(n^{18} \cdot \ln n)$, i.e., polynomial running time for fixed k.

Contrary to our considerations of blowing up the hypergraph, i.e., taking copies, we will work on small subhypergraphs of $\mathcal{G}$ to keep the running times of the corresponding algorithms small; cf. Kortsarz and Peleg [33]. Throughout this paper, k will always be a fixed integer with $k \ge 2$ and t^k, the average degree, will always be an increasing function of n, the number of vertices of the hypergraph, i.e., $t = t(n) \to \infty$ with $n \to \infty$.

Theorem 2.3 does not seem to be applicable to many hypergraphs. In general, the hypergraphs under consideration are not uncrowded and have quite a lot of 2-, 3-, or 4-cycles. However, as we will see, in several cases the trick is to choose at random an appropriate small subhypergraph, which turns out to be "nearly" uncrowded, i.e., has only a few small cycles. After deleting these cycles one can apply Theorem 2.3. Indeed, seemingly against the intuition, one chooses vertices with a probability $p = t^{-1+\epsilon}$ for some $\epsilon > 0$, which is a little bigger than that which one would usually take, i.e., $p = t^{-1}$. But this gives the improvement by a logarithmic factor.

In the following we state our main results. First, we improve Theorem 2.3 as follows.

THEOREM 2.4. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G}$ be an uncrowded $(k+1)$-uniform hypergraph on n vertices with average degree t^k and $t \to \infty$ with $n \to \infty$. Then, for each fixed $\delta > 0$, one can find in time*

$$O\left(n \cdot t^k + \frac{n^3}{t^{3-\delta}}\right) \tag{2.5}$$

an independent set in $\mathcal{G}$ of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.

The parameter $\delta > 0$ can be chosen to be small. Thus, the more edges the hypergraph has, the closer the running time is to the linear one, $O(n \cdot t^k)$. However, as $t = O(n^{1/k})$ for $k \geq 3$ and $\delta > 0$, the time bound is always $O(|\mathcal{E}| + n^{3-3/k+\delta}) = O(n^{3-3/k+\delta})$.

The proof of Theorem 2.4 shows that with the factor t^δ, $0 < \delta < 1$, in the running time there comes a factor $(\delta/(6k+4))^{1/k}$ in the quality of the solution. Hence, for $k \geq 3$, dropping the running time from $O(n^3/t^{3-\delta})$ to $O(n^3/t^{3-\delta/2})$, i.e., by the factor $t^{\delta/2}$, means that we lose in the guaranteed quality the factor $2^{1/k}$.

COROLLARY 2.5. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G}$ be an uncrowded $(k+1)$-uniform hypergraph on n vertices with average degree at most t^k and $t \to \infty$ with $n \to \infty$. Then, for each fixed $\delta > 0$, one can find in time*

$$O\left(n \cdot t^k + \frac{n^3}{t^{3-\delta}}\right) \tag{2.6}$$

an independent set in $\mathcal{G}$ of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.

Proof. If the average degree $t(\mathcal{G})^k$ of $\mathcal{G}$ satisfies $t(\mathcal{G})^k \leq t^k/\ln t$, then by Theorem 2.1 we find in time $O(n \cdot t(\mathcal{G})^k) = O(n \cdot t^k)$ an independent set of size

$$\Omega\left(\frac{n}{t(\mathcal{G})}\right) = \Omega\left(\frac{n}{t} \cdot (\ln t)^{1/k}\right).$$

Otherwise, if $t^k \geq t(\mathcal{G})^k > t^k/\ln t$, then we apply Theorem 2.4 with $\delta' = \delta/2$ and we obtain in time

$$O\left(n \cdot t(\mathcal{G})^k + \frac{n^3}{t(\mathcal{G})^{3-\delta'}}\right) = O\left(n \cdot t^k + \frac{n^3}{t^{3-\delta'}} \cdot (\ln t)^{(3-\delta')/k}\right) = O\left(n \cdot t^k + \frac{n^3}{t^{3-\delta}}\right)$$

an independent set of size at least $\Omega(n/t(\mathcal{G}) \cdot (\ln t(\mathcal{G}))^{1/k}) = \Omega(n/t \cdot (\ln t)^{1/k})$. Here we used that the function $f(t) = (\ln t)^{1/k}/t$ is decreasing for $t \geq 2$. $\square$

It turns out that the 2-cycles are important. For a hypergraph $\mathcal{G} = (V, \mathcal{E})$, a 2-cycle $\{E_1, E_2\}$ with $E_1, E_2 \in \mathcal{E}$ is called *$(2, j)$-cycle* if $|E_1 \cap E_2| = j$. Let $s_{2,j}(\mathcal{G})$ denote the number of $(2, j)$-cycles in $\mathcal{G}$.

The next theorem yields an algorithmic version of the existence result which was proved by Duke, Lefmann, and Rödl [18]. Corollary 2.8 gives an algorithmic solution of a conjecture of Spencer [49].

THEOREM 2.6. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph on n vertices with average degree t^k, where $t \to \infty$ with $n \to \infty$. If the $(2, j)$-cycles satisfy*

$$s_{2,j}(\mathcal{G}) \leq c \cdot n \cdot t^{2k+1-j-\gamma}$$

for $j = 2, 3, \ldots, k$ and some constants $c, \gamma > 0$, then one can find for every fixed $\delta > 0$ in time $O(n \cdot t^k + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}) + n^3/t^{3-\delta})$ an independent set of size at least

$$C(k, \gamma, \delta, c) \cdot \frac{n}{t} \cdot (\ln t)^{1/k} .$$

COROLLARY 2.7. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph on n vertices with average degree at most t^k, where $t \to \infty$ with $n \to \infty$. If the $(2, j)$-cycles satisfy*

$$s_{2,j}(\mathcal{G}) \leq c \cdot n \cdot t^{2k+1-j-\gamma}$$

for $j = 2, 3, \ldots, k$ and some constants $c, \gamma > 0$, then one can find for every fixed $\delta > 0$ in time $O(n \cdot t^k + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}) + n^3/t^{3-\delta})$ an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.

Proof. If the average degree $t(\mathcal{G})^k$ of $\mathcal{G}$ satisfies $t^k \geq t(\mathcal{G})^k \geq t^k / \ln t$, then the $(2, j)$-cycles satisfy

$$s_{2,j}(\mathcal{G}) \leq c \cdot n \cdot t^{2k+1-j-\gamma} \leq c \cdot n \cdot t(\mathcal{G})^{2k+1-j-\gamma/2}$$

for t large. Thus, the assumptions of Theorem 2.6 are fulfilled, and we obtain for every fixed $\delta > 0$ in time $O(n \cdot t^k + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}) + n^3/t^{3-\delta})$ an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$. Otherwise, if $t(\mathcal{G})^k < t^k / \ln t$, we apply Theorem 2.1 and obtain in time $O(n \cdot t^k)$ an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$. $\square$

As an immediate consequence of Corollary 2.7 we have the following.

COROLLARY 2.8. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph on n vertices with average degree at most t^k, where $t \to \infty$ with $n \to \infty$. If $\mathcal{G}$ does not contain any 2-cycles, then one can find for every fixed $\delta > 0$ in time $O(n \cdot t^k + n^3/t^{3-\delta})$, an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.*

3. Choosing small subhypergraphs.

For proving our results, we use the idea of choosing small subhypergraphs on which we control certain parameters such as the number of vertices, edges, or cycles. The control on the small subhypergraphs reflects a probabilistic approach.

LEMMA 3.1. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph with $s_{2,j}(\mathcal{G})$ many $(2, j)$-cycles, $j = 2, 3, \ldots, k$. For every p with $0 \leq p \leq 1$, one can find in time $O(|V| + |\mathcal{E}| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}))$ an induced subhypergraph $\mathcal{H} = (V', \mathcal{E}')$ such that for $j = 2, 3, \ldots, k$ it holds*

$$|V'| \geq p/3 \cdot |V| \quad \textit{and} \quad |\mathcal{E}'| \leq 3 \cdot p^{k+1} \cdot |\mathcal{E}| \quad \textit{and} \quad s_{2,j}(\mathcal{H}) \leq 3 \cdot (k-1) \cdot p^{2k+2-j} \cdot s_{2,j}(\mathcal{G}).$$

Proof. By inspecting each two-element subset which is contained in an edge $E \in \mathcal{E}$, we obtain for each two-element set $\{x, y\}$ all edges $E \in \mathcal{E}$ with $\{x, y\} \subset E$. This can be done in time $O(|\mathcal{E}|)$. Then, the sets C_j, which contain the vertex sets of all $(2,j)$-cycles in $\mathcal{G}$, $j = 2, 3, \ldots, k$, can be constructed in time $O(|V| + |\mathcal{E}| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}))$.

Let $V = \{v_1, v_2, \ldots, v_n\}$. Every vertex v_i will be assigned a weight $p_i \in [0,1]$, $i = 1, 2, \ldots, n$. Define a potential $V(p_1, p_2, \ldots, p_n)$ by

$$V(p_1, p_2, \ldots, p_n) = 3^{pn/3} \cdot \prod_{i=1}^{n} \left(1 - \frac{2}{3} \cdot p_i\right) + \frac{\sum_{E \in \mathcal{E}} \prod_{v_i \in E} p_i}{3 \cdot p^{k+1} \cdot |\mathcal{E}|} + \sum_{j=2}^{k} \frac{\sum_{C \in C_j} \prod_{v_i \in C} p_i}{3 \cdot (k-1) \cdot p^{2k+2-j} \cdot |C_j|} \,.$$

If $pn < 6$, any two vertices will do since they do not yield an edge. Hence, we assume that $pn \geq 6$. With $p_1 = \cdots = p_n = p$ in the beginning, and using $1 - x \leq e^{-x}$, we infer

$$V(p, \ldots, p) = 3^{pn/3} \cdot \left(1 - \frac{2}{3} \cdot p\right)^n + \frac{p^{k+1} \cdot |\mathcal{E}|}{3 \cdot p^{k+1} \cdot |\mathcal{E}|} + \sum_{j=2}^{k} \frac{p^{2k+2-j} \cdot s_j(\mathcal{G})}{3 \cdot (k-1) \cdot p^{2k+2-j} \cdot s_{2,j}(\mathcal{G})}$$
$$\leq \left(\frac{3}{e^2}\right)^{pn/3} + \frac{2}{3} < 1 \,.$$

The potential $V(p_1, p_2, \ldots, p_n)$ is linear in each p_i. As in the algorithmic argument for proving Theorem 2.1 step by step, we decide which choice of p_i, either $p_i = 0$ or $p_i = 1$, minimizes the current value of $V(p_1, p_2, \ldots, p_n)$. We put vertex v_1 in V' iff $V(1, p_2, \ldots, p_n) \leq V(0, p_2, \ldots, p_n)$. Doing this for all vertices $v_1, v_2, \ldots, v_n$ yields the desired set $V' = \{v_i \in V \mid p_i = 1\}$. We always choose the value of $p_i \in \{0,1\}$ to minimize the value of the potential, i.e., finally, we have $V(p_1, p_2, \ldots, p_n) < 1$, too. All summands in $V(p_1, p_2, \ldots, p_n)$ are nonnegative. If the chosen subhypergraph $\mathcal{H} = (V', \mathcal{E}')$ with $\mathcal{E}' = \mathcal{E} \cap [V']^{k+1}$ violated one of the desired properties, then the corresponding summand would be bigger than 1, which is not possible by our strategy.

The computation of $V(p, p, \ldots, p)$ in the beginning can be done in time $O(|V| + |\mathcal{E}| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}))$, and updating all the potentials $V(p_1, \ldots, p_n)$ during the algorithm can be done in the same time. □

For a hypergraph $\mathcal{G}$, let $\mu_i(\mathcal{G})$, $i = 3, 4$, denote the number of i-cycles in $\mathcal{G}$.

LEMMA 3.2. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph on $|V| = n$ vertices with average degree t^k. If $\mathcal{G}$ does not contain any 2-cycles, then one can find in time $O(n + n \cdot t^k)$ an induced subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$ with $|V'| \geq |V|/2$ and*

$$\mu_i(\mathcal{G}) \leq c_i(k) \cdot n \cdot t^{(i-1)k} \tag{3.1}$$

for $i = 3, 4$. The sets C_i of i-cycles, $i = 3, 4$, can be computed in time $O(n + n \cdot t^{(i-1)k}) = O(n + n \cdot t^{3k})$.

Proof. For each edge $E \in \mathcal{E}$ we mark in time $O(|\mathcal{E}|)$ all two-element subsets which are contained in E by "E." Then, in time $O(1)$ we can decide whether a two-element set is contained in some edge $E \in \mathcal{E}$.

First we discard all vertices $v \in V$ with degree $d(v) > 2(k+1) \cdot t^k =: \Delta$ and all edges incident with such vertices v. This can be done in time $O(n + n \cdot t^k)$. The resulting induced hypergraph $\mathcal{G}' = (V', \mathcal{E}')$ of $\mathcal{G}$ has $|V'| \geq n/2$ vertices. As $\mathcal{G}$ contains no 2-cycles for each two distinct vertices $x, y \in V'$, there exists at most one edge which contains both x and y. The maximum degree of $\mathcal{G}'$ is at most Δ; hence

$$\mu_3(\mathcal{G}') \leq \frac{1}{3} \cdot n \cdot \binom{\Delta}{2} \cdot k^2 \leq c_3(k) \cdot n \cdot t^{2k},$$
$$\mu_4(\mathcal{G}') \leq \frac{1}{4} \cdot n \cdot \binom{\Delta}{2} \cdot k \cdot \Delta \cdot k^2 \leq c_4(k) \cdot n \cdot t^{3k}.$$

To determine the set C_3 of all 3-cycles in $\mathcal{G}'$, we consider each vertex $v \in V_0$ and all pairs $\{E_1, E_2\} \in [\mathcal{E}']^2$ of edges with $v \in E_1$ and $v \in E_2$. This can be done in time $O(n \cdot \Delta^2)$. Then, for each two vertices $x \in E_1 \setminus \{v\}$ and $y \in E_2 \setminus \{v\}$, we test in time $O(1)$ whether there exists an edge $E \in \mathcal{E}'$ with $x, y \in E$. Thus, determining the set C_3 of 3-cycles in $\mathcal{G}'$ can be done in time $O(n + n \cdot t^{2k})$. Similarly, one can determine the set C_4 of 4-cycles in $\mathcal{G}'$ in time $O(n + n \cdot t^{3k})$. ∎

LEMMA 3.3. *Let $k \geq 2$ be a fixed integer. Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph on $|V| = n$ vertices with average degree t^k, where $\mathcal{G}$ contains no 2-cycles. For every p with $0 \leq p \leq 1$, one can find in time $O(n + n \cdot t^{3k})$ an induced subhypergraph $\mathcal{H} = (V', \mathcal{E}')$ of $\mathcal{G}$ such that*

$$\begin{aligned} &|V'| \geq p/6 \cdot |V|, \\ &|\mathcal{E}'| \leq 3 \cdot p^{k+1} \cdot |\mathcal{E}|, \\ &\mu_i(\mathcal{H}) \leq 6 \cdot p^{ik} \cdot c_i(k) \cdot n \cdot t^{(i-1)k} \quad \textit{for } i = 3, 4. \end{aligned} \tag{3.2}$$

Proof. By Lemma 3.2, we find in $\mathcal{G}$ in time $O(n + n \cdot t^k)$ an induced subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$ with $V' = \{v_1, v_2, \ldots, v_l\}$ and $|V'| \geq |V|/2$, which contains at most $c_i(k) \cdot n \cdot t^{(i-1)k}$ i-cycles. The sets C'_i of i-cycles in $\mathcal{G}'$, $i = 3, 4$, can be computed in time $O(n + n \cdot t^{3k})$. Then we apply to $\mathcal{G}'$ the same derandomization technique as in the proof of Lemma 3.1 using the potential

$$V(p_1, p_2, \ldots, p_l) = 3^{pn/6} \prod_{v_i \in V'} \left(1 - \frac{2}{3} \cdot p_i\right) + \frac{\sum_{E \in \mathcal{E}'} \prod_{v_i \in E} p_i}{3 \cdot p^{k+1} \cdot |\mathcal{E}|}$$
$$+ \sum_{j=3}^{4} \frac{\sum_{C \in C'_j} \prod_{v_i \in C} p_i}{6 \cdot p^{jk} \cdot c_j(k) \cdot n \cdot t^{(j-1)k}},$$

and we obtain in time $O(n + n \cdot t^{3k})$ an induced subhypergraph fulfilling all three properties (3.2). ∎

Similarly, one can show the following.

LEMMA 3.4. *Let $\mathcal{G} = (V, \mathcal{E})$ be a $(k+1)$-uniform hypergraph with $|V| = n$. For every p with $0 \leq p \leq 1$, one can find in time $O(n + |\mathcal{E}|)$ an induced subhypergraph $\mathcal{H} = (V', \mathcal{E}')$ of $\mathcal{G}$ such that*

$$|V'| \geq p/2 \cdot |V| \quad \textit{and} \quad |\mathcal{E}'| \leq 2 \cdot p^{k+1} \cdot |\mathcal{E}| \,.$$

Proof. We apply the same derandomization technique as in the proof of Lemma 3.1 by using the potential

$$V(p_1, p_2, \ldots, p_n) = 2^{pn/2} \cdot \prod_{i=1}^{n} \left(1 - \frac{p_i}{2}\right) + \frac{\sum_{E \in \mathcal{E}} \prod_{v_i \in E} p_i}{2 \cdot p^{k+1} \cdot |\mathcal{E}|}. \qquad ∎$$

4. Avoiding 2-cycles.

Here we will give the proof of Theorem 2.4.

Proof. Let $\mathcal{G}$ be an uncrowded $(k+1)$-uniform hypergraph on n vertices with average degree t^k; thus $t = O(n^{1/k})$. The idea is to choose a small induced subhypergraph $\mathcal{G}_0$ of $\mathcal{G}$ to which we apply Theorem 2.3.

First, we apply Lemma 3.4 to $\mathcal{G}$ with $p = t^{-1+\epsilon}$, where $\epsilon = \min\{\frac{\delta}{6k+4}, \frac{k-1}{4k}\}$, and we obtain in time $O(n \cdot t^k)$ an induced subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_0)$ of $\mathcal{G}$ with

$$|V_0| \geq p/2 \cdot |V| \quad \text{and} \quad |\mathcal{E}_0| \leq 2 \cdot p^{k+1} \cdot |\mathcal{E}| \, .$$

Notice that if we had chosen the probability to be $p = 1/t$, then the resulting subhypergraph would only have n/t vertices, but our aim is to find an independent set of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.

The hypergraph $\mathcal{G}_0$ has average degree

$$t(\mathcal{G}_0)^k \leq t_0^k := 4 \cdot (p \cdot t)^k \, .$$

If $t(\mathcal{G}_0) \leq p \cdot t/(\ln(p \cdot t))^{1/k}$, then we apply Theorem 2.1 and we obtain in time $O(p \cdot n + p^{k+1} \cdot |\mathcal{E}|)$ an independent set of size

$$\Omega\left(\frac{p \cdot n}{t(\mathcal{G}_0)}\right) = \Omega\left(\frac{p \cdot n}{p \cdot t} \cdot (\ln(p \cdot t))^{1/k}\right) = \Omega\left(\frac{n}{t} \cdot (\ln t)^{1/k}\right).$$

Otherwise, let $t(\mathcal{G}_0) > p \cdot t/(\ln(p \cdot t))^{1/k}$. We have $t^\epsilon/(\epsilon \cdot \ln t)^{1/k} > k$, as k, ϵ are constants and $t \to \infty$ with $n \to \infty$. With $t = O(n^{1/k})$ and $\epsilon \leq (k-1)/(4k)$, we infer

$$p/2 \cdot n = n/2 \cdot t^{-1+\epsilon} > k^4 \cdot 4^4 \cdot t^{4k\epsilon} = k^4 \cdot t_0^{4k};$$

hence the assumptions of Theorem 2.3 are fulfilled and we apply it to the hypergraph $\mathcal{G}_0$. By omitting some more vertices we can assume without loss of generality (w.l.o.g.) that $|V_0| = p/2 \cdot n$. As $\epsilon \leq \delta/(6k+4)$, we obtain in time

$$O\left((p \cdot n)^3 \cdot (p \cdot t)^{6k} \cdot \ln t\right) = O\left(\frac{n^3}{t^{3-3\epsilon-6k\epsilon}} \cdot \ln t\right) = O\left(\frac{n^3}{t^{3-\delta}}\right)$$

an independent set of size at least

$$\Omega\left(\frac{p \cdot n}{t_0} \cdot (\ln t_0)^{1/k}\right) = \Omega\left(\frac{p \cdot n}{p \cdot t} \cdot (\ln(p \cdot t))^{1/k}\right) = \Omega\left(\frac{n}{t} \cdot (\ln t^\epsilon)^{1/k}\right) = \Omega\left(\frac{n}{t} \cdot (\ln t)^{1/k}\right)$$

as $\epsilon > 0$ is a constant. Here, we used that the function $f(t) = (\ln t)^{1/k}/t$ is decreasing for $t \geq 2$. The algorithm has the desired running time $O(n \cdot t^k + n^3/t^{3-\delta})$. $\square$

Indeed, provided an algorithm does not require any assumptions on the mutual growth of k, t, n, we have the following *Meta-Algorithm*.

Let $k \geq 2$ be a fixed integer. Let $\mathcal{G}$ be an uncrowded $(k+1)$-uniform hypergraph on n vertices with average degree t^k and $t \to \infty$ with $n \to \infty$. Let $\delta > 0$ be any fixed real number.

If one can find in time $f(n,t)$ an independent set in $\mathcal{G}$ of size $\Omega(n/t \cdot (\ln t)^{1/k})$, then one can find in time $O(f(n/t^{1-\delta}, t^\delta) + n \cdot t^k)$ an independent set in $\mathcal{G}$ of size at least $\Omega(n/t \cdot (\ln t)^{1/k})$.

Now we come to the proof of Theorem 2.6.

Proof. Let the $(2,j)$-cycles of $\mathcal{G}$ satisfy $s_{2,j}(\mathcal{G}) \leq c \cdot n \cdot t^{2k+1-j-\gamma}$ for $j = 2, 3, \ldots, k$ and constants $c, \gamma > 0$. Set

$$p = t^{-1+\epsilon}, \tag{4.1}$$

where

$$\epsilon = \min \left\{ \frac{\gamma}{2k}, \frac{1}{3k+2} \right\}.$$

By Lemma 3.1, we obtain in time

$$O\left(|V| + |\mathcal{E}| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}) \right) \tag{4.2}$$

an induced subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_0)$ of $\mathcal{G}$ such that, possibly after omitting some more vertices, we have

$$\begin{aligned} |V_0| &= p/3 \cdot |V| \text{ , and} \\ |\mathcal{E}_0| &\leq 3 \cdot p^{k+1} \cdot |\mathcal{E}| \text{ , and} \\ s_{2,j}(\mathcal{G}_0) &\leq 3 \cdot (k-1) \cdot p^{2k+2-j} \cdot s_{2,j}(\mathcal{G}) \qquad \text{for } j = 2, 3, \ldots, k. \end{aligned}$$

With (4.1) for $t^\epsilon \geq 2$, i.e., $t \geq t_0(\epsilon)$, we have for $j = 2, 3, \ldots, k$ that

$$p^{2k+2-j} \cdot n \cdot t^{2k+1-j-\gamma} \geq 2 \cdot p^{2k+2-j-1} \cdot n \cdot t^{2k+1-j-1-\gamma}.$$

Hence, since $\epsilon \leq \gamma/2k$ and $t \geq t_0(k, \gamma)$, i.e., $t^\gamma \geq (36 \cdot c \cdot (k-1))^{2k}$, we have

$$\begin{aligned} \sum_{j=2}^{k} s_{2,j}(\mathcal{G}_0) &\leq \sum_{j=2}^{k} 3 \cdot (k-1) \cdot p^{2k+2-j} \cdot s_{2,j}(\mathcal{G}) \\ &\leq 6 \cdot (k-1) \cdot c \cdot p^{2k} \cdot n \cdot t^{2k-1-\gamma} \\ &\leq 6 \cdot (k-1) \cdot c \cdot n \cdot t^{-1-\gamma+2k\epsilon} \\ &\leq 36 \cdot (k-1) \cdot c \cdot t^{-\frac{\gamma}{2k}} \cdot (p \cdot n/6) \\ &\leq p \cdot n/6; \end{aligned}$$

thus,

$$|V_0| \geq 2 \cdot \sum_{j=2}^{k} s_{2,j}(\mathcal{G}_0) \ . \tag{4.3}$$

The sets of $(2,j)$-cycles in $\mathcal{G}_0$ can be constructed in time $O(p \cdot n + |\mathcal{E}_0| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}_0))$. By losing at most half of the vertices, cf. (4.3), we delete in $\mathcal{G}_0$ in time

$$\begin{aligned} O\left(p \cdot n + p^{k+1} \cdot |\mathcal{E}| + \sum_{j=2}^{k} s_{2,j}(\mathcal{G}) \cdot p^{2k+2-j} \right) &= O\left(\frac{n}{t^{1-\epsilon}} + \frac{n}{t^{1-(k+1)\epsilon}} + \frac{n}{t^{1+\gamma-2k\epsilon}} \right) \\ &= o(n) \end{aligned}$$

for $\epsilon \leq 1/(k+2)$ and $\epsilon \leq \gamma/(2k)$ one vertex from each 2-cycle. By deleting possibly more vertices, we obtain an induced subhypergraph $\mathcal{G}_1 = (V_1, \mathcal{E}_1)$ of $\mathcal{G}_0$ with

$$|V_1| = p \cdot n/6 \quad \text{and} \quad |\mathcal{E}_1| \leq 3 \cdot p^{k+1} \cdot |\mathcal{E}|$$

such that $\mathcal{G}_1$ no longer contains any 2-cycles and we have

$$t(\mathcal{G}_1)^k \leq \frac{(k+1) \cdot |\mathcal{E}_1|}{p \cdot n/6} = 18 \cdot (p \cdot t)^k \ . \tag{4.4}$$

Next we apply Lemma 3.3 to the hypergraph $\mathcal{G}_1$ with

$$p_1 = \left(\frac{1}{p \cdot t}\right)^{1-\epsilon_1} = t^{-\epsilon(1-\epsilon_1)},$$

where $\epsilon_1 = (k-1)/(4k)$, and we obtain a subhypergraph $\mathcal{G}_2 = (V_2, \mathcal{E}_2)$ of $\mathcal{G}_1$, which, after possibly deleting more vertices, satisfies

$$\begin{aligned} &|V_2| = p_1 p \cdot n/36, \\ &|\mathcal{E}_2| \leq 3 \cdot p_1^{k+1} \cdot |\mathcal{E}_1| \leq 9 \cdot (p_1 p)^{k+1} \cdot |\mathcal{E}| \end{aligned}$$

and, by (3.2), (4.4), for $i = 3, 4$ also fulfills

$$\mu_i(\mathcal{G}_2) \leq 6 \cdot p_1^{ik} \cdot c_i(k) \cdot \frac{p \cdot n}{6} \cdot t(\mathcal{G}_1)^{(i-1)k} \leq c_i(k) \cdot p_1^{ik} \cdot p \cdot n \cdot (18 \cdot p^k \cdot t^k)^{i-1}.$$

This can be done in time $O(p \cdot n + p \cdot n \cdot (p \cdot t)^{3k}) = O(n \cdot t^{-1+\epsilon(3k+1)}) = o(n)$. We claim that

$$|V_2| \geq 4 \cdot (\mu_3(\mathcal{G}_2) + \mu_4(\mathcal{G}_2)). \tag{4.5}$$

Namely, by our choice of p and p_1, i.e., $p = t^{-1+\epsilon}$ and $p_1 = t^{-\epsilon(1-\epsilon_1)}$, we have

$$\begin{aligned} &\frac{p_1 p \cdot n}{36} \geq 8 \cdot c_i(k) \cdot p_1^{ik} \cdot pn \cdot (18 \cdot p^k \cdot t^k)^{i-1} \\ \Longleftrightarrow\ &1 \geq 288 \cdot c_i(k) \cdot p_1^{ik-1} \cdot (18 \cdot p^k \cdot t^k)^{i-1} \\ \Longleftrightarrow\ &1 \geq 288 \cdot 18^{i-1} \cdot c_i(k) \cdot t^{-\epsilon(k-1)+\epsilon\epsilon_1(ik-1)}. \end{aligned}$$

Thus, for $\epsilon_1 < (k-1)/(4k-1)$ and t large enough, (4.5) holds and $\mathcal{G}_2$ contains at least four times as many vertices as 3- and 4-cycles. Moreover, the average degree of $\mathcal{G}_2$ satisfies $t(\mathcal{G}_2)^k = O((p_1 p \cdot t)^k)$. We omit in $\mathcal{G}_2$ all vertices of degree bigger than $2 \cdot (k+1) \cdot t(\mathcal{G}_2)^k$; hence losing at most $p_1 p \cdot n/72$ vertices, and then we determine in the resulting induced subhypergraph $\mathcal{G}_2'$ of $\mathcal{G}_2$ the sets of 3- and 4-cycles in time

$$O(p_1 p \cdot n + p_1 p \cdot n \cdot t(\mathcal{G}_2)^{3k}) = O(p_1 p \cdot n \cdot (p_1 p \cdot t)^{3k}) = O(n \cdot t^{-1+\epsilon\epsilon_1(3k+1)}) = o(n)$$

for $\epsilon\epsilon_1 < 1/(3k+1)$. By deleting in time $o(n)$ one vertex from each 3- or 4-cycle from $\mathcal{G}_2'$, we obtain a subhypergraph $\mathcal{G}_3 = (V_3, \mathcal{E}_3)$ of $\mathcal{G}_2$, which does not contain any 2-, 3-, or 4-cycles, i.e., $\mathcal{G}_3$ is uncrowded and satisfies w.l.o.g.

$$|V_3| = |V_2|/2 = p_1 p \cdot n/144 \quad \text{and} \quad |\mathcal{E}_3| \leq 9 \cdot (p_1 p)^{k+1} \cdot |\mathcal{E}|;$$

thus,

$$t(\mathcal{G}_3)^k \leq \frac{(k+1) \cdot 9 \cdot (p_1 p)^{k+1} \cdot |\mathcal{E}|}{p_1 p \cdot n/144} = 1296 \cdot (p_1 p)^k \cdot t^k.$$

To the hypergraph $\mathcal{G}_3$ we apply Corollary 2.5, and for fixed $\delta > 0$ we obtain in time

$$O\left(n \cdot t(\mathcal{G}_3)^k + \frac{(p_1 p \cdot n)^3}{(p_1 p \cdot t)^{3-\delta}}\right) = O\left(\frac{n^3}{t^{3-\delta}}\right)$$

an independent set in $\mathcal{G}_3$, and hence in $\mathcal{G}$, of size at least

$$\Omega\left(\frac{p_1 p \cdot n}{p_1 p \cdot t} \cdot (\ln(p_1 p \cdot t))^{1/k}\right) = \Omega\left(\frac{n}{t} \cdot (\ln t)^{1/k}\right)$$

as desired. The overall running time is given by (4.2). $\Box$

5. Applications. In this section, we give applications of our results to some combinatorial problems. Our arguments are guided by the probabilistic existence proofs. However, for computational reasons additional ideas are required.

DEFINITION 5.1. *Let X be a set. A triple system $\mathcal{F} \subseteq [X]^3$ is called a* partial Steiner triple system *if for any two vertices $x_1, x_2 \in X$ there is at most one set $F \in \mathcal{F}$ with $\{x_1, x_2\} \subset F$.*

Hence, every partial Steiner triple system $\mathcal{F}$ is nothing else than a 3-uniform hypergraph which contains no 2-cycles, i.e., $|\mathcal{F}| \leq 1/3 \cdot \binom{n}{2}$.

The next result is essentially a reformulation of Corollary 2.8.

COROLLARY 5.2. *Let X be an n-element set, and let $\mathcal{F} \subset [X]^3$ be a partial Steiner triple system.*

Then, one can find in time $O(n^2)$ an independent set $I \subseteq X$ with

$$|I| \geq c \cdot \sqrt{n \cdot \ln n}. \tag{5.1}$$

We remark that the existence of an independence set of size as in (5.1) was shown first by Phelps and Rödl [40].

Proof. Since $t^2 = O(n)$, by Corollary 2.8 we find such an independent set of size at least $\Omega(\sqrt{n \cdot \ln n})$ in time $O(n^2)$. $\Box$

Next we consider a generalization of partial Steiner triple systems.

DEFINITION 5.3. *Let h, k be positive integers with $h \leq k$. Let X be an n-element set. A family $\mathcal{F} \subseteq [X]^{k+1}$ of $(k+1)$-element subsets is called* $(n, k+1, h)$-Steiner system *if for any two distinct sets $F_1, F_2 \in \mathcal{F}$, it holds that $|F_1 \cap F_2| < h$.*

In an $(n, k+1, h)$-Steiner system $\mathcal{F}$ on X, each h-element subset of X is contained in at most one set $F \in \mathcal{F}$; hence, $|\mathcal{F}| \leq \binom{n}{h}/\binom{k+1}{h}$. Note that an $(n, 3, 2)$-Steiner system is a partial Steiner triple system. In every $(n, k+1, 1)$-Steiner system, $\mathcal{F}$ distinct sets $F, F' \in \mathcal{F}$ are disjoint; hence, $|\mathcal{F}| = O(n)$ and $\alpha(\mathcal{F}) \geq \frac{k}{k+1} \cdot n$, and such an independent set can be found easily in time $O(n)$. For values $h \geq 2$, we have the following result.

THEOREM 5.4. *Let X be an n-element set, and let $\mathcal{F} \subset [X]^{k+1}$ be an $(n, k+1, h)$-Steiner system where $h \geq 2$. Then, for every $\delta > 0$, one can find in time $O(n^h + n^{3-3\cdot(h-1)/k+\delta} + n^{2h-3-(h-1)/k+\delta})$ an independent set $I \subseteq X$ with*

$$|I| \geq c \cdot n^{\frac{k-h+1}{k}} \cdot (\ln n)^{\frac{1}{k}}\ . \tag{5.2}$$

The existence of an independent set of size at least as given in (5.2) was proved by Rödl and Siňajová [44]. By using the local lemma of Lovász (cf. [5]), they also showed that there exists an $(n, k+1, h)$-Steiner system $\mathcal{F}$ with independence number bounded from above by $C \cdot n^{(k-h+1)/k} \cdot (\ln n)^{1/k}$ for some constant $C > 0$. Hence in general, for $|\mathcal{F}| = \Theta(n^h)$ one cannot expect, up to constant factors, any better bound than (5.2).

Concerning the running times in Theorem 5.4, we have $O(n^2)$ for $h = k = 2$ and $O(n^{3-3/k+\delta})$ for $h = 2$ and $k \geq 3$. For $h \geq 3$ we have the time bound $O(n^{2h-3-(h-1)/k+\delta})$.

Proof. Let $\mathcal{F}$ be an $(n, k+1, h)$-Steiner system on X. The corresponding hypergraph $(X, \mathcal{F})$ is $(k+1)$-uniform with $|\mathcal{F}| \leq \binom{n}{h}/\binom{k+1}{h}$ edges and has average degree

$$t(\mathcal{F})^k \leq t^k = \frac{(k+1) \cdot \binom{n}{h}}{\binom{k+1}{h} \cdot n} \leq c_1 \cdot n^{h-1}.$$

First, we consider the $(2, j)$-cycles in $\mathcal{G} = (X, \mathcal{F})$. For each set $F \in \mathcal{F}$, we take a j-element subset $S \subset F$ and try to extend it to a set $F' \in \mathcal{F}$. This can be done in at most $O(n^{h-j})$ ways. Thus,

$$s_{2,j}(\mathcal{G}) \leq c_j \cdot |\mathcal{F}| \cdot n^{h-j} \leq c_{2,j} \cdot n^{2h-j}$$

for $j = 2, 3, \ldots, h-1$. Moreover, constructing the set of $(2, j)$-cycles can be done in time

$$O\left(|\mathcal{F}| \cdot \binom{|V|}{h-j}\right).$$

Also for every subset $X' \subseteq X$, the induced subsystem $\mathcal{G}' = (X', \mathcal{F}')$ with $\mathcal{F}' = \mathcal{F} \cap [X']^{k+1}$ satisfies

$$s_{2,j}(\mathcal{G}') \leq c'_{2,j} \cdot |\mathcal{F}'| \cdot \binom{|X'|}{h-j}$$

for $j = 2, 3, \ldots, h-1$, and the set of $(2, j)$-cycles in $\mathcal{G}'$ can be determined in time $O(|\mathcal{F}'| \cdot |X'|^{h-j})$.

Given $\delta > 0$, by Lemma 3.4 for $p = n^{-1/k+\epsilon}$ with $0 < \epsilon \leq \min\{1/k, \delta/(k+h-1)\}$, we select in time $O(n^h)$ an induced subsystem $(X_0, \mathcal{F}_0)$ of $(X, \mathcal{F})$ which satisfies, after possibly omitting more vertices,

$$|X_0| = p/2 \cdot |X| \text{ and } |\mathcal{F}_0| \leq 2 \cdot p^{k+1} \cdot |\mathcal{F}|.$$

Then

$$t(\mathcal{F}_0)^k \leq t_0^k = 4 \cdot (p \cdot t)^k .$$

For sequences a_n, b_n, $n = 1, 2, \ldots$, let $a_n \ll b_n$ if $a_n/b_n \to 0$ with $n \to \infty$. To apply Corollary 2.7, we want to have for some constant $\gamma > 0$ that

$$p^{k+1} \cdot |\mathcal{F}| \cdot \binom{p \cdot n}{h-j} \ll p \cdot n \cdot (t_0)^{2k+1-j-\gamma}$$
$$\iff p^{-(k-h+1)+\gamma} \cdot n^{-(j-1)\cdot\frac{k-h+1}{k}+\gamma\cdot\frac{h-1}{k}} \ll 1$$
$$\iff n^{-(j-2)\cdot\frac{k-h+1}{k}-\epsilon\cdot(k-h+1)+\gamma\cdot\frac{h-2+\epsilon k}{k}} \ll 1,$$

and the last inequality holds since $j \geq 2$ for $\gamma < (\epsilon \cdot (k-h+1))/((h-2)/k+\epsilon)$. By Corollary 2.7 we obtain an independent set in $\mathcal{F}_0$; hence in $\mathcal{F}$, of size at least

$$\Omega\left(\frac{p \cdot n}{t_0} \cdot (\ln t_0)^{\frac{1}{k}}\right) = \Omega\left(\frac{p \cdot n}{p \cdot t} \cdot (\ln(p \cdot t))^{\frac{1}{k}}\right) = \Omega\left(n^{\frac{k-h+1}{k}} \cdot (\ln n)^{\frac{1}{k}}\right).$$

The time for doing this is for any $\delta, \delta^* > 0$ and $\delta^* \cdot \frac{h-1}{k} < \delta$,

$$\begin{aligned}
&O\left(p \cdot n + p^{k+1} \cdot n^h + \sum_{j=2}^{h-1} p^{k+1} \cdot n^h \cdot (p \cdot n)^{h-j} + \frac{(p \cdot n)^3}{(t_0)^{3-\delta^*}}\right) \\
&= O\left(p^{k+1} \cdot n^h \cdot (p \cdot n)^{h-2} + n^{3-(3-\delta^*)\cdot\frac{h-1}{k}}\right) \\
&= O\left(n^{2h-3-\frac{h-1}{k}+\epsilon(k+h-1)} + n^{3-3(h-1)/k+\delta}\right) \\
&= O\left(n^{2h-3-\frac{h-1}{k}+\delta} + n^{3-3(h-1)/k+\delta}\right).
\end{aligned}$$

Thus, the total running time is $O(n^h + n^{3-3(h-1)/k+\delta} + n^{2h-3-(h-1)/k+\delta})$ for each given $\delta > 0$. $\square$

The following problem was considered for the first time by Erdös and Guy [20]: Determine the maximum cardinality of a subset X of the $n \times n$-grid such that all mutual Euclidean distances between distinct points of X are distinct. By a Greedy-type argument it was shown in [20] that for every $\epsilon > 0$, such a set X with $|X| \geq c_1 \cdot n^{2/3-\epsilon}$ exists. By a probabilistic argument, this lower bound was improved by Thiele [50] to $|X| \geq c_2 \cdot n^{2/3}/(\ln n)^{1/3}$. Subsequently, in [37] the existence of such a set X with $|X| \geq c_3 \cdot n^{2/3}$ was shown. The problem of how to achieve this nonconstructive lower bound remained open. Here we will give such an algorithm. Such problems are related to problems arising from measuring distances using sonar signals, cf. [19], [23], and [24].

THEOREM 5.5. *One can determine in time $O(n^6 \cdot \ln n)$ a subset X of the $n \times n$-grid such that the distances between distinct points of X are mutually distinct and*

$$|X| \geq c \cdot n^{2/3}. \tag{5.3}$$

We remark that by results from number theory, currently one can show only the upper bound $|X| \leq c \cdot n/(\ln n)^{1/4}$, cf. [20].

Proof. Let $G_n = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ denote the vertex set of the $n \times n$-grid. We form a (nonuniform) hypergraph $\mathcal{G} = (V, \mathcal{E}_3 \cup \mathcal{E}_4)$ on $V = G_n$ with $\mathcal{E}_3 \subseteq [V]^3$ and $\mathcal{E}_4 \subseteq [V]^4$ as follows. Let $d(x, y)$ denote the Euclidean distance between x and y, i.e., $d(x, y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ for $x = (x_1, x_2)$ and $y = (y_1, y_2)$. For pairwise distinct vertices $x, y, z \in V$ we form a *three-element edge* $E = \{x, y, z\} \in \mathcal{E}_3$ iff $d(x, y) = d(x, z)$. Moreover, for pairwise distinct vertices $x^1, x^2, x^3, x^4 \in V$ we form a *four-element edge* $E = \{x^1, x^2, x^3, x^4\} \in \mathcal{E}_4$ iff $d(x^1, x^2) = d(x^3, x^4)$.

Our strategy will be to find a large independent set $I \subseteq V$ in the hypergraph $\mathcal{G}$. Clearly, an independent set $I \subseteq V$ is (in the grid) a set with mutual distinct distances. We will find this independent set $I \subseteq V$ again by picking vertices at random and using derandomization. The usual process of choosing a subhypergraph and controlling the number of $(2, 2)$-cycles right from the beginning would result in a running time of $O\left(n^{8+\frac{c}{\ln \ln n}}\right)$. Therefore, some additional ideas are involved here to get the time bound $O(n^6 \cdot \ln n)$. To achieve this, we will control the $(2, 2)$-cycles at a later stage. We remark that $|\mathcal{E}_3 \cup \mathcal{E}_4| = O(n^6 \cdot \ln n)$, as will be seen later. Thus, our algorithm will be linear in the size of $\mathcal{G}$.

In a preprocessing, we compute and store the sets S_d of all solutions of the diophantine equation $x^2 + y^2 = d$ for $d = 1, 2, \ldots, 2(n-1)^2$ with integers x and y. For a fixed value of d this can be done in time $O(\sqrt{d})$ by simply inserting

$x = -\lfloor\sqrt{d}\rfloor, -\lfloor\sqrt{d}\rfloor + 1, \ldots, \lfloor\sqrt{d}\rfloor$ in $x^2 + y^2 = d$ and solving this for y. This preprocessing can be done in time

$$O\left(\sum_{d=1}^{2(n-1)^2} \sqrt{d}\right) = O\left(n^3\right).$$

For positive integers d, let $r_2(d)$ denote the number of solutions of the equation $x^2 + y^2 = d$ within the integers. By a result of Wigert (cf. [27]), we have

$$\max\{r_2(d) \mid d = 1, 2, \ldots, n\} \leq n^{\frac{c'}{\ln \ln n}} \tag{5.4}$$

for some constant $c' > 0$; hence,

$$|S_d| = r_2(d) \leq n^{\frac{c'}{\ln \ln n}}$$

for $d = 1, 2, \ldots, 2(n-1)^2$. We will use the following identity by Ramanujan [43]:

$$\sum_{d=1}^{m} (r_2(d))^2 = \Theta(m \cdot \ln m). \tag{5.5}$$

Given the $n \times n$ grid, we first construct the sets $\mathcal{E}_3$ and $\mathcal{E}_4$ of edges. For $d = 1, 2, \ldots, 2(n-1)^2$ and for each vertex $x_0 \in V$, there are at most $|S_d|$ vertices $x^1, x^2, \ldots, x^l \in V$ with $d(x_0, x^j)^2 = d$, $j = 1, 2, \ldots, l$, and all these vertices can easily be determined in time $O(r_2(d))$. Then, $\{x_0, x^i, x^j\}$, $1 \leq i < j \leq l$ yields a 3-element edge in $\mathcal{E}_3$. Hence, using (5.5) the set $\mathcal{E}_3$ of 3-element edges can be constructed in time

$$O\left(n^2 \cdot \sum_{d=1}^{2(n-1)^2} \binom{r_2(d)}{2}\right) = O\left(n^2 \cdot \sum_{d=1}^{2(n-1)^2} (r_2(d))^2\right) = O\left(n^4 \cdot \ln n\right),$$

and we also infer that

$$|\mathcal{E}_3| \leq c_3 \cdot n^4 \cdot \ln n. \tag{5.6}$$

In a similar fashion we construct the set $\mathcal{E}_4$ of 4-element edges. Namely, we pick two distinct vertices $x, y \in V$ and consider the sets $A = \{z \in V \mid d(x,z)^2 = d\}$ and $B = \{w \in V \mid d(y,w)^2 = d\}$ for $d = 1, 2, \ldots, 2(n-1)^2$. Then, $\{x, y, z, w\}$ with $z \in A$, $w \in B$, and $z \neq w$ yields a 4-element edge $E \in \mathcal{E}_4$. Using (5.5), this can be done in time

$$O\left(n^2 \cdot n^2 \cdot \sum_{d=1}^{2(n-1)^2} (r_2(d))^2\right) = O\left(n^6 \cdot \ln n\right),$$

and also

$$|\mathcal{E}_4| \leq c_4 \cdot n^6 \cdot \ln n. \tag{5.7}$$

We want to find again a small, but large enough, uncrowded induced subhypergraph in $\mathcal{G}$. The running time will depend essentially on the number of $(2,j)$-cycles, which we have to handle and which are determined by pairs of distinct edges from $\mathcal{E}_4$. Consider only the 4-element edges in $\mathcal{G}$, i.e., $\mathcal{G}|\mathcal{E}_4 = (V, \mathcal{E}_4)$. We show how to construct the sets $C_{2,2}$ and $C_{2,3}$ of $(2,2)$- and $(2,3)$-cycles in $\mathcal{G}|\mathcal{E}_4$, $j = 2, 3$. Fix an integer d with $1 \le d \le 2(n-1)^2$. For an edge $E = \{x^1, x^2, x^3, x^4\} \in \mathcal{E}_4$ with $d(x^1, x^2) = d(x^3, x^4)$ and distinct vertices $x^i, x^j \in E$ and $y, z \in V \setminus E$ with $d(x^i, x^j) = d(y, z)$ or with $d(x^i, y) = d(x^j, z)$, a pair $\{\{x^1, x^2, x^3, x^4\}, \{x^i, x^j, y, z\}\}$ yields a $(2,2)$-cycle. Hence, using (5.4) and (5.7) the time to construct the set $C_{2,2}$ of $(2,2)$-cycles is given by

$$\begin{aligned} (5.8) \qquad & O\left(|\mathcal{E}_4| \cdot n^2 \cdot \max\{r_2(d) \mid 1 \le d \le 2(n-1)^2\}\right) \\ & = O\left(n^{8+\frac{c_1}{\ln \ln n}} \cdot \ln n\right) = O\left(n^{8+\frac{c}{\ln \ln n}}\right) \end{aligned}$$

for some constant $c > 0$. Thus, for some constant $c_{2,2} > 0$ we have

$$s_{2,2}(\mathcal{G}|\mathcal{E}_4) \le c_{2,2} \cdot n^{8+\frac{c}{\ln \ln n}}.$$

To construct the set $C_{2,3}$ of $(2,3)$-cycles in $\mathcal{G}|\mathcal{E}_4$, we pick an edge $E \in \mathcal{E}_4$ and a 3-element subset $S \subset E$, which determines a constant number of distances. Then S can be extended in at most $O(\max\{r_2(d) \mid 1 \le d \le 2(n-1)^2\})$ ways to an edge $E' \in \mathcal{E}_4$ with $S \subset E'$. Thus, we can construct the set $C_{2,3}$ in $\mathcal{G}|\mathcal{E}_4$ in time

$$(5.9) \qquad O(|\mathcal{E}_4| \cdot \max\{r_2(d) \mid 1 \le d \le 2(n-1)^2\}),$$

and also for some constant $c_{2,3} > 0$, we have

$$s_{2,3}(\mathcal{G}|\mathcal{E}_4) \le c'_{2,3} \cdot n^6 \cdot \ln n \cdot n^{\frac{c_1}{\ln \ln}} \le c_{2,3} \cdot n^{6+\frac{c}{\ln \ln n}}.$$

However, we do not construct the sets $C_{2,j}$ of $(2,j)$-cycles right from the beginning. We first determine a small, but big enough, induced subhypergraph, where we only control the number of vertices and edges, and in this we will construct the sets of $(2,j)$-cycles, $j = 2, 3$.

From the considerations leading to (5.8) we infer that every induced subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$ of the hypergraph $\mathcal{G}|\mathcal{E}_4 = (V, \mathcal{E}_4)$, consisting only of the 4-element edges, satisfies

$$\begin{aligned} (5.10) \qquad s_{2,2}(\mathcal{G}') & = O\left(|\mathcal{E}^{'}| \cdot |V^{'}| \cdot \max\{r_2(d) \mid 1 \le d \le 2(n-1)^2\}\right) \\ & = O\left(|\mathcal{E}^{'}| \cdot |V^{'}| \cdot n^{\frac{c}{\ln \ln n}}\right). \end{aligned}$$

Similarly, with the consideration before (5.9) we infer that

$$(5.11) \qquad s_{2,3}(\mathcal{G}') = O\left(|\mathcal{E}'| \cdot n^{\frac{c}{\ln \ln n}}\right).$$

Also, the set of $(2,2)$-cycles in $\mathcal{G}'$ can be determined in time $O(|\mathcal{E}^{'}| \cdot |V^{'}| \cdot n^{\frac{c}{\ln \ln n}})$, and the set of $(2,3)$-cycles can be constructed in time $O(|\mathcal{E}'| \cdot n^{\frac{c}{\ln \ln n}})$.

Lemma 5.6. *Let p be a real number, $0 \le p \le 1$. Then, one can construct in time $O(|V| + |\mathcal{E}_3| + |\mathcal{E}_4|)$ a subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_3^0 \cup \mathcal{E}_4^0)$ of $\mathcal{G} = (V, \mathcal{E}_3 \cup \mathcal{E}_4)$ with $\mathcal{E}_i^0 \subseteq \mathcal{E}_i$, $i = 3, 4$, such that $|V_0| = p/3 \cdot |V|$ and $|\mathcal{E}_3^0| \le 3 \cdot p^3 \cdot |\mathcal{E}_3|$ and $|\mathcal{E}_4^0| \le 3 \cdot p^4 \cdot |\mathcal{E}_4|$.*

Proof. The proof follows earlier arguments; cf. Lemma 3.4. For $V = \{v_1, v_2, \ldots, v_m\}$, we use the potential

$$V(p_1, p_2, \ldots, p_m) = 3^{pm/3} \cdot \prod_{i=1}^{m} \left(1 - \frac{2}{3} \cdot p_i\right) + \frac{\sum_{E \in \mathcal{E}_3} \prod_{v_i \in E} p_i}{3 \cdot p^3 \cdot |\mathcal{E}_3|} + \frac{\sum_{E \in \mathcal{E}_4} \prod_{v_i \in E} p_i}{3 \cdot p^4 \cdot |\mathcal{E}_4|}.$$

We obtain in time $O(|V| + |\mathcal{E}_3| + |\mathcal{E}_4|)$ a subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_3^0 \cup \mathcal{E}_4^0)$ of $\mathcal{G} = (V, \mathcal{E}_3 \cup \mathcal{E}_4)$ with $|V_0| \geq p/3 \cdot |V|$ and $|\mathcal{E}_3^0| \leq 3 \cdot p^3 \cdot |\mathcal{E}_3|$ and $|\mathcal{E}_4^0| \leq 3 \cdot p^4 \cdot |\mathcal{E}_4|$. By possibly omitting more vertices we get $|V_0| = p/3 \cdot |V|$. □

In time $O(|V| + |\mathcal{E}_3| + |\mathcal{E}_4|) = O(n^6 \cdot \ln n)$ we apply Lemma 5.6 to the hypergraph $\mathcal{G}$ with

$$p = n^{-2/3+\epsilon}, \tag{5.12}$$

where $0 < \epsilon < 2/3$, and we obtain a subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_3^0 \cup \mathcal{E}_4^0)$ of $\mathcal{G}$ with

$$|V_0| = p \cdot n^2/3, \quad |\mathcal{E}_3^0| \leq 3 \cdot p^3 \cdot |\mathcal{E}_3|, \quad |\mathcal{E}_4^0| \leq 3 \cdot p^4 \cdot |\mathcal{E}_4|. \tag{5.13}$$

We did not control the number $s_{2,j}(\mathcal{G}_0|\mathcal{E}_4^0)$ of $(2,j)$-cycles, $j = 2, 3$, in $\mathcal{G}_0|\mathcal{E}_4^0$. However, by (5.11) we know for $\epsilon < 4/15$ that

$$\begin{aligned} s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) &\leq c'_{2,2} \cdot |\mathcal{E}_4^0| \cdot |V_0| \cdot n^{\frac{c}{\ln \ln n}} \\ &\leq c^*_{2,2} \cdot p^4 \cdot n^6 \cdot \ln n \cdot p \cdot n^2 \cdot n^{\frac{c}{\ln \ln n}} \\ &= c^*_{2,2} \cdot p^5 \cdot n^{8+\frac{c}{\ln \ln n}} \cdot \ln n \\ &\leq c^*_{2,2} \cdot n^{14/3+5\epsilon+\frac{c}{\ln \ln n}} \cdot \ln n \\ &= o(n^6 \cdot \ln n). \end{aligned} \tag{5.14}$$

Moreover, for $\epsilon < 2/3$ we infer from (5.11) that

$$\begin{aligned} s_{2,3}(\mathcal{G}_0, \mathcal{E}_4^0) &\leq c'_{2,3} \cdot |\mathcal{E}_4^0| \cdot n^{\frac{c}{\ln \ln n}} \\ &\leq c^*_{2,3} \cdot p^4 \cdot n^{6+\frac{c}{\ln \ln n}} \cdot \ln n \\ &\leq c^*_{2,3} \cdot n^{10/3+4\epsilon+\frac{c}{\ln \ln n}} \cdot \ln n \\ &= o(n^6 \cdot \ln n). \end{aligned} \tag{5.15}$$

The value p is chosen in (5.12) such that the assumptions of Corollary 2.7 are fulfilled for the hypergraph $\mathcal{G}_0|\mathcal{E}_4^0$, i.e., we choose for p that minimal value for which the following two conditions hold for some constants $c, \gamma > 0$:

$$\begin{aligned} s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) &\leq c \cdot p \cdot n^2 \cdot t(\mathcal{G}_0|\mathcal{E}_4^0)^{5-\gamma}, \\ s_{2,3}(\mathcal{G}_0|\mathcal{E}_4^0) &\leq c \cdot p \cdot n^2 \cdot t(\mathcal{G}_0|\mathcal{E}_4^0)^{4-\gamma}. \end{aligned}$$

We proceed similarly to the proof of Theorem 2.6. Namely, we use the following lemma, whose proof is along the lines of former statements.

LEMMA 5.7. *For every p_1 with $0 \leq p_1 \leq 1$, one can construct in time*

$$O\left(|V_0| + |\mathcal{E}_3^0| + |\mathcal{E}_4^0| + s_{2,3}(\mathcal{G}|\mathcal{E}_4^0) + s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0)\right)$$

a subhypergraph $\mathcal{G}_1 = (V_1, \mathcal{E}_3^1 \cup \mathcal{E}_4^1)$ of $\mathcal{G}_0 = (V_0, \mathcal{E}_3^0 \cup \mathcal{E}_4^0)$ with $\mathcal{E}_i^1 \subseteq \mathcal{E}_i^0$, $i = 3, 4$, such that

$$|V_1| = p_1/5 \cdot |V_0|\,, \quad |\mathcal{E}_3^1| \leq 5 \cdot p_1^3 \cdot |\mathcal{E}_3^0|\,, \quad |\mathcal{E}_4^1| \leq 5 \cdot p_1^4 \cdot |\mathcal{E}_4^0|\,, \quad s_{2,3}(\mathcal{G}_1|\mathcal{E}_4^1) \leq 5 \cdot p_1^5 \cdot s_{2,3}\left(\mathcal{G}_0|\mathcal{E}_4^0\right),$$

and $s_{2,2}(\mathcal{G}_1|\mathcal{E}_4^1) \leq 5 \cdot p_1^6 \cdot s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^1)$.

Before applying Lemma 5.7, we estimate the corresponding running time, namely, using (5.6), (5.7), (5.12), (5.13), (5.14), and (5.15) we obtain

$$O\left(|V_0| + |\mathcal{E}_3^0| + |\mathcal{E}_4^0| + s_{2,3}(\mathcal{G}|\mathcal{E}_4^0) + s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0)\right) = o(n^6 \cdot \ln n).$$

Let

$$p_1 = \left(\frac{(\ln n)^{1/3}}{p \cdot t}\right)^{1-\epsilon_1} = \left(\frac{1}{n^{2/3+\epsilon}}\right)^{1-\epsilon_1},$$

where

$$\epsilon_1 = \frac{\epsilon}{5\epsilon + 11/3} . \tag{5.16}$$

By Lemma 5.7 we obtain in time $o(n^6 \cdot \ln n)$ a subhypergraph $\mathcal{G}_1 = (V_1, \mathcal{E}_3^1 \cup \mathcal{E}_4^1)$ of $\mathcal{G}_0$ with $\mathcal{E}_i^1 \subseteq \mathcal{E}_i^0$, $i = 3, 4$, where

$$|V_1| = \frac{p_1}{5} \cdot |V_0| = \frac{p_1 p}{15} \cdot n^2 = \frac{1}{15} \cdot n^{2/3+2\epsilon_1/3+\epsilon\epsilon_1}$$

and

$$|\mathcal{E}_3^1| \leq 5 \cdot p_1^3 \cdot |\mathcal{E}_3^0| \leq 15 \cdot p_1^3 \cdot p^3 \cdot |\mathcal{E}_3| \leq 15 \cdot c_3 \cdot n^{2\epsilon_1+3\epsilon\epsilon_1} \cdot \ln n = o(|V_1|) .$$

For the number of $(2,3)$-cycles, we have using (5.15) that

$$\begin{aligned} s_{2,3}\left(\mathcal{G}_1|\mathcal{E}_4^1\right) &\leq 5 \cdot p_1^5 \cdot s_{2,3}\left(\mathcal{G}_0|\mathcal{E}_4^0\right) \\ &\leq 15 \cdot c_{2,3} \cdot p_1^5 \cdot n^{10/3+4\epsilon+\frac{c}{\ln\ln n}} \cdot \ln n \\ &\leq 15 \cdot c_{2,3} \cdot n^{-\epsilon+10\epsilon_1/3+5\epsilon\epsilon_1+\frac{c}{\ln\ln n}} \cdot \ln n \\ &= o(|V_1|). \end{aligned}$$

Finally, using (5.14) we obtain for the number of $(2,2)$-cycles that

$$\begin{aligned} s_{2,2}\left(\mathcal{G}_1|\mathcal{E}_4^1\right) &\leq 5 \cdot p_1^6 \cdot s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) \\ &\leq 5 \cdot c_{2,2}^* \cdot p_1^6 \cdot n^{14/3+5\epsilon+\frac{c}{\ln\ln n}} \cdot \ln n \\ &\leq 5 \cdot c_{2,2}^* \cdot n^{2/3-\epsilon+4\epsilon_1+6\epsilon\epsilon_1+\frac{c}{\ln\ln n}} \cdot \ln n \\ &= o(|V_1|). \end{aligned}$$

Summarizing, we have $|\mathcal{E}_3^1| + s_{2,3}(\mathcal{G}_1|\mathcal{E}_4^1) + s_{2,2}(\mathcal{G}_1|\mathcal{E}_4^1) = o(|V_1|)$. We delete from $\mathcal{G}_1$ one vertex from each 3-element edge $E \in \mathcal{E}_3^1$ and from each $(2,2)$- or $(2,3)$-cycle in $\mathcal{G}_1|\mathcal{E}_4^0$, and, possibly deleting more vertices, we obtain a 4-uniform induced subhypergraph $\mathcal{G}_2 = (V_2, \mathcal{E}_4^2)$ of $\mathcal{G}_1$ with

$$|V_2| = \frac{|V_1|}{2} = \frac{p_1 p}{30} \cdot n^2$$

and

$$|\mathcal{E}_4^2| \leq 15 \cdot c_4 \cdot (p_1 p)^4 \cdot n^6 \cdot \ln n,$$

and $\mathcal{G}_2$ no longer contains any 2-cycles. By Corollary 2.8 we obtain an independent set in $\mathcal{G}_2$, and hence in $\mathcal{G}$, of size

$$\begin{aligned} \Omega\left(\frac{p_1 p \cdot n^2}{p_1 p \cdot t} \cdot (\ln(p_1 p \cdot t))^{1/3}\right) &= \Omega\left(\frac{n^2}{t} \cdot \left(\ln\left(n^{2\epsilon_1/3+\epsilon\epsilon_1} \cdot \ln n\right)\right)^{1/3}\right) \\ &= \Omega\left(\frac{n^2}{n^{4/3} \cdot (\ln n)^{1/3}} \cdot (\ln n)^{1/3}\right) \\ &= \Omega\left(n^{2/3}\right) \end{aligned}$$

as $\epsilon, \epsilon_1 > 0$ are fixed.

The time for doing this is for any given δ with $0 < \delta < 1$

$$\begin{aligned} O\left(p_1 p \cdot n^2 \cdot (p_1 p \cdot t)^3 + \frac{(p_1 p \cdot n^2)^3}{(p_1 p \cdot t)^{3-\delta}}\right) &= O\left((p_1 p)^4 \cdot n^2 \cdot t^3 + \frac{n^6}{t^{3-\delta}}\right) \\ &= O\left((p_1 p)^4 \cdot n^6 \cdot \ln n + n^{2+4\delta/3}\right) \\ &= O\left(n^{2/3+8\epsilon_1/3+4\epsilon\epsilon_1} \cdot \ln n + n^{2+4\delta/3}\right) \\ &= o\left(n^6 \cdot \ln n\right). \end{aligned}$$

Hence, we have an overall running time of $O(n^6 \cdot \log n)$. □

The next result gives a k-dimensional version of Theorem 5.5.

THEOREM 5.8. *Let $k \geq 3$ be a fixed integer. For every fixed $\delta > 0$, one can find in time $O(n^{6k-22/3+\delta})$ a subset X of the k-dimensional $n \times \cdots \times n$ grid such that the distances between distinct points of X are mutually distinct and for some constant $c = c(k, \delta) > 0$ it is*

$$|X| \geq c \cdot n^{2/3} \cdot (\ln n)^{1/3}. \tag{5.17}$$

The existence of such a set X which satisfies (5.17) was shown in [37]. We remark that here only the upper bound $|X| \leq c \cdot \sqrt{k} \cdot n$ is known, as shown by Erdös and Guy [20].

Proof. Let $V = \{1, 2, \ldots, n\}^k$ be the set of vertices of the k-dimensional grid. As in the proof of Theorem 5.5, we form a nonuniform hypergraph $\mathcal{G} = (V, \mathcal{E}_3 \cup \mathcal{E}_4)$ with $\mathcal{E}_3 \subseteq [V]^3$ and $\mathcal{E}_4 \subseteq [V]^4$ as follows. For vertices $x = (x_1, x_2, \ldots, x_k) \in V$ and $y = (y_1, y_2, \ldots, y_k) \in V$ let $d(x, y) = \sqrt{\sum_{i=1}^k (x_i - y_i)^2}$ denote the Euclidean distance between x and y. For pairwise distinct vertices $x, y, z \in V$ let $\{x, y, z\} \in \mathcal{E}_3$ iff $d(x, y) = d(x, z)$. Also, for pairwise distinct vertices $x^1, x^2, x^3, x^4 \in V$ let $\{x^1, x^2, x^3, x^4\} \in \mathcal{E}_4$ iff $d(x^1, x^2) = d(x^3, x^4)$. Again our strategy will be to find a large independent set in $\mathcal{G}$. In a preprocessing we compute and store the sets S_d^k of all solutions of the diophantine equation $x_1^2 + x_2^2 + \cdots + x_k^2 = d$ for $d = 1, 2, \ldots, k(n-1)^2$. By inserting integers $x_1, x_2, \ldots, x_{k-1}$ with $|x_i| \leq \sqrt{d}$ and solving for x_k, this can be done in time

$$O\left(\sum_{d=1}^{k(n-1)^2} (2 \cdot \sqrt{d})^{k-1}\right) = O\left(\int_1^{k \cdot n^2} x^{\frac{k-1}{2}} dx\right) = O\left(n^{k+1}\right).$$

For positive integers d, let $r_k(d)$ denote the number of solutions of $x_1^2 + x_2^2 + \cdots + x_k^2 = d$ within the integers. Rewriting this as $x_1^2 + x_2^2 = d - \sum_{i=3}^k x_i^2$, we infer with (5.4) that

$$r_k(n) \leq (2 \cdot \sqrt{n})^{k-2} \cdot \max_{1 \leq d \leq \sqrt{n}} r_2(d) = O\left(n^{\frac{k}{2} - 1 + \frac{c'}{\ln \ln n}}\right).$$

Hence, for fixed integers $k \geq 3$, we have

$$\max\{r_k(d) \mid d = 1, 2, \ldots, k(n-1)^2\} \leq n^{k-2+\frac{c}{\ln \ln n}} \tag{5.18}$$

for some constant $c > 0$, and therefore,

$$|S_d^k| = r_k(d) \leq n^{k-2+\frac{c}{\ln \ln n}}.$$

For fixed $k \geq 3$, we have by results from [37] that

$$\sum_{d=1}^{n} (r_k(d))^2 = \Theta\left(n^{k-1}\right). \tag{5.19}$$

First we construct the sets $\mathcal{E}_3$ and $\mathcal{E}_4$. For $d = 1, 2, \ldots, k(n-1)^2$ and for each vertex $x_0 \in V$, there are at most $|S_d^k|$ vertices $x^1, x^2, \ldots, x^l \in V$ with $d(x_0, x^j)^2 = d$, $j = 1, 2, \ldots, l$, and all these vertices can easily be determined by a coordinate transformation in time $O(r_k(d))$. Then, $\{x_0, x^i, x^j\}$, $1 \leq i < j \leq l$ yields a 3-element edge $E \in \mathcal{E}_3$. Using (5.19), the set $\mathcal{E}_3$ of 3-element edges can be constructed in time

$$O\left(n^k \cdot \sum_{d=1}^{k(n-1)^2} \binom{r_k(d)}{2}\right) = O\left(n^k \cdot (k \cdot n^2)^{k-1}\right) = O\left(n^{3k-2}\right)$$

and also

$$|\mathcal{E}_3| \leq c_3 \cdot n^{3k-2}.$$

In a similar fashion, we construct the set $\mathcal{E}_4$ of 4-element edges. We pick two distinct vertices $x, y \in V$ and consider the sets $A_d = \{z \in V \mid d(x,z)^2 = d\}$ and $B_d = \{w \in V \mid d(y,w)^2 = d\}$ for $d = 1, 2, \ldots, k(n-1)^2$. Then, $\{x, y, z, w\}$ with $z \in A$ and $w \in B$ yields a 4-element edge $E \in \mathcal{E}_4$ if x, y, z, w are pairwise distinct. Using (5.19), this can be done in time

$$O\left(n^k \cdot n^k \cdot \sum_{d=1}^{k(n-1)^2} (r_k(d))^2\right) = O\left(n^{4k-2}\right)$$

and also

$$|\mathcal{E}_4| \leq c_4 \cdot n^{4k-2}. \tag{5.20}$$

As in the proof of Theorem 5.5, and using (5.18), the set $C_{2,3}$ of $(2,3)$-cycles in the 4-uniform subhypergraph $\mathcal{G}|\mathcal{E}_4 = (V, \mathcal{E}_4)$ can be constructed in time

$$O(|\mathcal{E}_4| \cdot \max\{r_k(d) \mid 1 \leq d \leq k(n-1)^2\}) = O\left(n^{5k-4+\frac{c}{\ln \ln n}}\right),$$

and, for some constant $c_{2,3} > 0$, we have

$$s_{2,3}(\mathcal{G}|\mathcal{E}_4) \leq c_{2,3} \cdot n^{5k-4+\frac{c}{\ln \ln n}}.$$

This shows that for each induced subhypergraph $\mathcal{G}' = (V', \mathcal{E}_4')$ of $\mathcal{G}|\mathcal{E}_4 = (V, \mathcal{E}_4)$, the set of $(2,3)$-cycles can be constructed in time

$$O\left(|\mathcal{E}_4'| \cdot n^{k-2+\frac{c}{\ln \ln n}}\right)$$

and

$$s_{2,3}(\mathcal{G}') = O\left(|\mathcal{E}_4'| \cdot n^{k-2+\frac{c}{\ln \ln n}}\right). \tag{5.21}$$

Again, as in the proof of Theorem 5.5, using (5.18) and (5.20) the construction of the set $C_{2,2}$ of $(2,2)$-cycles in $\mathcal{G}|\mathcal{E}_4$ can be done in time

$$O\left(|\mathcal{E}_4| \cdot |V| \cdot n^{k-2+\frac{c}{\log \log n}}\right) = O\left(n^{6k-4+\frac{c}{\ln \ln n}}\right),$$

and, for some constant $c_{2,2} > 0$, we have

$$s_{2,2}(\mathcal{G}|\mathcal{E}_4) \leq c_{2,2} \cdot n^{6k-4+\frac{c}{\ln\ln n}}.$$

Moreover, for every subhypergraph $\mathcal{G}' = (V', \mathcal{E}_4^{'})$ of $\mathcal{G}|\mathcal{E}_4 = (V, \mathcal{E}_4)$, we have

$$s_{2,2}(\mathcal{G}') = O\left(|\mathcal{E}_4'| \cdot |V'| \cdot n^{k-2+\frac{c}{\ln\ln n}}\right). \tag{5.22}$$

By Lemma 5.6 with $p = n^{-2/3+\epsilon}$, where $0 < \epsilon < 2/3$, we obtain in time $O(|V| + |\mathcal{E}_3| + |\mathcal{E}_4|) = O(n^{4k-2})$ a subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_3^0 \cup \mathcal{E}_4^0)$ of $\mathcal{G}$ with $\mathcal{E}_3^0 \subseteq \mathcal{E}_3$ and $\mathcal{E}_4^0 \subseteq \mathcal{E}_4$ such that

$$|V_0| = p/3 \cdot |V|, \qquad |\mathcal{E}_3^0| \leq 3 \cdot p^3 \cdot |\mathcal{E}_3|, \qquad |\mathcal{E}_4^0| \leq 3 \cdot p^4 \cdot |\mathcal{E}_4|.$$

The value of p is chosen such that the assumptions of Corollary 2.7 are fulfilled for some constants $c, \gamma > 0$:

$$\begin{aligned} s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) &\leq c \cdot p \cdot n^k \cdot (p \cdot t)^{5-\gamma}, \\ s_{2,3}(\mathcal{G}_0|\mathcal{E}_4^0) &\leq c \cdot p \cdot n^k \cdot (p \cdot t)^{4-\gamma}, \end{aligned}$$

where

$$\begin{aligned} s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) &= O(p^4 \cdot |\mathcal{E}_4| \cdot p \cdot n^k \cdot n^{k-2+\frac{c}{\log\log n}}) \\ &= O(n^{6k-22/3+5\epsilon+\frac{c}{\log\log n}}) \end{aligned}$$

and

$$\begin{aligned} s_{2,3}(\mathcal{G}_0|\mathcal{E}_4^0) &= O(p^4 \cdot |\mathcal{E}_4| \cdot n^{k-2+\frac{c}{\log\log n}}) \\ &= O(n^{5k-20/3+4\epsilon+\frac{c}{\log\log n}}). \end{aligned}$$

Let t^3 be the average degree of $\mathcal{G}|\mathcal{E}_4$. By (5.20) we have

$$t^3 \leq \frac{4 \cdot c_4 \cdot n^{4k-2}}{n^k} \leq 4 \cdot c_4 \cdot n^{3k-2}. \tag{5.23}$$

By Lemma 5.7 with $p_1 = (p \cdot t)^{-1+\epsilon_1}$, where $\epsilon_1 = \epsilon/(5k + 5\epsilon)$, we obtain a subhypergraph $\mathcal{G}_1 = (V_1, \mathcal{E}_3^1 \cup \mathcal{E}_4^1)$ of $\mathcal{G}_0$ with $\mathcal{E}_i^1 \subseteq \mathcal{E}_i^0$ for $i = 3, 4$ such that $|V_1| = p_1/5 \cdot |V_0|$, $|\mathcal{E}_3^1| \leq 5 \cdot p_1^3 \cdot |\mathcal{E}_3^0|$, $|\mathcal{E}_4^1| \leq 5 \cdot p_1^4 \cdot |\mathcal{E}_4^0|$, $s_{2,3}(\mathcal{G}_1|\mathcal{E}_4^1) \leq 5 \cdot p_1^5 \cdot s_{2,3}(\mathcal{G}_1|\mathcal{E}_4^1)$, and $s_{2,2}(\mathcal{G}_1|\mathcal{E}_4^1) \leq 5 \cdot p_1^6 \cdot s_{2,2}(\mathcal{G}_1|\mathcal{E}_4^1)$. This can be done in time

$$\begin{aligned} &O(|V_0| + |\mathcal{E}_3^0| + |\mathcal{E}_4^0| + s_{2,3}(\mathcal{G}_0|\mathcal{E}_4^0) + s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0)) \\ &= O\left(p \cdot n^k + p^3 \cdot |\mathcal{E}_3| + p^4 \cdot |\mathcal{E}_4|\right. \\ &\quad \left. + p^4 \cdot |\mathcal{E}_4| \cdot n^{k-2+\frac{c}{\ln\ln n}} + p^4 \cdot |\mathcal{E}_4| \cdot p \cdot n^k \cdot n^{k-2+\frac{c}{\ln\ln n}}\right) \\ &= O\left(n^{k-2/3+\epsilon} + n^{3k-4+3\epsilon} + n^{6k-22/3+5\epsilon+\frac{c}{\ln\ln n}}\right) \\ &= O\left(n^{6k-22/3+5\epsilon+\frac{c}{\ln\ln n}}\right). \end{aligned} \tag{5.24}$$

Since $p_1 p = c \cdot n^{-k+2/3+\epsilon_1(k-4/3+\epsilon)}$, we have by choice of ϵ_1 that

$$|\mathcal{E}_3^1| \leq 15 \cdot (p_1 p)^3 \cdot |\mathcal{E}_3| = O(n^{3\epsilon_1(k-4/3+\epsilon)}) = o(|V_1|).$$

Also, by choice of ϵ_1 we have

$$s_{2,2}(\mathcal{G}_1|\mathcal{E}_4^1) = O(p_1^6 \cdot s_{2,2}(\mathcal{G}_0|\mathcal{E}_4^0) = O(n^{2/3-\epsilon+6\epsilon_1(k-4/3+\epsilon)+\frac{c}{\ln\ln n}}) = o(|V_1|),$$
$$s_{2,3}(\mathcal{G}_1|\mathcal{E}_4^1) = O(p_1^5 \cdot s_{2,3}(\mathcal{G}_0|\mathcal{E}_4^0) = O(n^{-\epsilon+5\epsilon_1(k-4/3+\epsilon)+\frac{c}{\ln\ln n}}) = o(|V_1|).$$

In the resulting subhypergraph $\mathcal{G}_1 = (V_1, \mathcal{E}_3^1 \cup \mathcal{E}_4^1)$, we delete in time $O(|V_1|)$ one vertex from each 3-element edge and from all $(2,2)$- and $(2,3)$-cycles in $\mathcal{G}_1|\mathcal{E}_4^1$. In losing $o(|V_1|)$ vertices, we obtain a subhypergraph $\mathcal{G}_2 = (V_2, \mathcal{E}_4^2)$, which no longer contains any 2-cycles and, possibly after deleting more vertices, satisfies

$$|V_2| = |V_1|/2 = 1/30 \cdot p_1 p \cdot n^k$$

and

$$|\mathcal{E}_4^2| \leq 15 \cdot c_4 \cdot (p_1 p)^4 \cdot n^{4k-2}.$$

We apply Corollary 2.8 to $\mathcal{G}_2$, and we obtain an independent set in $\mathcal{G}_2$; hence in $\mathcal{G}$ of size at least

$$\Omega\left(\frac{p_1 p \cdot n^k}{p_1 p \cdot t} \cdot (\ln(p_1 p \cdot t))^{1/3}\right) = \Omega\left(\frac{n^k}{n^{k-2/3}} \cdot (\ln n)^{1/3}\right) = \Omega\left(n^{2/3} \cdot (\ln n)^{1/3}\right).$$

The time for doing this is for $0 < \delta_1 < 1$ and $\delta_2 = (k - 2/3) \cdot \delta_1$ given by

$$O\left(p_1 p \cdot n^k \cdot (p_1 p \cdot t)^3 + \frac{(p_1 p \cdot n^k)^3}{(p_1 p \cdot t)^{3-\delta_1}}\right) = O\left((p_1 p)^4 \cdot n^k \cdot t^3 + n^{2+\delta_2}\right)$$
$$= o\left(n^{6k-22/3+5\epsilon+\frac{c}{\ln\ln n}}\right).$$

By (5.24), for $\epsilon < \delta/5$ the overall running time is

$$O\left(n^{6k-22/3+5\epsilon+\frac{c}{\ln\ln n}}\right) = O\left(n^{6k-22/3+\delta}\right). \qquad \square$$

The next problem that we consider concerns colorings of the edges of a complete graph. The colorings are such that each color class is a *matching*; that is, edges of the same color have no vertex in common.

THEOREM 5.9. *Let the edges of the complete graph K_n be colored such that every color class is a matching. Then, one can determine in time $O(n^3)$ a complete subgraph K_l of K_n such that the edges of K_l are totally multicolored (injectively colored) and*

$$l \geq c \cdot (n \cdot \ln n)^{1/3}. \tag{5.25}$$

The existence of such a totally multicolored subgraph satisfying (5.25) was shown in [8]. Moreover, Babai [11] showed the existence of colorings of the complete graph K_n, where every color class is a matching, such that every totally multicolored complete subgraph K_l satisfies $l \leq C \cdot (n \cdot \ln n)^{1/3}$, where $C \approx 7.3$. The value of the constant C was improved in [36] to $C = 2.21$. Hence, up to a constant factor we cannot do better in Theorem 5.9 with respect to the size of l.

Proof. Let $\Delta: E(K_n) \longrightarrow N$ be a coloring of the edges of the complete graph K_n with n-element vertex set V, where every color class is a matching.

We form a 4-uniform hypergraph $\mathcal{G} = (V, \mathcal{E})$ on the same vertex set as K_n by collecting pairs of edges of the same color. For distinct edges $e_1 = \{v_1, w_1\} \in E(K_n)$

and $e_2 = \{v_2, w_2\} \in E(K_n)$, we have $E = e_1 \cup e_2 \in \mathcal{E} \in [V]^4$ iff $\Delta(e_1) = \Delta(e_2)$, i.e., $\Delta(e_1) = \Delta(e_2)$ implies $e_1 \cap e_2 = \emptyset$, as every color class is a matching. With $|\Delta^{-1}(i)| \leq n/2$ for $i \in N$ we infer that

$$|\mathcal{E}| = \sum_{i \in N} \binom{|\Delta^{-1}(i)|}{2} \leq \frac{\binom{n}{2}}{\frac{n}{2}} \cdot \binom{\frac{n}{2}}{2} < \frac{1}{8} \cdot n^3. \tag{5.26}$$

By first sorting the edges in $E(K_n)$ according to their color, the hypergraph $\mathcal{G}$ can be constructed in time $O(n^3)$. The average degree $t(\mathcal{G})^3$ of $\mathcal{G}$ satisfies

$$t(\mathcal{G})^3 = \frac{4 \cdot |\mathcal{E}|}{|V|} \leq \frac{n^2}{2} = t^3.$$

First, we construct the $(2,3)$-cycles in $\mathcal{G}$. For each edge $E \in \mathcal{E}$ and each 3-element subset $S \subset E$, the set S can be extended in at most a constant number of ways to an edge $E' \in \mathcal{E}$, i.e., $S \subset E'$, as every color class is a matching. Thus the set of $(2,3)$-cycles can be constructed in time $O(|\mathcal{E}|) = O(n^3)$, and for some constant $c_{2,3} > 0$ we have

$$s_{2,3}(\mathcal{G}) \leq c_{2,3} \cdot n^3. \tag{5.27}$$

We do not construct the set of $(2,2)$-cycles right now. To do this, observe that for each edge $E \in \mathcal{E}$ and each 2-element subset $e \in [E]^2$ there are less than n edges $e' \in E(K_n)$ with $\Delta(e) = \Delta(e')$. Also for distinct vertices $v, w \in E$ and $z \in V \setminus E$, there is at most one edge $e' \in E(K_n)$ with $w \in e'$ and $\Delta(\{v, z\}) = \Delta(e')$. Thus the set of $(2,2)$-cycles can be constructed in time $O(|\mathcal{E}| \cdot n) = O(n^4)$. Similarly, for each subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$ of $\mathcal{G}$, its set of $(2,2)$-cycles can be constructed in time $O(|\mathcal{E}'| \cdot |V'|)$, and also

$$s_{2,2}(\mathcal{G}') \leq c'_{2,2} \cdot |\mathcal{E}'| \cdot |V'|. \tag{5.28}$$

By Lemma 5.6, for $p = n^{-1/3+\epsilon}$ with $0 < \epsilon < 1/3$, using (5.26) and (5.27) we find in time $O(|V| + |\mathcal{E}| + s_{2,3}(\mathcal{G})) = O(n^3)$ a subhypergraph $\mathcal{G}_0 = (V_0, \mathcal{E}_0)$ of $\mathcal{G} = (V, \mathcal{E})$ with

$$|V_0| = p/3 \cdot |V|\ , \quad |\mathcal{E}_0| \leq 3 \cdot p^4 \cdot |\mathcal{E}|\ , \quad \text{and} \quad s_{2,3}(\mathcal{G}_0) \leq 3 \cdot p^5 \cdot s_{2,3}(\mathcal{G}). \tag{5.29}$$

Then, $t(\mathcal{G}_0)^3 \leq 9 \cdot (p \cdot t)^3$. We claim that for $\gamma = \epsilon/(1+\epsilon)$ the following holds:

$$s_{2,2}(\mathcal{G}_0) \ll p \cdot n \cdot (p \cdot t)^{5-\gamma}.$$

To see this, observe, using (5.28), that

$$\begin{aligned} & p^4 \cdot |\mathcal{E}| \cdot p \cdot n \ll p \cdot n \cdot (p \cdot t)^{5-\gamma} \\ \iff & p^{-1+\gamma} \cdot n^{-1/3+2\gamma/3} \ll 1 \\ \iff & n^{-\epsilon+\gamma(\epsilon+1/3)} \ll 1, \end{aligned}$$

and the last inequality holds. Moreover, by (5.27) and (5.29), we have

$$s_{2,3}(\mathcal{G}_0) \leq 3 \cdot p^5 \cdot s_{2,3}(\mathcal{G}) \leq c'_{2,3} \cdot p^5 \cdot n^3 \ll p \cdot n \cdot (p \cdot t)^{4-\gamma}.$$

The assumptions of Corollary 2.7 are fulfilled, and we obtain an independent set of size at least

$$\Omega\left(\frac{p \cdot n}{p \cdot t} \cdot (\ln(p \cdot t))^{1/3}\right) = \Omega\left((n \cdot \ln n)^{1/3}\right).$$

The time for doing this is by (5.28) and (5.29) for $0 < \delta < 1$ and $\epsilon < 2/15$ given by

$$\begin{aligned} & O\left(p \cdot n + p^4 \cdot n^3 + s_{2,2}(\mathcal{G}_0) + s_{2,3}(\mathcal{G}_0) + \frac{(p \cdot n)^3}{(p \cdot t)^{3-\delta}}\right) \\ &= O\left(p \cdot n + p^4 \cdot n^3 + p^5 \cdot |\mathcal{E}| \cdot n + p^5 \cdot n^3 + \frac{n^3}{t^{3-\delta}}\right) \\ &= O\left(n^{7/3+5\epsilon} + \frac{n^3}{n^{2-2\delta/3}}\right) \\ &= O\left(n^{7/3+5\epsilon} + n^{1+2\delta/3}\right) \\ &= o(n^3). \end{aligned}$$

The overall running time is $O(n^3)$. □

Closely related to the problem just considered is that of finding large Sidon sets in Abelian groups [11].

DEFINITION 5.10. *Let $(G, +)$ be an Abelian group. A subset $S \subset G$ is called a* Sidon set *if all pairwise sums $s_1 + s_2$ with $s_1, s_2 \in S$ and $s_1 \neq s_2$ are distinct.*

In the following we will assume that each addition $g + h$ of elements $g, h \in G$ can be done in constant time.

COROLLARY 5.11. *Let $(G, +)$ be an Abelian group. Let $W \subseteq G$ be an n-element subset of G. Then one can compute in time $O(n^3)$ a Sidon set $S \subseteq W$ with*

$$|S| \geq c \cdot (n \cdot \ln n)^{1/3}. \tag{5.30}$$

Note that if $W = G$, where $|W| = n$, each Sidon set S satisfies $\binom{|S|}{2} \leq n$, i.e., $|S| \leq 1 + \sqrt{2 \cdot n}$. The existence of a Sidon set with size as in (5.30) follows from results in [8].

We remark that within the set of integers, finding a Sidon set $S \subseteq \{1, 2, \ldots, n\}$ with $|S| \geq c \cdot \sqrt{n}$ can easily be done in time $O(n)$ using Singer sets; cf. [47].

Proof. Given the set $W \subseteq G$ with $|W| = n$, we form a complete graph K_n with vertex set W. Then we color the edges $\{v, w\}$ of K_n by color $v + w$. This can be done in time $O(n^2)$. As $(G, +)$ is Abelian, every color class is a matching. Then, we apply Theorem 5.9 to our colored K_n, and we obtain in time $O(n^3)$ a totally multicolored complete subgraph K_l with $l \geq c \cdot (n \cdot \ln n)^{1/3}$. As K_l is totally multicolored, the vertices of K_l yield in W a Sidon set. □

The next problem—Sidon sets within the set $\{1^2, 2^2, \ldots, n^2\}$ of squares of integers—was considered for the first time by Alon and Erdös [7], where for any $\epsilon > 0$ the lower bound $|S| \geq c \cdot n^{2/3-\epsilon}$ was proved by a Turán strategy.

THEOREM 5.12. *For every fixed $\delta > 0$, one can find in time $O(n^{2+\delta})$ a Sidon set $S \subset \{1^2, 2^2, \ldots, n^2\}$ with*

$$|S| \geq c_\delta \cdot n^{2/3}. \tag{5.31}$$

The existence of an independent set of size at least as in (5.31) was proved in [37]. By a result of Landau [34], one has the upper bound $|S| \leq c \cdot n/(\ln n)^{1/4}$ for every Sidon set $S \subseteq \{1^2, 2^2, \ldots, n^2\}$.

Proof. We consider a 4-uniform hypergraph $\mathcal{G} = (V, \mathcal{E})$ with vertex set $V = \{1^2, 2^2, \ldots, n^2\}$ and edge set $\mathcal{E} \subseteq [V]^4$. Let $\{i_1^2, i_2^2, i_3^2, i_4^2\} \in \mathcal{E}$ iff $i_1^2 + i_2^2 = i_3^2 + i_4^2$. To construct $\mathcal{G}$, we consider all pairs (i^2, j^2), $1 \leq i < j \leq n$, and sort them according to the value of their sums $i^2 + j^2$ in time $O(n^2 \cdot \ln n)$. Then, we collect pairwise the pairs with the same value of the sum. Recall, that $r_2(d)$ denotes the number of representations of d as a sum of two squares, i.e., $x^2 + y^2 = d$ for integers x, y. By (5.4), we have

$$r_2(n) \leq n^{\frac{c'}{\ln \ln n}}.$$

Using (5.5), we infer that

$$|\mathcal{E}| \leq \sum_{d=1}^{2n^2} \binom{r_2(d)}{2} \leq c_4 \cdot n^2 \cdot \ln n, \tag{5.32}$$

and $\mathcal{E}$ can be constructed in time $O(n^2 \cdot \ln n)$.

Now we consider the 2-cycles in the hypergraph $\mathcal{G}$. To count the number $s_{2,2}(\mathcal{G}')$ of $(2,2)$-cycles in any subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$, consider a fixed edge $\{i_1^2, i_2^2, i_3^2, i_4^2\} \in \mathcal{E}'$. The number of edges $\{i_1^2, i_2^2, x^2, y^2\} \in \mathcal{E}'$ with $i_1^2 + i_2^2 = x^2 + y^2$ is at most $r_2(i_1^2 + i_2^2) \leq n^{\frac{c_1}{\ln \ln n}}$. Moreover, the number of edges $\{i_1^2, i_2^2, x^2, y^2\} \in \mathcal{E}'$ with $i_1^2 + x^2 = i_2^2 + y^2$ is given by a constant times the number of divisors of $i_1^2 - i_2^2$ and hence is at most $n^{\frac{c_2}{\ln \ln n}}$; cf. [27], [37]. Thus, we have for constants $c_3, c_{2,2} > 0$ that

$$s_{2,2}(\mathcal{G}') \leq c_{2,2} \cdot |\mathcal{E}'| \cdot n^{\frac{c_3}{\ln \ln n}} ,$$

and the set of $(2,2)$-cycles in $\mathcal{G}'$ can be constructed in time $O(|\mathcal{E}'| \cdot n^{\frac{c_3}{\ln \ln n}})$.

As every 3-element subset $S \subset V$ can be extended only in a constant number of ways to an edge $E \in \mathcal{E}$, i.e., $S \subset E$, every subhypergraph $\mathcal{G}' = (V', \mathcal{E}')$ of $\mathcal{G}$ satisfies for some constant $c_{2,3} > 0$ that

$$s_{2,3}(\mathcal{G}') \leq c_{2,3} \cdot |\mathcal{E}'|,$$

and the set of $(2,3)$-cycles in $\mathcal{G}'$ can be constructed in time $O\left(|\mathcal{E}'|\right)$.

Now, for the average degree $t(\mathcal{G})^3$ of $\mathcal{G}$ we have by (5.32) that

$$t(\mathcal{G})^3 \leq t^3 = \frac{4 \cdot c_4 \cdot n^2 \cdot \ln n}{n} = c_5 \cdot n \cdot \ln n.$$

Then, for $0 < \gamma < 1/3$, some constant $c > 0$, and n large we have

$$\begin{aligned} s_{2,2}(\mathcal{G}) &\leq c_{2,2} \cdot n^2 \cdot \ln n \cdot n^{\frac{c_3}{\ln \ln n}} \ll n^{8/3 - \gamma/3} \leq c \cdot n \cdot t^{5-\gamma}, \\ s_{2,3}(\mathcal{G}) &\leq c_{2,3} \cdot n^2 \cdot \ln n \ll n^{7/3 - \gamma/3} \leq c \cdot n \cdot t^{4-\gamma}; \end{aligned}$$

hence, the assumptions of Corollary 2.7 are fulfilled, and we obtain for any fixed $\delta' > 0$ with $\delta' < 3\delta$ in time

$$O\left(n^2 \cdot \ln n \cdot n^{\frac{c_3}{\ln \ln n}} + \frac{n^3}{t^{3-\delta'}}\right) = O\left(n^{2+\delta'/3} \cdot (\ln n)^{\delta'/3}\right) = O\left(n^{2+\delta}\right)$$

an independent set of size at least

$$\Omega\left(\frac{n}{(n \cdot (\ln n))^{1/3}} \cdot (\ln n)^{1/3}\right) = \Omega\left(n^{2/3}\right). \qquad \square$$

Sidon sets in arbitrary groups were considered by Babai and Sós [12]. They distinguish two types, as shown in the following.

Definition 5.13. *Let $(G, \cdot)$ be a group, and let S be a subset of G. The set S is a Sidon set of the* first kind *if for all $x, y, z, w \in S$, where at least three are distinct, it is*

$$x \cdot y \neq z \cdot w.$$

The set S is a Sidon set of the second kind *if*

$$x \cdot y^{-1} \neq z \cdot w^{-1}.$$

Theorem 5.14. *Let $(G, \cdot)$ be an arbitrary group. Then for every subset $W \subseteq G$ with $|W| = n$, one can determine in time $O(n^3)$ a subset $S \subseteq W$, which is a Sidon set of both kinds, first and second, and satisfies*

$$|S| \geq c \cdot (n \cdot \ln n)^{1/3}. \tag{5.33}$$

The existence of a Sidon set with $|S| \geq c \cdot n^{1/3}$ was first shown by Babai and Sós [12]. In [36] the existence of a Sidon set S satisfying (5.33) was shown.

Proof. We sketch the arguments for Sidon sets of the second kind. The arguments for Sidon sets of the first kind are similar (also for Sidon sets of both kinds). By results from [12], a subset $S \subseteq G$ is a Sidon set of the second kind if for pairwise distinct elements $x, y, z \in S$ it is

$$x \cdot y^{-1} \neq y \cdot z^{-1} \tag{i}$$

and for all pairwise distinct elements $x, y, z, w \in S$ it is

$$x \cdot y^{-1} \neq z \cdot w^{-1}. \tag{ii}$$

Given a subset $W \subseteq G$, we form a hypergraph $\mathcal{G} = (W, \mathcal{E}_3 \cup \mathcal{E}_4)$ by collecting triples and quadruples of W, which violate conditions (i) or (ii). For pairwise distinct elements $x, y, z \in W$ let $\{x, y, z\} \in \mathcal{E}_3$ iff $x \cdot y^{-1} = y \cdot z^{-1}$. Moreover, for pairwise distinct elements $x, y, z, w \in W$, let $\{x, y, z, w\} \in \mathcal{E}_4$ iff $x \cdot y^{-1} = z \cdot w^{-1}$. Constructing the sets $\mathcal{E}_3, \mathcal{E}_4$ can be done in time $O(n^3)$. Again, we want to find a large independent set in $\mathcal{G}$.

Let $\mathcal{G}|\mathcal{E}_4 = (W, \mathcal{E}_4)$. It is easy to see (cf. [36]) that

$$|\mathcal{E}_3| \leq c_3 \cdot n^2, \quad |\mathcal{E}_4| \leq c_4 \cdot n^3, \quad s_{2,3}(\mathcal{G}|\mathcal{E}_4) \leq c_{2,3} \cdot n^3, \quad \text{and} \quad s_{2,2}(\mathcal{G}|\mathcal{E}_4) \leq c_{2,2} \cdot n^4$$

and the sets of $(2,3)$-cycles can be constructed in time $O(s_{2,3}(\mathcal{G}|\mathcal{E}_4))$. Moreover, for every subhypergraph $\mathcal{G}' = (W', \mathcal{E}_4')$ of $\mathcal{G}|\mathcal{E}_4$, we have

$$s_{2,2}(\mathcal{G}') \leq c'_{2,2} \cdot |\mathcal{E}'| \cdot |W'|.$$

The situation is similar to that in the proof of Theorem 5.9. We first choose with $p = n^{-1/3+\epsilon}$, where $0 < \epsilon < 1/3$, in time $O(n^3)$ a small subhypergraph $\mathcal{G}_0$ of $\mathcal{G}$, where we control the number of vertices, edges, and $(2,3)$-cycles in $\mathcal{G}|\mathcal{E}_4$. In the resulting hypergraph $\mathcal{G}_0$, the number of $(2,2)$-cycles is only $O(p^4 \cdot |\mathcal{E}_4| \cdot p \cdot n) = o(n^3)$ for $\epsilon < 2/15$. Then, on this small subhypergraph we again choose in time $o(n^3)$ with $p = n^{-1/3+\epsilon_1}$ a small subhypergraph $\mathcal{G}_1$, where we control the number of vertices, edges, and, among the 4-element edges, the $(2,3)$-cycles and now also the $(2,2)$-cycles. In the resulting hypergraph $\mathcal{G}_1$, we delete in time $o(n)$ one vertex from each 3-element edge, each $(2,2)$-cycle, and each $(2,3)$-cycle, and we obtain a 4-uniform hypergraph without any

2-cycles, to which we apply Corollary 2.8 in time $o(n^3)$, and we get a desired Sidon set. $\square$

DEFINITION 5.15. *Let $(G, \cdot)$ be a group. Let $C \subset G$ be a subset of G with $1 \notin C$, where C is invariant under taking inverses, i.e., with $C = C^{-1} = \{c^{-1} \mid c \in C\}$. The* Cayley graph *$\Gamma(G, C)$ has vertex set G and edge set $\{\{g, h\} \mid g \cdot h^{-1} \in C$ and $g, h \in G\}$.*

COROLLARY 5.16. *Let $(G, \cdot)$ be a group. There exists a constant $c > 0$ such that the following holds. Let H be a graph on n vertices. Then, for every subset $W \subseteq G$ with $|W| \geq c \cdot n^3/\ln n$, one can construct in time $O(n^9/(\ln n)^3)$ some Cayley subgraph of G and find in it an induced subgraph, which is isomorphic to H.*

The existence of such a subset W with $|W| \geq c \cdot n^3$ was shown first by Babai and Sós [12]. This was improved to $|W| \geq c \cdot n^3/\log n$ in [36].

Proof. The arguments follow [12]. Let $H = (V, E)$ be a graph on n vertices, and let $W \subseteq G$ be a subset of G with $|W| \geq c \cdot n^3/\ln n$, where $c > 0$ is a large enough constant. By Theorem 5.14 we obtain in time

$$O\left(\left(\frac{n^3}{\ln n}\right)^3\right) = O\left(\frac{n^9}{(\ln n)^3}\right)$$

a subset $S \subseteq W$ with $|S| = n$, where S is a Sidon set of the second kind. Now, we identify S with the vertex set V of H. Set $C = \{s \cdot t^{-1} \mid \{s, t\} \in E\}$. Then $1 \notin C$ and $C = C^{-1}$. Moreover, the Cayley graph $\Gamma(S, C)$ is an induced copy of the graph H. $\square$

In connection with the study of random Turán numbers the existence of graphs with many edges and without cycles of small lengths was shown by Kohayakawa, Kreuter, and Steger [31]. We mention without detailed proof that along the lines discussed in this paper the following can be shown.

PROPOSITION 5.17. *Let $k \geq 2$ be a fixed integer. Then, one can compute in time $O(n^{4k-3})$ a graph G on n vertices, which does not contain any cycle $C_3, C_4, \ldots, C_{2k}$, and the number of edges in G is at least*

$$\Omega\left(n^{1+\frac{1}{2k-1}} \cdot (\ln n)^{\frac{1}{2k-1}}\right).$$

Although there are better constructions known, for example, Ramanujan graphs arising from the work of Lubotzky, Phillips, and Sarnak [38], i.e., graphs with at least $\Omega(n^{1+\frac{2}{3k+3}})$ edges which do not contain any cycle $C_3, C_4, \ldots, C_{2k}$, the method used in [31] is interesting to gain a logarithmic factor. Namely, seemingly against the intuition they delete edges in a complete graph K_n with probability $p = n^{-1+\frac{1}{2k-1}+\epsilon}$, where $\epsilon > 0$, which is above the usual choice $p_0 = \Theta(n^{-1+\frac{1}{2k-1}})$. Then they form a hypergraph with vertices being the edges of the resulting random graph G_n and (hyper-)edges consisting of the cycles $C_3, C_4, \ldots, C_{2k}$ in G_n. The resulting hypergraph contains an independent set of size $\Omega(n^{1+\frac{1}{2k-1}} \cdot (\ln n)^{\frac{1}{2k-1}})$, i.e., the corresponding edges in the graph form no cycles $C_3, C_4, \ldots, C_{2k}$.

For algorithmic reasons, we do not use the probabilistic argument from [31]. We consider the complete graph K_n on n vertices. We form a hypergraph $\mathcal{G}$ with vertices being the edges of K_n. The (hyper-)edges in $\mathcal{G}$ are determined by the edges of cycles of length at most $2k$ in K_n. Thus, the hypergraph $\mathcal{G}$ has $\binom{n}{2}$ vertices and $\Theta(n^i)$ edges of cardinality i, where $i = 3, 4, \ldots, 2k$. Then we determine all 2-cycles among the $2k$-element edges of $\mathcal{G}$. The number of $(2, j)$-cycles, $j = 2, 3, \ldots, 2k-1$, among the $2k$-element edges of $\mathcal{G}$ is $\Theta(n^{4k-j-1})$. Hence, the assumptions of Corollary 2.7 are fulfilled. We get rid of those (hyper-)edges with cardinality less than $2k$ by choosing vertices of $\mathcal{G}$ with probability $p = n^{-\frac{2k-2}{2k-1}+\epsilon}$ for some $\epsilon > 0$, i.e., using the derandomized argument, and we obtain a $2k$-uniform subhypergraph to which we apply Corollary 2.7. The running time of this algorithm is given by the number of $(2, 2)$-cycles, i.e., is at most $O(n^{4k-3})$.

We remark that Proposition 5.17 was recently generalized in [15]; see also [35]. Moreover, an algorithm for Heilbronn's problem (cf. [32]) was given in [14].

REFERENCES

[1] M. Ajtai, P. Erdös, J. Komlós, and E. Szemerédi, *On Turán's theorem for sparse graphs,* Combinatorica, 1 (1981), pp. 313–317.

[2] M. Ajtai, J. Komlós, J. Pintz, J. Spencer, and E. Szemerédi, *Extremal uncrowded hypergraphs*, J. Combin. Theory Ser. A, 32 (1982), pp. 321–335.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, *A note on Ramsey numbers*, J. Combin. Theory Ser. A, 29 (1980), pp. 354–360.

[4] M. Ajtai, J. Komlós, and E. Szemerédi, *A dense infinite Sidon sequence*, European J. Combin., 2 (1981), pp. 2–11.

[5] N. Alon, *A parallel algorithmic version of the local lemma*, Random Structures Algorithms, 2 (1991), pp. 367–378.

[6] N. Alon, L. Babai, and A. Itai, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, J. Algorithms, 7 (1986), pp. 567–583.

[7] N. Alon and P. Erdös, *An application of graph theory to additive number theory*, European J. Combin., 6 (1980), pp. 201–203.

[8] N. Alon, H. Lefmann, and V. Rödl, *On an anti-Ramsey type result*, Colloq. Math. Soc. János Bolyai, 60 (1991), pp. 9–22.

[9] N. Alon and J. Spencer, *The Probabilistic Method*, John Wiley, New York, 1992.

[10] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Los Alamitos, CA, 1992, pp. 14–23.

[11] L. Babai, *An anti-Ramsey theorem,* Graphs Combin., 1 (1985), pp. 23–28.

[12] L. Babai and V. T. Sós, *Sidon sets in groups and induced subgraphs of Cayley graphs,* European J. Combin., 6 (1985), pp. 101–114.

[13] M. Bellare, O. Goldreich, and M. Sudan, *Free bits, PCPs, and non-approximability: Towards tight results*, in Proc. 36th IEEE Symposium on Foundations of Computing, Los Alamitos, CA, 1995, pp. 422–431.

[14] C. Bertram-Kretzberg, T. Hofmeister, and H. Lefmann, *An algorithm for Heilbronn's problem*, Lecture Notes in Comput. Sci. 1276, T. Jiang and D. T. Lee, eds., Springer, Berlin, 1997, pp. 23–31.

[15] C. Bertram-Kretzberg, T. Hofmeister, and H. Lefmann, *Sparse* $0,1$-*matrices and forbidden hypergraphs*, in Proc. 9th ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1998, pp. 181–187.

[16] R. Boppana and M. Halldórsson, *Approximating maximum independent sets by excluding subgraphs*, BIT, 32 (1992), pp. 180–196.

[17] S. Brandt, *Computing the independence number of dense triangle-free graphs*, Lecture Notes in Comput. Sci., R. Möhring, ed., Springer, Berlin, 1997, pp. 100–108.

[18] R. A. Duke, H. Lefmann, and V. Rödl, *On uncrowded hypergraphs,* Random Structures Algorithms, 6 (1995), pp. 209–212.

[19] P. Erdös, R. L. Graham, I. Rusza, and H. Taylor, *Bounds for arrays of dots with distinct slopes or lengths*, Combinatorica, 12 (1992), pp. 39–44.

[20] P. Erdös and R. Guy, *Distinct distances between lattice points*, Elem. Math., 25 (1970), pp. 121–123.

[21] A. Fundia, *Derandomizing Chebychev's inequality to find independent sets in uncrowded hypergraphs*, Random Structures Algorithms, 8 (1996), pp. 131–147.

[22] M. Goldberg and T. Spencer, *An efficient parallel algorithm that finds independent sets of guaranteed size*, SIAM J. Discrete Math., 6 (1993), pp. 443–459.

[23] S. W. Golomb, *Construction of signals with favourable correlation properties*, in Surveys in Combinatorics, London Mathematical Soc. Lecture Note Ser. 166, 1991, pp. 1–39.

[24] S. W. Golomb and H. Taylor, *Two-dimensional synchronization patterns for minimum ambiguity*, IEEE Trans. Inform. Theory, IT-28 (1982), pp. 600–604.
[25] M. Halldórsson and J. Radhakrishnan, *Greed is good: Approximating independent sets in sparse and bounded-degree graphs*, Algorithmica, 18 (1997), pp. 145–163.
[26] M. Halldórsson and J. Radhakrishnan, *Improved approximations of independent sets in bounded-degree graphs via subgraph removal*, Nordic J. Comput., 1 (1994), pp. 475–492.
[27] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, Oxford University Press, London, 1979.
[28] J. Håstad, *Clique is hard to approximate within $n^{1-\epsilon}$*, in Proc. 37th IEEE Symposium on Foundations of Computer Science, Los Alamitos, CA, 1996, pp. 627–636.
[29] T. Hofmeister and H. Lefmann, *Derandomization for sparse approximations and independent sets*, Lecture Notes in Comput. Sci. 969, J. Wiedermann and P. Hájek, eds., Springer, Berlin, 1995, pp. 201–210.
[30] P. Kelsen, *On the parallel complexity of computing a maximal independent set in a hypergraph*, in Proc. 24th ACM Symposium on the Theory of Computing, New York, 1992, pp. 339–350.
[31] Y. Kohayakawa, B. Kreuter, and A. Steger, *An extremal problem for random graphs and the number of graphs with large even-girth*, Combinatorica, 18 (1998), pp. 101–120.
[32] J. Komlós, J. Pintz, and E. Szemerédi, *A lower bound for Heilbronn's problem*, J. London Math. Soc., 25 (1982), pp. 13–24.
[33] G. Kortsarz and D. Peleg, *On choosing a dense subgraph (extended abstract)*, in Proc. 34th IEEE Symposium on Foundations of Computer Science, Los Alamitos, CA, 1993, pp. 692–701.
[34] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*, Teubner, Leipzig, 1909.
[35] H. Lefmann, P. Pudlák, and P. Savický, *On sparse parity check matrices*, Des., Codes Cryptogr., 12 (1997), pp. 107–130.
[36] H. Lefmann, V. Rödl, and B. Wysocka, *Multicolored subsets in colored hypergraphs*, J. Combin. Theory Ser. A, 74 (1996), pp. 209–248.
[37] H. Lefmann and T. Thiele, *Point sets with distinct distances*, Combinatorica, 15 (1995), pp. 379–408.
[38] A. Lubotzky, R. Phillips, and P. Sarnak, *Ramanujan graphs*, Combinatorica, 8 (1988), pp. 261–277.
[39] T. Łuczak and E. Szymańska, *A parallel randomized algorithm for finding a maximal independent set in a linear hypergraph*, J. Algorithms, 25 (1997), pp. 311–320.
[40] K. T. Phelps and V. Rödl, *Steiner triple systems with minimum independence number*, Ars Combin., 21 (1986), pp. 167–172.
[41] P. Pudlák and V. Rödl, *Modified ranks of tensors and the size of circuits*, in Proc. 25th ACM Symposium on the Theory of Computing, New York, 1993, pp. 523–531.
[42] P. Raghavan, *Probabilistic Construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.
[43] S. Ramanujan, *Collected Papers*, Chelsea Publishing Company, New York, 1962, pp. 133–135.
[44] V. Rödl and E. Siňajová, *Note on independent sets in Steiner systems*, Random Structures Algorithms, 5 (1994), pp. 183–190.
[45] J. Shearer, *A note on the independence number of triangle-free graphs*, Discrete Math., 46 (1983), pp. 83–87.
[46] J. Shearer, *A note on the independence number of triangle-free graphs*, II, J. Combin. Theory, 53 (1991) pp. 300–307.
[47] J. Singer, *A theorem in finite projective geometry and some applications to number theory*, Trans. Amer. Math. Soc., 43 (1938), pp. 377–385.
[48] J. Spencer, *Turán's theorem for k-graphs*, Discrete Math., 2 (1972), pp. 183–186.
[49] J. Spencer, *Uncrowded hypergraphs*, in Mathematics of Ramsey Theory, J. Nešetřil and V. Rödl, eds., Springer, Berlin, 1990, pp. 253–262.
[50] T. Thiele, *Point Sets with Distinct Slopes or Lengths*, preprint, Freie Universitát, Berlin, 1993.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 231–257

ACHILLES, TURTLE, AND UNDECIDABLE BOUNDEDNESS PROBLEMS FOR SMALL DATALOG PROGRAMS[*]

JERZY MARCINKOWSKI[†]

Abstract. DATALOG is the language of logic programs without function symbols. It is considered to be the paradigmatic database query language. If it is possible to eliminate recursion from a DATALOG program then it is *bounded.* Since bounded programs can be executed in parallel constant time, the possibility of automatized boundedness detecting is believed to be an important issue and has been studied in many papers. Boundedness was proved to be undecidable for different kinds of semantical assumptions and syntactical restrictions. Many different proof techniques were used.

In this paper we propose a uniform proof method based on the discovery of, as we call it, the Achilles–Turtle machine, and make strong improvements on most of the known undecidability results. In particular we solve the famous open problem of Kanellakis showing that uniform boundedness is undecidable for single rule programs (called also *sirups*).

This paper is the full version of [J. Marcinkowski, *Proc.* 13*th STACS*, Lecture Notes in Computer Science 1046, pp. 427–438], and [J. Marcinkowski, 11*th IEEE Symposium on Logic in Computer Science*, pp. 13–24].

Key words. DATALOG, query optimization, decidability

AMS subject classification. 68P15

PII. S0097539797322140

1. Introduction.

1.1. Introduction. Consider the query relation R, which answers for a given directed graph and for two given nodes A and B of the graph, whether it is possible to reach B from A in an odd number of steps. It turns out that R is not expressible in first order logic. That is due to lack of recursion in the first order logic. This observation led to the study of DATALOG (DATAbase LOGic) programs which combine existential positive first order logic with recursion. For example, the relation R can be defined by an "odd-distance" DATALOG program:

(i) $R(X,Y){:-}\, E(X,Y)$,

(ii) $R(X,Y){:-}\, R(X,Z), R(Z,W), E(W,Y)$,

where E is the edge relation of the graph. E is the so-called extensional predicate (EDB): we treat it as an input and are not able to prove new facts about it. R is the output, or intensional predicate (IDB). The program proves facts about it. The first rule is an initialization rule: it has only the extensional predicate in the body. But the second rule contains the intensional predicate within its premises; it can be used recursively and deep proofs can be constructed. It is clear that if in some graph there is a path from an element A to B of an odd length n, then to prove $R(A,B)$ for such elements, a proof of a depth about $\log n$ may be needed. Thus in huge databases arbitrarily deep proofs are necessary to evaluate the program.

On the other hand, consider a program which computes the "has 3-tail" query:

(iii) $1TAIL(Z){:-}\, E(Z,X)$,

(iv) $2TAIL(Z){:-}\, 1TAIL(Y), E(Z,Y)$,

[*]Received by the editors May 30, 1997; accepted for publication (in revised form) July 10, 1998; published electronically September 14, 1999. This work was supported by the KBN grant 8T11C02913.

http://www.siam.org/journals/sicomp/29-1/32214.html

[†]Institute of Computer Science, Wrocław University, ul Przesmyckiego 20, 51151 Wrocław, Poland (jma@tcs.uni.wroc.pl).

(v) $3TAIL(Z)$:– $2TAIL(Y), E(Z,Y)$.

If $3TAIL(A)$ is provable for some A, then there exists a proof of the fact which is not deeper than 3, regardless of the number of elements in the database. $1TAIL$, $2TAIL$, and $3TAIL$ are IDB and the second and third rules are recursive. But in fact, the recursion can be eliminated completely from the last program. It is possible to write an equivalent one where only proofs of depth 1 will be necessary:

$3TAIL(X)$:– $E(T,W), E(W,Z), E(Z,X)$.

The recursion can be eliminated from a given program, and the program is equivalent to a first order query if and only if there is an a priori upper bound on the depth of the proofs needed to evaluate queries. Every fact that can be derived by the program can be derived in constant time (in parallel, with polynomially many processors) independent of the size of the database (this equivalence was proved in [3]; the "if" direction is nontrivial). Such programs are called *bounded*.

1.2. Previous works and our contribution. The problem of distinction, meaning whether a given DATALOG program is bounded or not, is important for DATALOG queries optimization, but is, in general, undecidable. Sufficient conditions for boundedness were given in [17], [10], [18], and [19]. The decidability-undecidability border, for cases of different syntactical restrictions and semantical assumptions, has been studied in [20], [5], [2], [6], [8], [9], [24], [23].

The syntactical restrictions considered were the number of rules or the number of recursive rules in the program, maximal arity of the IDB symbols, and the linearity of rules.

The semantical assumptions concern the status of the IDB relations before the execution of the program. If they are empty, then we deal with weak (program) boundedness. While arbitrary relations must be considered as possible IDB inputs, strong (uniform) boundedness is studied.

Undecidability of uniform boundedness implies undecidability of program boundedness for fixed syntactical restrictions (with possibly some additional initialization rules; see section 1.7 for a discussion). The survey of previously known results ((i)-(v) below) illustrates the difference in the level of difficulty of undecidability proofs for uniform and program boundedness.

Decidability has been proved for monadic programs program boundedness (so also for the uniform) [6], [5] and for typed single rule programs [20]. It is also known that the program (and uniform) boundedness is decidable for programs with a single linear recursive rule if the IDB is binary [24]. Moreover, program boundedness is decidable for binary programs if each IDB is defined by only one recursive rule [23].

Undecidability has been proved for

(i) program boundedness of linear binary programs [9],

(ii) program boundedness of programs with one recursive rule and two initializations [2],

(iii) program boundedness of programs consisting of two linear recursive rules and one initialization [9],

(iv) uniform boundedness of ternary programs [9],

(v) uniform boundedness of arity 5 linear programs [8].

Decidability of the uniform boundedness for programs consisting of only one rule was stated as an open problem in [11], where NP-hardness of the problem was proved, and then in [2] and [12]. No undecidability results for uniform boundedness of programs with a small number of rules have been proved since then.

In this paper we give strong improvements of the results (ii)–(v) showing that

(vi) uniform boundedness is undecidable for ternary linear programs (section 3.1) (which improves the results (iv) and (v)).

(vii) uniform boundedness is undecidable for single recursive rule ternary programs (section 3.3) (which improves (iv)).

The additional improvement is that our program is syntactically simpler: the recursive rule is quasi-linear, which means that, generally speaking, it has the form $I(tuple1)$:– $J(tuple2), I(tuple3)$, where I and J are intensional predicates. Since it is the only recursive rule, the proof from the program is a tree with only one (possibly) long branch.

Notice that in (vi) and (vii) we still allow a number of initializations so the results hold also for program boundedness.

(viii) Uniform and program boundedness are undecidable for programs consisting of one linear recursive rule and one initialization (section 4.3).

Since program boundedness is clearly decidable for programs consisting of one rule, the result (viii) closes the number/linearity of rules classification for program boundedness. It is a strong improvement of (ii) and (iii).

Finally, in section 4.5 we solve the problem of Kanellakis showing that

(ix) uniform boundedness of single rule programs is undecidable.

1.3. The method. While different techniques were used in the proofs of the results (i)–(v) (reduction to the halting and mortality problems of a Turing machine, reduction from the halting problem of a two counters machine, syntactical reduction of an arbitrary DATALOG program to a single recursive rule program), we develop for all our results a universal method, based on an encoding of Conway functions. We have learned about Conway functions from the paper of Philippe Devienne, Patrick Lebègue and Jean-Christophe Routier [7], who used them to prove undecidability of the so-called cycle unification. We feel that our paper would not have been written without their previous work. Our encoding is nevertheless quite different from the one in [7]: the first difference is that a language with functions was used there.

We construct, as we call it, the Achilles-Turtle machine, a variant of the Turing machine. Next, we use a version of the Conway theorem to prove that what we constructed is really a universal machine. Then we encode the Achilles-Turtle machine with DATALOG programs. Due to the particular simplicity of Achilles-Turtle machine (one is really tempted to claim that it is the simplest known universal machine), it is possible to encode it with syntactically very small DATALOG programs. We believe that this is not the last time that the Achilles-Turtle machine is used in undecidability proofs.

We combine the Conway functions method with the technique of using a binary EDB relation as an order: if there is a chain long enough in the relation, then we can think that it represents a tape of the machine. If there is no such chain then proofs are not too long. This method goes back to [9] and [8].

1.4. Open problems. While the classification is finished for program boundedness, the following syntactical restrictions still give interesting open problems concerning decidability of uniform boundedness:

(i) binary programs,

(ii) linear binary programs,

(iii) programs consisting of a single linear rule.

We do not know any example of syntactical restrictions for which uniform boundedness would be decidable and program boundedness not. It seems that the most likely candidate for the example is the class of linear binary programs. Program boundedness is known to be undecidable for the class.

1.5. Preliminaries. A *DATALOG program* is a finite set of Horn clauses (called *rules*) in the language of first order logic without equality and without functions. The predicates, used in the program, but only in the bodies of the rules, are called *extensional predicates* or EDB. A predicate which occurs in a head of some rule is called *intensional* or IDB. A rule is called *recursive* if an IDB occurs in its body. A rule which is not recursive is an *initialization rule*. A recursive rule is *linear* if it has only one occurrence of an IDB in the body. A program is linear if each of its recursive clauses is linear. *Arity* of a DATALOG program is the highest arity of the IDB predicates used.

So, for example, in the two programs of section 1.1 the predicate E is extensional, and all the other predicates are intensional. The rules (i) and (iii) are initializations. Rules (ii), (iv), and (v) are recursive. Rules (iv) and (v) are linear and so the "has 3-tail" program is linear. It is also monadic, while the "odd-distance" program is binary.

A database is a finite set of ground atomic formulas. A *derivation* (or a *proof*) of a ground atomic formula $\mathcal{A}$, from the program $\mathcal{P}$ and the database $\mathcal{D}$, is a finite tree such that (i) each of its nodes is labeled with a ground atomic formula; (ii) each leaf is labeled with an atom from $\mathcal{D}$; (iii) for each nonleaf node there exists a rule R in the program $\mathcal{P}$ and a substitution σ, such that σ of the head of R is the label of the node and the substitutions of the body of R are the labels of its children; (iv) $\mathcal{A}$ is the label of the root of the tree. The *depth* of the proof is the depth of the derivation tree.

Instead of writing *proof from the program* $\mathcal{P}$ *in the database* $\mathcal{D}$ we use the expression $\mathcal{P} - \mathcal{D}$*-proof*, or simply *proof* if the context is clear.

Notice that if $\mathcal{P}$ is a linear program, then a $\mathcal{P}$-proof is a sequence of ground atomic formulas. In such a case we use the word *length* for the depth of the proof.

In general, a program $\mathcal{P}$ is bounded if for every database $\mathcal{D}$, if an atom can be proved from $\mathcal{P}$ and $\mathcal{D}$, then it has a proof not deeper than a fixed constant c.

Different conventions concerning the input and output of a DATALOG program correspond to different definitions of boundedness: *predicate*, *program*, and *uniform* boundedness are studied. A program is *predicate bounded*, with respect to a fixed predicate PRE, if there is a constant c such that for every database $\mathcal{D}$, such that there are no facts about IDBs in $\mathcal{D}$; for every ground atom $\mathcal{A} = PRE(tuple)$, if the atom has a proof from $\mathcal{P}$ and $\mathcal{D}$, then it has a proof not deeper than c. This definition reflects the situation when the EDBs are the input and only one predicate is the output of the program. A program $\mathcal{P}$ is *program bounded* if it is predicate bounded for all IDBs.

A program is *uniformly bounded* if there is a constant c such that for every database $\mathcal{D}$ (here we do not suppose that the IDB predicates do not occur in $\mathcal{D}$), and for every ground atom $\mathcal{A}$, if the atom has a proof from $\mathcal{P}$ and $\mathcal{D}$, then it has a proof not deeper then c. Here all the predicates are viewed as the input and as the output of a program.

1.6. Example: Program boundedness vs. uniform boundedness. To make the difference between program boundedness and uniform boundedness clear to the reader, we give an example of a program which is bounded but not uniformly bounded.

The signature of the program consists of one extensional predicate E and one intensional predicate I. Both predicates are binary.

The rules are
(i) $I(X,X) : -E(X,Y), E(Y,Z)$,
(ii) $I(W,R) : -I(Z,Z), E(X,Y), E(Y,Z)$,
(iii) $I(X,Y) : -I(Y,X)$,
(iv) $I(X,Y) : -I(Z,Y), E(X,Z)$,
(v) $I(X,Y) : -I(Z,Y), E(Z,X)$.

It is convenient to think that E is a graph, and I is a kind of a pebble game: by the initialization rule (i) we can start the game by placing both the pebbles in any node which has a tail of length at least 2. By rule (iii) we do not need to distinguish between the pebbles. By rules (iv) and (v) we can always move one of the pebbles to a neighboring node, and finally, if the two pebbles meet in the node that is the end of a tail of length at least two, then by rule (ii) we can move the pebbles to any two nodes.

We prove that the program is program bounded but not uniformly bounded.

LEMMA 1.1. *For a database $\mathcal{D}$ such that the input predicate I is empty, either there are no proofs in $\mathcal{D}$ or for each pair D, E of elements of $\mathcal{D}$, the fact $I(D,E)$ can be proved in no more than 7 derivation steps.*

Proof. We consider two cases.

Case 1. There are elements A, B, C in $\mathcal{D}$ such that $E(A,B)$ and $E(B,C)$ hold. Then, in the first step, we use rule (i) to prove $I(A,A)$. Then, using rule (v) twice we get $I(C,A)$. Then we use rule (iii) to get $I(A,C)$ and rule (v) twice to get $I(C,C)$. Finally, rule (ii) can be used to derive $I(D,E)$.

Case 2. There are no such elements A, B, C in the database $\mathcal{D}$. Then, since I is given as empty, no proofs at all are possible. □

The structure of the proof of the Lemma 1.1 above, as well as the structure of the program itself, is a good illustration of one of the ideas of the proofs in sections 3 and 4. The program contains some initialization rules which allows us to start a kind of game, or computation, if only there exists a substructure of required form in the database. Then, if there is enough facts in the database, we can proceed with the computation and, when it terminates, use an analogue of rule (ii) to "flood" the database. Otherwise, if there are not enough facts, then only short proofs are possible (or no proofs at all, as in the example).

LEMMA 1.2. *For each constant c there exist a database $\mathcal{D}$ with nonempty input predicate I, and elements A, B of $\mathcal{D}$ such that $I(A,B)$ is $\mathcal{P}$-provable, but the shortest proof of the fact requires more than c steps.*

Proof. The database contains the elements $C_1, C_2, \dots C_{2c}$ and the following facts: $E(C_{2k-1}, C_{2k})$ for all $1 \le k \le c$, $E(C_{2k+1}, C_{2k})$ for all $1 \le k \le c-1$, and $I(C_1, C_1)$. We will show that the fact $I(C_{2c}, C_1)$ is provable, and the shortest proof has exactly $2c-1$ steps. First we show that such a proof exists: in the kth step use the already proved fact $I(C_k, C_1)$ to derive $I(C_{k+1}, C_1)$. The rule used in the kth derivation step is (v) if k is odd and (iv) if k is even.

To see that shorter proofs are not possible, notice that only the bodies of the rules (iii)–(v) can be satisfied in $\mathcal{D}$, and so only those rules can be used.

Define the distance between nodes of $\mathcal{D}$ as follows: the distance between A and A is 0, and the distance between A and B is less than or equal to k if and only if there exists a node C such that either $E(B,C)$ or $E(C,B)$ and the distance between A and B is at most $k-1$. The distance between a pair of nodes A, B in $\mathcal{D}$ and a pair of nodes C, D in $\mathcal{D}$ is defined as a sum of distances from A to C and from B to D. Now, notice that if a fact of the form $I(A,B)$ is derived in k steps from the fact $I(C,D)$, and if only rules (iii)–(v) are used in the proof, then the distance between A, B and C, D is not greater than k. Finally, observe that the distance in $\mathcal{D}$ between C_{2c}, C_1 and C_1, C_1 is $2c - 1$. $\square$

1.7. Program boundedness vs. uniform boundedness. Discussion. The notions of uniform and program boundedness formalize, on the technical level, the informal notion of boundedness. Uniform boundedness is what we need when the program under consideration is a subprogram of a bigger one. Then it can happen that the predicates that are supposed to be the output of the program are also a part of the input. Program boundedness, on the other hand, corresponds to the view of an entire DATALOG program as a definition of possibly many output predicates. This is similar to the distinction between program and uniform equivalence of DATALOG programs (see [21]), where again the first notion applies to entire programs, while the second one to subprograms equivalence. It is known that program equivalence is undecidable, while uniform equivalence is decidable [5], [21], [22]. We can observe that also for the case of boundedness, the uniform version, for given syntactical restrictions, is a priori "more decidable." Suppose that program boundedness is decidable for some syntactical restrictions and that the restrictions allow an arbitrary number of initializations. Then uniform boundedness is also decidable for the restrictions. To see that, consider a program $\mathcal{P}$ over a signature with IDB symbols I_i, where $1 \le i \le k$. Let $\mathcal{Q}$ be the program $\mathcal{P}$ with its signature enriched with new EDB symbols E_i, where $1 \le i \le k$ and for each i the arity of E_i is equal to the arity of I_i, and with k new rules: $I_i(X_1, X_2, \ldots, X_{a_i}) : -E_i(X_1, X_2, \ldots, X_{a_i})$.

It is easy to see that $\mathcal{Q}$ is program bounded if and only if $\mathcal{P}$ is uniformly bounded. So we reduced the decision problem of the uniform boundedness of $\mathcal{P}$ to the problem of program boundedness of the program $\mathcal{Q}$.

The survey of results gives evidence that it is more difficult to prove undecidability of uniform boundedness than undecidability of program boundedness; the argument above shows that there are reasons for that. But on the other hand we do not know any example of syntactical restrictions for which uniform boundedness would be decidable and program boundedness not. The most likely candidate for the example is the class of linear binary programs. Program boundedness is undecidable for the class and decidability of uniform boundedness is open.

2. Achilles-Turtle machine.

2.1. The tool: Conway functions.

Definition 2.1. *A Conway function is a function g with natural arguments defined by a system of equations:*

$$g(n) = \begin{cases} a_0 n/q_0 & (n \equiv 0 \pmod{p}), \\ \cdots & \cdots \\ a_i n/q_i & (n \equiv i \pmod{p}), \\ \cdots & \cdots \\ a_{p-1} n/q_{p-1} & (n \equiv p-1 \pmod{p}), \end{cases}$$

where a_i,q_i are natural numbers, $q_i | i$ (which means that i/q_i is a natural number), and $q_i | p$ for each i and $(a_i/q_i) \le p$ for each i.

For a Conway function g and a given natural number N let $C(g, N)$ be a statement asserting that there exists a natural number i such that $g^i(N) = 1$.

See section 2.3 to find a nice example giving an idea of what a Conway function is. Proof of the following theorem can be found in [4], [14], or [7].

THEOREM 2.2 (Conway). *The following problem is undecidable given a Conway function g and a natural number N, does $C(g, N)$ hold?*

Our main tool is the following refined version of Theorem 2.2.

THEOREM 2.3.

1. *There exists a computable sequence $\{g_n\}$ of Conway functions such that*

(i) *$\{n : C(g_n, 2)\}$ is not recursive (is r.e. complete);*

(ii) *for each g_n, if a_i and q_i are coefficients from the definition of the function g_n, then $(a_i/q_i) \leq 2$;*

(iii) *for each g_n, if there are such i, k that $k \equiv 1 \pmod p$ and $g_n^i(2) = k$, then $k = 1$.*

2. *There exists a universal Conway function g, such that*

(i) *the set $\{N : C(g, 2^N)\}$ is not recursive (is r.e. complete);*

(ii) *if a_i and q_i are coefficients from the definition of the function g, then $(a_i/q_i) \leq 2$;*

(iii) *for each N, if there are such i, k that $k \equiv 1 \pmod p$ and $g^i(2^N) = k$ then $k = 1$.*

Proof. 1. It is known that the problem: *given a finite automaton with* 2 *counters, does the computation starting from fixed beginning state, and from empty counters reach some fixed final state* is undecidable, even if we require that the final state can be reached only if the both counters are empty (read the remark in the end of this section to see what precisely we mean by *a finite automaton with* 2 *counters*).

For a given automaton $\mathcal{A}$ of this kind we will construct a Conway function $g_{\mathcal{A}}$ which satisfies conditions (ii) and (iii) of the theorem and such that $C(g_{\mathcal{A}}, 2)$ holds if and only if the computation of $\mathcal{A}$ reaches the final state. First we need to modify $\mathcal{A}$ a little bit: we construct an automaton $\mathcal{B}$ which terminates if and only if $\mathcal{A}$ terminates and which satisfies the following conditions:

(iv) the second counter of $\mathcal{B}$ can be increased only if the first counter is decreased in the same computation step,

(v) the states of $\mathcal{B}$ are numbered. If any of the counters is increased in the computation step when the state s_i is being changed into s_j then $j < i$.

The details of the construction of $\mathcal{B}$ are left as an easy exercise. The hint is that all what must be done is adding a couple of new states. For example if there is an instruction of $\mathcal{A}$ which increases the second counter and keeps the first unchanged, it must be substituted by two instructions: first of them only increases the first counter and changes the state into a new one, the second increases the second counter and decreases the first.

Now, suppose that the states of the automaton $\mathcal{B}$ are $s_1, s_2, \ldots s_k$, where s_b is the beginning state. Let $p_1, p_2, \ldots p_k$ be an increasing sequence of primes such that $p_1 > 4$ and $2p_1 > p_k$ (such a sequence can be found for each k, since the density of primes around n is $c/\log n$). We encode the configuration of $\mathcal{B}$: *state is s_i, the first counter contains the number n and the second counter contains m,* as the natural number $2^n 3^m p_i$. It is easy to notice that, if x and y are codes of two subsequent configurations of $\mathcal{B}$ then y/x depends only of the remainder $x \pmod p$ where $p = 6p_1p_2 \ldots p_k$ and that $y/x \leq 2$. So we can define the required Conway function. To define the first step properly we put $a_2 = p_b$ and $q_2 = 2$, so $g(2) = p_b$ which is the code of the beginning configuration. We put also $a_{p_f} = 1$ and $q_{p_f} = p_f$ to reach 1 in the iteration of the function next to the one when the code of the final configuration is reached.

2. We use the well known fact that there exists a particular finite automaton with two counters for which the problem *does the computation starting from a fixed begin-*

ning state s_b, given first counter, and empty second counter, reaches the configuration of some fixed final state s_f, and empty counters is undecidable. Then the proof is similar as of (i). To start the computation properly we put $a_i = p_b$ and $q_i = 1$ for all such even i that $p_j|i$ does not hold for any $j = 1, 2 \dots k$. So for each N it holds that $g(2^N) = p_b 2^N$. The last is the code of the beginning configuration. □

Remark (automata with counters). Our notion of a finite automaton with two counters is similar to the one in Kozen's book [13], with the difference that we assume that the automaton has no input tape. Since two counter automata (with read-only input tape) are as powerful as Turing machines, the problem whether a given automaton of this kind will terminate for given input is undecidable. But, for each input, separately, we can hide the input in the finite control of the automaton (in fact the input tape **is** a finite object for each input). So also the problem whether a given automaton without input tape will terminate when started from a fixed beginning state and from empty counters is undecidable. Now we show, as it is needed in the proof of the second claim of Theorem 2.3, that there exists a particular finite automaton with two counters for which the problem *does the computation starting from a fixed beginning state s_b, given first counter, and empty second counter, reach the configuration of some fixed final state s_f and empty counters* is undecidable. First observe that there exists an automaton as required but with three counters: it is a universal Turing machine with the contents of the part of the tape left of the head remembered on one counter, right on the head on the second counter, and with an auxiliary third counter needed for operating the first two. Then use the standard techniques to encode the three counters as two. See [13] for details.

Convention 2.4. *Now we consider only Conway functions g_n (for $n = 1, 2 \dots$), whose existence was proved in Theorem* 2.3(1). *In particular we assume that the claims* (ii) *and* (iii) *from Theorem* 2.3(1) *hold.*

2.2. Achilles-Turtle machine. For a given Conway function g and given input N we will construct an Achilles-Turtle machine, which will compute the subsequent iterations of $g(N)$.

It is a variant of a multihead Turing Machine, with read-only tape. Each cell of the tape is colored with one of the colors K_0, K_1,$\dots K_{p-1}$, (where p is as in the definition of the function g). If the cell X is colored with the color K_i (we denote the fact as $K_i(X)$) and the cell $S(X)$ (S is a successor function on the tape) is coloured with K_j then $j - i \equiv 1 \pmod{p}$. The color K_0 will be called *white* and K_1 will be called *red.*

There are three heads. The first is called Achilles, the second is called the Turtle, and the third is called Guide. The transition rules will be designed in such a way, that the heads will never go left. Achilles and Guide will move right in each step of the computation. Achilles will try to catch the Turtle.

The configuration of the machine is described by the positions of the heads. In the beginning of the computation, Achilles is in some arbitrary white cell X on the tape. The Turtle and Guide are both in the cell $S^N(X)$. So the beginning configuration is $CON(X, S^N(X), S^N(X))$, where again S is the successor function on the tape. The idea is that the computation can reach a configuration of a form $CON(Y, S^k(Y), S^k(Y))$ or Achilles can be exactly k cells behind the Turtle, if $g^i(N) = k$ for some i.

In each computation step the heads of the machine move according to one of the following transition rules ($i = 0, 1, \dots, p-1$):
(R_i) $CON(S^p(A), T, S^{p(a_i/q_i)}(G)) \Leftarrow CON(A, T, G), K_i(T),$

(J_i) $CON(S^p(A), S^{d_i}(G), S^{d_i}(G)) \Leftarrow CON(A, S^i(A), G)$.
where $d_i = i(a_i/q_i) + p - i$. Notice that, since $a_i/q_i < 2$ and $i < p$ also $0 < d_i < 2p$.

Rules (R_i) are *run rules* and rules (J_i) are *jump rules.* Configurations of the form $CON(A, T, T)$ are called *special*.

See section 2.3 for a nice example of the Achilles-Turtle machine. The following easy lemma gives an intuition of how the computation of the machine proceeds.

LEMMA 2.5. (i) *If, in some configuration of the machine, the Turtle is in the cell X and the Guide is in Y, then $Y = S^k(X)$ for some $k \geq 0$.*

(ii) *If, in some configuration of the machine, the Turtle is in the cell X and Achilles is in some $S^k(X)$, where $0 < k$, then none of the jump rules will be used later in the computation.*

(iii) *Suppose that in some configuration of the machine, Achilles is in some cell X, Turtle is in some $S^t(X)$, and the Guide is in $S^r(X)$. If one of the jump rules can be used later, then $0 \leq t \leq r$.*

(iv) *A special configuration can only be a result of a transition done according to one of the jump rules.*

(v) *Achilles is always in a white cell.*

(vi) *If in some configuration of the machine the Guide is in the cell X, then in the next configuration he will be in $S^r(X)$ for some $0 < r \leq 2p$*

Proof. (i) The claim is true for the beginning configuration and for every configuration being a result of a use of a jump rule. The run rules move the Guide right and keep the Turtle in his cell.

(ii) If Achilles is right of the Turtle, then the jump rule cannot be used. But the run rules move Achilles only right.

(iii) This rule follows from (i) and (ii).

(iv) By (i) the Guide can never be left of the Turtle. The run rules move him right, so after the execution of a run rule he is right of the Turtle.

(v) He starts in a white cell and moves p cells right in each step.

(vi) Since $0 < p(a_i/q_i) < 2p$ and $0 < d_i < 2p$ hold for every i (see Convention 2.4). ◻

Now we will formulate and prove some lemmas about the equivalence between the behavior of the Conway function and the result of the computation of the Achilles-Turtle machine. Our goal is as follows.

LEMMA 2.6. *The following conditions are equivalent:*

(i) *$C(g, N)$ holds.*

(ii) *The Achilles-Turtle machine can reach a configuration of form $CON(A, T, T)$, where $K_1(T)$.*

(iii) *The Achilles-Turtle machine can reach a configuration of form $CON(A, S(A), S(A))$.*

(iv) *The Achilles-Turtle machine can reach a configuration of form $CON(A, T, G)$, where $K_1(T)$.*

LEMMA 2.7. *Suppose in some special configuration of the machine, Achilles is in some cell A, and Turtle and Guide are in some $T = S^{kp+i}(A)$, where $0 < i < p$ (so $K_i(T)$). Then*

(i) *after k steps the configuration will be $CON(S^{-i}(T), T, S^{kp(a_i/q_i)}(T))$,*

(ii) *there are exactly two configurations that can be reached after $k+1$ steps: $CON(S^{p-i}(T), T, S^{(k+1)p(a_i/q_i)}(T))$ and $CON(S^{p-i}(T), S^{d_i+kp(a_i/q_i)}(T), S^{d_i+kp(a_i/q_i)}(T))$.*

Proof. (i) Each of the k steps will be done according to the rule R_i. So after k steps Achilles will be in the cell $S^{kp}(A) = S^{-i}(T)$, Guide will be in $S^{kp(a_i/q_i)}(T)$, and the Turtle in T.

(ii) By (i), $CON(S^{-i}(T), T, S^{kp(a_i/q_i)}(T))$ is reached after k steps. Then rule R_i may be used once again, which leads to $CON(S^{p-i}(T), T, S^{(k+1)p(a_i/q_i)}(T))$. Also the rule J_i may be used, which leads to $CON(S^{p-i}(T), S^{d_i+kp(a_i/q_i)}(T), S^{d_i+kp(a_i/q_i)}(T))$. ∎

Lemma 2.8. *Suppose in some special configuration of the machine Achilles is in the cell A, and Turtle and Guide are in some $T = S^m(A)$, where $m = kp + i$, $0 \le i < p$. Then the following two conditions are equivalent:*

(i) *it is possible to reach a special configuration $CON(X, S^l(X), S^l(X))$ as the next special configuration,*

(ii) $l = g(m)$.

Proof. By Lemma 2.7(ii) the configuration after $k+1$ steps will be either $CON(S^{p-i}(T), T, S^{(k+1)p(a_i/q_i)}(T))$ or $CON(S^{p-i}(T), S^{d_i+kp(a_i/q_i)}(T), S^{d_i+kp(a_i/q_i)}(T))$.

In the first case Achilles will be already right of the Turtle and, by Lemma 2.5 (ii), (iv), a special configuration will no longer be reached.

To prove the equivalence we show that the configuration reached in the second case is just of the form $CON(X, S^{g(m)}(X), S^{g(m)}(X))$.

In fact,

$$\begin{aligned}
&CON(S^{p-i}(T), S^{d_i+kp(a_i/q_i)}(T), S^{d_i+kp(a_i/q_i)}(T))\\
&= CON(S^{p-i}(T), S^{d_i+kp(a_i/q_i)-p+i}S^{p-i}(T), S^{d_i+kp(a_i/q_i)-p+i}S^{p-i}(T))\\
&= CON(X, S^{d_i+kp(a_i/q_i)-p+i}(X), S^{d_i+kp(a_i/q_i)-p+i}(X))\\
&= CON(X, S^{i(a_i/q_i)+p-i+kp(a_i/q_i)-p+i}(X), S^{i(a_i/q_i)+p-i+kp(a_i/q_i)-i+p}(X))\\
&= CON(X, S^{(i+kp)(a_i/q_i)}(X), S^{(i+kp)(a_i/q_i)}(X))\\
&= CON(X, S^{g(m)}(X), S^{g(m)}(X)). \qquad ∎
\end{aligned}$$

Lemma 2.9. *The following two conditions are equivalent:*

(i) *The Achilles-Turtle machine can reach a configuration of the form $CON(X, S^l(X), S^l(X))$.*

(ii) *There exists a natural number j such that $g^j(N) = l$.*

Proof. The (i)⇒(ii) implication is proved by induction on the number of special configurations reached during the computation.

The (ii)⇒(i) implication is proved by induction on j.

In both cases Lemma 2.8 is used for the induction step. ∎

Proof of Lemma 2.6. (i), (ii), and (iii) are equivalent by Lemma 2.9 and Convention 2.4 (claim (iii) of Theorem 2.3(i)). Clearly, (ii) implies (iv). Also (iv) implies (ii): if a configuration $CON(A, T, G)$ is reached after some number of steps, and $K_1(T)$ holds, then consider the configuration after the last step of the computation which was done according to a jump rule (the last step when the Turtle was moved). This configuration is $CON(A', T, T)$ for some A'. ∎

2.3. Achilles-Turtle machine. An example. In order to give the reader an idea of how the machine works we are going to provide a nice example of a Conway function (or rather Conway-like function) and of the Achilles-Turtle machine built for this function. The function g that we start from will be the well-known Collatz function. Take a natural number: if it is even then divide it by two, if it is odd then multiply it by three and add one. The problem if the iterations of the procedure give finally the result 1, regardless of the natural number that we start from, is open. More formally, in the spirit of Definition 2.1 we can define function g as

$$g(n) = \begin{cases} n/2 & (n \equiv 0 \pmod 2), \\ 3n+1 & (n \equiv 1 \pmod 2), \end{cases}$$

and the open problem is then whether

$$\{N : C(g,N)\} = \mathcal{N}.$$

We do not only multiply the number, but also add 1, so this is not really a Conway function in the sense of Definition 2.1, but we find this example to be interesting nonetheless. We can and will construct our Achilles-Turtle machine for the following function.

The rules of the Example Achilles-Turtle machine
initial configuration: $CON(X, S^N(X), S^N(X)) : -WHITE(X).$

transition rules:

run rules:

$CON(S^2(A), T, S^6(G)) : -CON(A, T, G), RED(T).$

$CON(S^2(A), T, S(G)) : -CON(A, T, G), WHITE(T).$

jump rules:

$CON(S^2(A), S^5(G), S^5(G)) : -CON(A, S(A), G), RED(S(A)).$

$CON(S^2(A), S^2(G), S^2(G)) : -CON(A, A, G), WHITE(A).$

final configuration: $CON(X, S(X), S(X)).$

The coefficients in run rules and in the *white* jump rule are here calculated according to the definitions of (R_i) and (J_i) from the beginning of section 2.2. The left-hand side of the *red* jump rule is not $CON(S^2(A), S^4(G), S^4(G))$, as would follow from the definition: this is the place where we add 1 to form the $3n+1$.

Now suppose, for concreteness of the example, that N is 5. Then the subsequent iterations of g are 5, 16, 8, 4, 2, 1. The beginning configuration of the machine will be then $CON(A, S^5(A), S^5(A))$ for some white cell A and the computation sequence of the machine is as follows:
$CON(S^2(X), S^5(X), S^{11}(X))$ (RR⋆),
$CON(S^4(X), S^5(X), S^{17}(X))$ (RR⋆),
$CON(S^6(X), S^{22}(X), S^{22}(X))$ (RJ⋆),
$CON(S^8(X), S^{22}(X), S^{23}(X))$ (WR⋆),
$CON(S^{10}(X), S^{22}(X), S^{24}(X))$ (WR),
$CON(S^{12}(X), S^{22}(X), S^{25}(X))$ (WR),
$CON(S^{14}(X), S^{22}(X), S^{26}(X))$ (WR),
$CON(S^{16}(X), S^{22}(X), S^{27}(X))$ (WR),
$CON(S^{18}(X), S^{22}(X), S^{28}(X))$ (WR),
$CON(S^{20}(X), S^{22}(X), S^{29}(X))$ (WR),
$CON(S^{24}(X), S^{32}(X), S^{32}(X))$ (WJ),
$CON(S^{26}(X), S^{32}(X), S^{33}(X))$ (WR),
$CON(S^{28}(X), S^{32}(X), S^{34}(X))$ (WR),
$CON(S^{30}(X), S^{32}(X), S^{35}(X))$ (WR),
$CON(S^{32}(X), S^{32}(X), S^{36}(X))$ (WR),
$CON(S^{34}(X), S^{38}(X), S^{38}(X))$ (WJ),
$CON(S^{36}(X), S^{38}(X), S^{39}(X))$ (WR),
$CON(S^{38}(X), S^{38}(X), S^{40}(X))$ (WR),
$CON(S^{40}(X), S^{42}(X), S^{42}(X))$ (WJ),
$CON(S^{42}(X), S^{42}(X), S^{43}(X))$ (WR),

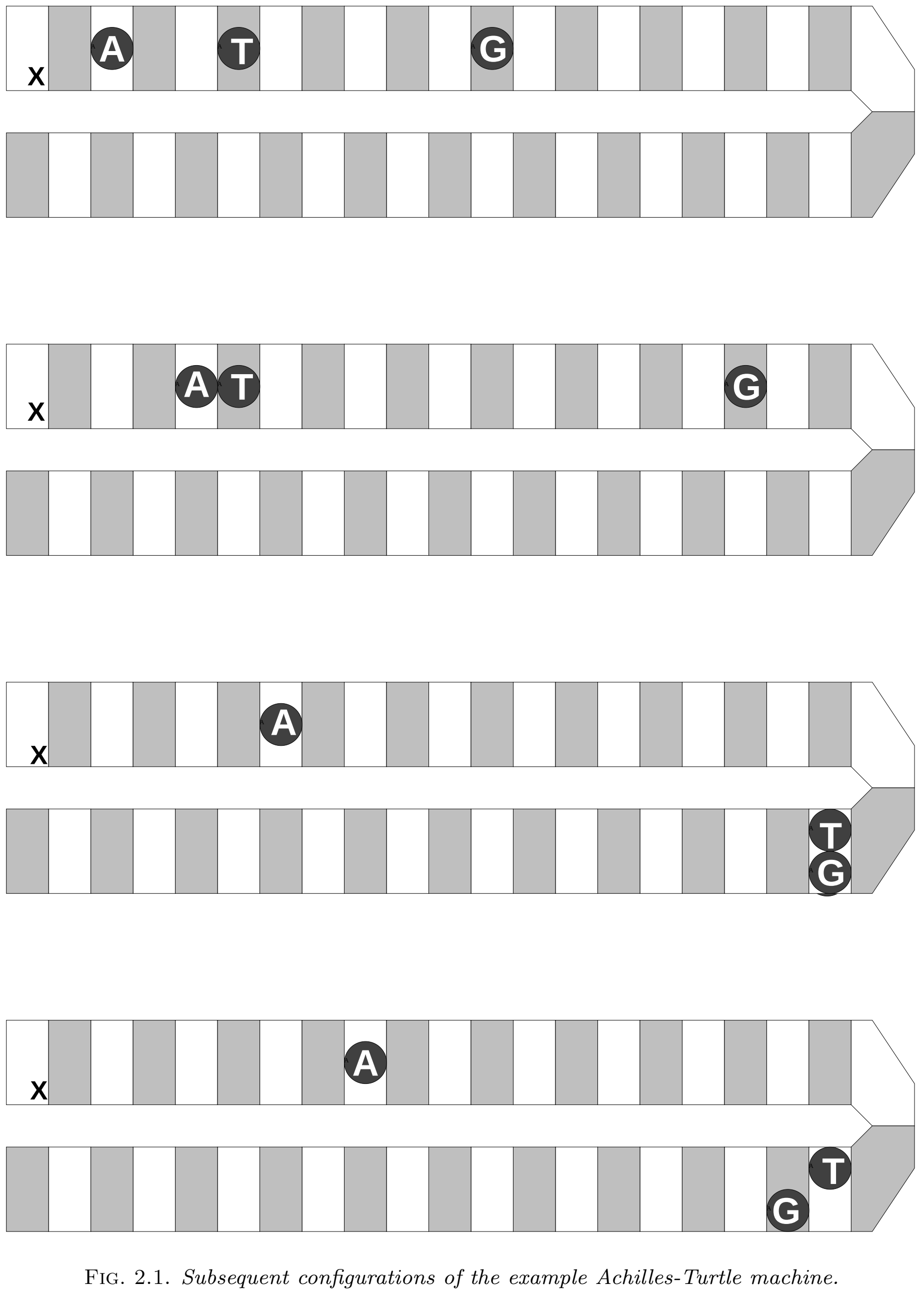


FIG. 2.1. *Subsequent configurations of the example Achilles-Turtle machine.*

$CON(S^{22}(X), S^{22}(X), S^{30}(X))$ (WR),

where RR means that red run rule was used to obtain the configuration, WR is white run rule, RJ red jump rule, and WJ is white jump rule. The configurations marked with $\star$ are depicted below (see Figure 2.1).

3. Ternary programs.

3.1. The ternary linear program $\mathcal{P}$.

Theorem 3.1. *For each Conway function g_n from Theorem* 2.3(i) *there exists, and can be efficiently constructed, an arity* 3 *linear DATALOG program $\mathcal{P}$ with one IDB predicate which is uniformly bounded if and only if $C(g_n, 2)$ holds.*

The signature of the program contains one binary EDB symbol S, which is going to serve as a kind of order for us, p monadic EDB symbols which will play as the colors and a ternary IDB symbol CON. The program $\mathcal{P}$ consists of the following.

Transition rules (for each i, $0 \le i \le p-1$)**:**

$CON(S^p(A), T, S^{p(a_i/q_i)}(G),){:\!-}\ CON(A, T, G), K_i(T), K_0(A), \ldots, K_{p-1}(S^{p-1}(A)).$

$CON(S^p(A), S^{d_i}(G), S^{d_i}(G)){:\!-}\ CON(A, S^i(A), G), K_0(A), \ldots, K_{p-1}(S^{p-1}(A)).$

Flooding rule: $CON(X, Y, Z){:\!-}CON(S, T, R), K_1(T).$

Initialization: $CON(A, S^2(A), S^2(A)){:\!-}K_0(A), K_1(S(A)), K_2(S^2(A)).$

K_m is understood as K_i if $m \equiv i \pmod p$. Since S is no longer a true successor we must explain the meaning of the S^l symbols in the rules.

Notational Convention: (for example) a rule

$CON(S^2(X), S^4(Y), S(X)){:\!-}CON(X, S(X), Y)$

should be understood as

$CON(X2, Y4, X1)$
 ${:\!-}CON(X, X1, Y),$
 $S(Y, Y1), S(Y1, Y2), S(Y2, Y3), S(Y3, Y4), S(X, X1), S(X1, X2).$

Let us explain the meaning of the rules: The transition rules are the same as in the Achilles-Turtle machine, with the exception that they check if the cells (nodes) that Achilles runs over are painted properly. The flooding rule proves everything in one step if Turtle is in a red node. The initialization allows us to start the computation in each (white) node, if there is a properly colored piece of tape near the node.

Lemma 3.2. *If $C(g_n, 2)$ does not hold, then for each c there exists a database $\mathcal{D}$ and a tuple A, T, G of elements of $\mathcal{D}$ such that $CON(A, T, G)$ can be proved in $\mathcal{D}$ with P and the proof of $CON(A, T, G)$ requires more than c steps.*

Proof. $\mathcal{D}$ is just a long enough S-chain (see Definition 3.4 below) with empty IDB relation. First we prove that the flooding rule can not be used in such a database, provided that $C(g_n, 2)$ does not hold. Suppose it can be used. That means that $CON(A, T, G)$ can be proved for some red T. If we follow the proof of $CON(A, T, G)$ in $\mathcal{D}$ we will notice that it gives a legal computation of the Achilles-Turtle machine and that the first fact in the proof is the beginning configuration of the machine. That is a contradiction by Lemma 2.6. Now, take the first element Z of the order. By initialization we have $CON(Z, S^2(Z), S^2(Z))$. Using 2c times the run rule 2 we get the shortest proof of the fact $CON(S^{2pc}(Z), S^2(Z), S^{2+2cp(a_2/q_2)}(Z))$. ∎

Now we are going to prove the following lemma.

Lemma 3.3. *If $C(g_n, 2)$ holds, then there exists c such that, in any database $\mathcal{D}$, for every tuple A, B, C of elements of $\mathcal{D}$ if $CON(A, B, C)$ can be proved in $\mathcal{D}$ with* *the program P, then there exists a proof of $CON(A, B, C)$ shorter than c steps.*

Proof. Suppose $C(g_n, 2)$ holds. That means that if we start the computation of the Achilles-Turtle machine in a configuration $CON(X, S^2(X), S^2(X))$, then it is possible to reach a final configuration $CON(A, S(A), S(A))$.

Notice, that during the computation, none of the heads will move left of X or right of $S(A)$. Let K be the distance between X and $S(A)$ and let K' be the number of steps of the computation necessary to reach the final configuration. Clearly $pK' + 1 = K$. We are going to prove that $c = K' + 2$ is the proper constant.

We will need some definitions.

Definition 3.4. *An S-chain of elements of a database $\mathcal{D}$ is a set $X_0, X_1, \dots, X_k$ such that $S(X_m, X_{m+1})$ and $K_m(X_m)$ for $m = 0, 1, \dots, k$. A decreasing S-chain of elements of the database $\mathcal{D}$ is a set $X_0, X_1, X_2, \dots, X_k$ such that $S(X_{m+1}, X_m)$ and $K_m(X_m)$ for $m = 0, 1, \dots, k$ (where K_m should be understood as K_i if $m \equiv i \pmod p$). In both cases we say that the chain begins in X_0.*

Definition 3.5. *Let k be a natural number. We say that a node W of a database $\mathcal{D}$ is not k-founded if there exists a decreasing S-chain which begins in W and consists of more than k elements. W is k-founded if such a chain does not exist.*

Definition 3.6. *Let k be a natural number. We say that a database $\mathcal{D}$ is not k-founded if there exists an S-chain consisting of more than k elements.*

Obviously, $\mathcal{D}$ is k-founded if such a chain does not exist.

Now we consider 2 cases.

Lemma 3.7. *If $\mathcal{D}$ is not K-founded, then for each tuple A, B, C of elements of $\mathcal{D}$ the fact $CON(A, B, C)$ can be proved and the proof requires no more than $K' + 2$ steps.*

Proof. Take X such that there exists an S-chain of length K beginning in X. Thanks to the initialization rule $CON(X, S^2(X), S^2(X))$ is provable in $\mathcal{D}$ and has a proof of length 1. Now we can pretend that the chain from X to $S^K(X)$ is a tape and start a computation of the Achilles-Turtle machine. Since the transition rules of the machine are rules of program $\mathcal{P}$, each step of the computation can be encoded by one step of proof. So there exists an element T of the chain such that $S(T)$ is red, and $CON(T, S(T), S(T))$ can be proved after K'+1 steps. One more step (using flooding) is needed to prove CON of every tuple after that. □

Lemma 3.8. *Let $\mathcal{D}$ be a K-founded database.*

(i) *Suppose $\{CON(A_i, T_i, B_i)\}_{i=0}^m$ is a $\mathcal{P}$-proof in $\mathcal{D}$. If the flooding rule is not used in the proof then $m \leq K'$.*

(ii) *Let $\{CON(A_i, T_i, B_i)\}_{i=0}^m$ be a $\mathcal{P}$-proof in $\mathcal{D}$, and suppose it is the shortest possible proof of $CON(A_m, B_m, C_m)$. Then the flooding rule is used at most once, for the last step of the proof.*

(iii) *If $CON(A, B, C)$ can be $\mathcal{P}$-proved in $\mathcal{D}$ for some tuple A, B, C, the proof requires no more than $K' + 1$ steps.*

Proof. (i) The set A_i, $(i = 1 \dots m)$ is a subsequence of an S-chain of length pm.

(ii) Suppose that the step from $CON(A_i, T_i, B_i)$ to $CON(A_{i+1}, T_{i+1}, B_{i+1})$ for some $i \neq m - 1$ is done according to the flooding rule. Then

$CON(A_0, T_0, B_0), CON(A_1, T_1, B_1), \dots CON(A_i, T_i, B_i), CON(A_m, T_m, B_m)$

is a shorter proof of $CON(A_m, T_m, B_m)$.

(iii) It follows from (i) and (ii). □

This ends the proof of Lemma 3.3 and of Theorem 3.1. □

Theorem 3.9. *Uniform boundedness of ternary linear DATALOG programs is undecidable.*

Proof. This follows from Theorems 2.2 and 3.1. ◻

3.2. The arity 5 single recursive rule program $\mathcal{R}$.

THEOREM 3.10. *For each Conway function g_n from Theorem 2.3(i) there exists, and can be efficiently constructed, an arity 5 DATALOG program $\mathcal{R}$ consisting of one quasi-linear recursive rule and of some initializations, which is uniformly bounded if and only if $C(g_n, 2)$ holds.*

As in the previous subsection the signature of the program contains one binary EDB symbol S, which is going to serve as a kind of order for us, p monadic EDB symbols which will play as the colours and a ternary IDB symbol CON. There is also additional IDB symbol $STEER$ of arity 5. The program $\mathcal{R}$ consists of the following.

The recursive rule:
$CON(S^p(A), T', G')$:–
$CON(A, T, G)$,
$STEER(A, T, G, T', G')$,
$K_0(A), K_1(S(A)), \dots K_{p-1}(S^{p-1}(A)), K_0(S^p(A))$.

Initialization "transition" rules (for each i, $0 \le i \le p-1$)**:**

$STEER(A, T, G, T, S^{p(a_i/q_i)}(G))$:–$K_i(T)$.

$STEER(A, S^i(A), G, S^{d_i}(G), S^{d_i}(G))$:–$K_i(S^i(A))$.

The initialization "flooding" rule: $STEER(X, T, Y, R, S)$:–$K_1(T)$.
The initialization: $CON(A, S^2(A), S^2(A))$:–$K_0(A), K_1(S(A)), K_2(S^2(A))$.

Let us explain what is going on here: The triples (Achilles, Turtle, Guide) are nodes of a graph defined on $\mathcal{D}^3$ by means of order and coloring. Each proof, either from program $\mathcal{P}$ or from $\mathcal{R}$, is a path in the graph. The graph is not given by the EDB relations but can be defined from them by a DATALOG program without recursion. If we want to define the vertices beginning in a node A, T, G "online," when the computation reaches the node (as in $\mathcal{P}$), then we must use more than only one rule, but the rules are linear: they read nothing more than the information about the EDB situation around. If we define the graph in advance (by initializations), then one recursive rule is enough: we have a "graph accessibility" program in this case. But the rule is only "quasi-linear": it makes use of the additional IDB (but not recursive) predicate $STEER$. If I were the reader I would ask a question here: why is the $STEER$ predicate not of arity 6? Why do not we want to hide the rule for Achilles in the initializations and have a simpler recursive clause? In fact, some additional problems arise here, since we do not have a flooding rule for Achilles. We were forced to design the recursive rule in this way because of the uniformity reasons. It is crucial that Achilles goes down the chains. Thanks to that we can say: no long chains, no long proofs (Lemmas 3.8 and 3.12, case 1). We could write the initializations of the hypothetical 6-ary $STEER$ in such a way that Achilles would move only down the chains while running according to the $STEER$ facts proved by the initializations. But we would have no control of what is given as the $STEER$ at the beginning.

LEMMA 3.11. *If $C(g_n, 2)$ does not hold, then for each c there exists a database $\mathcal{D}$ and a tuple A, B, C of elements of $\mathcal{D}$ such that $CON(A, B, C)$ can be proved in $\mathcal{D}$ with a program $\mathcal{R}$, and the proof of $C(A, B, C)$ requires more than c steps.*

Proof. The proof is the same as that for Lemma 3.2. ◻

LEMMA 3.12. *If $C(g_n, 2)$ holds, then there exists a c such that, in every database $\mathcal{D}$, for every tuple A, B, C of elements of $\mathcal{D}$, if $CON(A, B, C)$ can be proved in $\mathcal{D}$ with $\mathcal{R}$, then there exists a proof of $CON(A, B, C)$ shorter than of c steps.*

DEFINITION 3.13. *If A, B, C is a tuple of elements of the database $\mathcal{D}$, then we say that $CON(A, B, C)$ is a fact about A.*

Proof of Lemma 3.12. Suppose $C(g_n, 2)$ holds. Then there is a computation of the Achilles-Turtle machine starting in some configuration $CON(X, S^2(X), S^2(X))$ and reaching $CON(Y, S(Y), S(Y))$. The computation requires space kp (that is the distance from X to Y is kp) for some natural k. We will consider two cases.

Case 1. A is $(k+1)p$-founded. Then every proof of a fact about A is no longer than $k+1$. That is because of the Achilles' part in the recursive rule. This is analogous to Lemma 3.8(i).

Case 2. C is not $(k+1)p$-founded.

So take V such that $A = S^{(k+1)p}(V)$ and there is a chain of length $(k+1)p$ from V to A. Because of the initialization rule $CON(V, S^2(V), S^2(V))$ is provable in $\mathcal{D}$ and has a proof of length 1. Now we can pretend that the chain from V to $S^{kp+1}(V)$ is a tape and let Achilles and Turtle play their game there. Among other rules possibly given by the predicate $STEER$ they have also the "standard" Achilles-Turtle machine rules. So, after k moves the configuration $CON(S^{kp}(V), S^{kp+1}(V), S^{kp+1}(V))$ will be reached and we will be allowed to use the flooding. Every fact of the form $CON(A, B, C)$ will be proved in one step. So no new facts about A can be proved later. Of course nothing new about the IDB predicate $STEER$ can be proved after the first step.

This ends the proof of Lemma 3.12 and of Theorem 3.10. □

3.3. The ternary single recursive rule program $\mathcal{Q}$.

THEOREM 3.14. *For each Conway function g_n from Theorem* 2.3(i) *there exists, and can be efficiently constructed, an arity* 3 *DATALOG program $\mathcal{Q}$ consisting of one quasi-linear recursive rule and of some initializations, which is uniformly bounded if and only if $C(g_n, 2)$ holds.*

Similarly, as in the previous subsection the signature of the program contains one binary EDB symbol S, which is going to serve as a kind of order for us, p monadic EDB symbols which will play as colors, and a ternary IDB symbol CON. The graph which was defined by a arity 5 relation in the previous section will be defined here as an intersection of four graphs defined by ternary constraints. So, we will have four additional ternary IDB symbols $E_{G,T,T'}$, $E_{A,T,T'}$, $E_{T,G,G'}$, and $E_{T,T',G'}$ in the language of the program. The rules of the program Q are as follows.

The recursive rule:

$CON(S^p(A), T', G')$:–
$CON(A, T, G)$,
$E_{G,T,T'}(G, T, T'), E_{A,T,T'}(A, T, T'), E_{T,G,G'}(T, G, G'), E_{T,T',G'}(T, T', G')$,
$K_0(A), K_1(S(A)), \dots K_{p-1}(S^{p-1}(A)), K_0(S^p(A)), K_1(S^{p+1}(A))$.

The initialization "constraints" rules:

$E_{G,T,T'}(G, T, T)$.

For each i $(0 \le i \le p-1)$ there is a rule : $E_{G,T,T'}(G, T, S^{d_i}(G))$:–$K_i(T)$.

$E_{G,T,T'}(G, T, T')$:–$K_1(T)$.

$$E_{A,T,T'}(A,T,T).$$

For each i $(0 \leq i \leq p-1)$ there is a rule : $E_{A,T,T'}(A,S^i(A),T').$

$$E_{A,T,T'}(A,T,T'):–K_1(T).$$

For each i $(0 \leq i \leq p-1)$ there is a rule : $E_{T',G,G'}(S^{d_i}(G),G,S^{d_i}(G)).$

For each i $(0 \leq i \leq p-1)$ there is a rule : $E_{T',G,G'}(T',G,S^{p(a_i/q_i)}(G)):–K_i(T).$

$$E_{T',G,G'}(T',G,G'):–K_1(T).$$

$$E_{T,T',G'}(T,T,G').$$

$$E_{T,T',G'}(T,T',T').$$

$$E_{T,T',G'}(T,T',G'):–K_1(T).$$

The initialization:

$CON(A,S^2(A),S^2(A)):–\ K_0(A),K_1(S(A))K_2(S^2(A)).$

To prove the correctness of the construction we shall argue that the ternary relations really define the same graph as the relation $STEER$ of the last section. It is easy to notice that if $STEER(A,T,G,T',G')$ can be proved by one of the initializations of $\mathcal{R}$, then $E_{G,T,T'}(G,T,T')$, $E_{A,T,T'}(A,T,T')$, $E_{T',G,G'}(T',G,G')$, and $E_{T,T',G'}(T,T',G)$ can also be proved. For the opposite inclusion, suppose that T is not red. We first consider the relation $E_{G,T,T'}$. Since the Guide "does not see" how far from each other Achilles and Guide are, the constraint allows the Turtle to stay in the same place or to jump according to the proper jump rule. It is the relation $E_{A,T,T'}$ that decides if the Turtle will be allowed to jump. If Achilles is far away, then the Turtle can only wait. If Achilles is about to catch the Turtle, then the Turtle is allowed to jump (see [1]) anywhere. But, because of the relation $E_{G,T,T'}$, this "anywhere" can be only $S^{d_i}(G)$ for the proper i. In this way, already the first two relations force the Turtle to behave as he should.

The relation $E_{T',G,G'}$ forces the Guide to move ahead. It allows the Guide to execute his jump rule but only if the Turtle jumps together with him (this prevents the danger that the Guide jumps while the Turtle runs). Whatever the Turtle is doing, the Guide is allowed to use his proper run rule. There is a danger here that Turtle will jump, and the Guide will only run, which is not allowed by the Achilles–Turtle machine rules. That is prevented by the relation $E_{T,T',G'}$: if Turtle remains in the same place then the Guide is allowed to go anywhere. But if he moves, then the Guide must join him.

If T is red, then the constraints allow the Guide and Turtle to go anywhere.

THEOREM 3.15. *Uniform boundedness of single recursive rule ternary DATALOG programs is undecidable.*

Proof. It follows from Theorems 2.3(1) and 3.14. □

Remark. We could use Theorem 2.3(2) instead of 2.3(1) and get more "universal" DATALOG programs. For example Theorem 3.1 would then have the following form.

There exists an arity 3 *linear DATALOG program* $\mathcal{P}$ *with one IDB predicate and a computable sequence* $\{p(N)\}$ *of initialization rules, such that the program* $\mathcal{P} \cup p(N)$ *is uniformly bounded if and only if* $C(g, n)$ *holds, where* g *is the universal Conway function from Theorem* 2.3(2).

In fact this is the form from [15]. It can not however be done in section 4, so we decided not to present the results in the most general versions but to preserve the notational uniformity instead.

4. Single rule programs.

4.1. Constants: Notational proviso. In sections 4.2–4.5 we are going to encode the computation of Achilles-Turtle machine into a very small number of rules (one or two). We can no longer afford having a separate predicate for each color. Instead, we are going to have one binary predicate COL, and understand COL(C,A) as "the color of A is C." So, instead of predicates we need to have constants to name colors.

There are no constants in DATALOG. But in fact, if we want to use some (say, k) constants we can simply increase the arity of all the IDB symbols with k and write a sequence $C_1, C_2, \ldots, C_k$ of variables as the k last arguments in each occurrence of each IDB predicate in the program. This is one of the reasons why the programs in the following sections are of high arity.

Example. The program

$$P(X,Y) \text{:-}\ P(X,Z), P(Z,Y), P(a,X), P(b,Z),$$

with constants a, b can be written as

$$P'(X,Y,A,B) \text{:-}\ P'(X,Z,A,B), P'(Z,Y,A,B), P'(A,X,A,B), P'(B,Z,A,B),$$

where $P'(X,Y,A,B)$ means "$P(X,Y)$ if the constants are understood as A, B."

Thanks to that we can suppose that there are constants in the language. We will use the constants *jump*, *run*, *joker* and constants for colours $colour_i$, for $i = 1 \ldots p{-}1$. The constant $colour_0$ will be also called *white*, $colour_1$ will be *red*, and $colour_2$ will be called *pink*.

4.2. The Achilles-Turtle game. In this section we will modify the description of the Achilles-Turtle machine and define its equivalent version with only one transition rule. To make our notation compatible with the database notation we are going to forget about the tape, and use a kind of infinite graph instead. To distinguish between the two, the version of the machine will be called the Achilles-Turtle game.

The transition rules of the Achilles-Turtle machine are indexed with three parameters: the first is either *jump* or *run*, the remaining two are the colors of the Turtle's cell and the Guide's cell before the transition. The idea of what follows is to treat the parameters as arguments occurring in the goals of the body of the single rule. While solving the first four goals of the body we will substitute proper parameters for the variables COND, TCOLOR, and GCOLOR. Then the parameters will be used to compute the positions of Achilles, Turtle, and Guide after the execution of the rule.

The following definition introduces the predicates that will be used in the construction of the single rule. We do not expect that the reader will understand the definition until he reads the proof of Lemma 4.2.

DEFINITION 4.1. *For a given Conway function g, as in Theorem 2.3(1) the Achilles–Turtle graph is the relational structure $\mathcal{G}$ with exactly the nodes and the relations listed below:*

(i) *the nodes of $\mathcal{G}$ are: $colour_0, colour_1, \ldots colour_{p-1}$, jump, run, joker and an infinite sequence of nodes $c_0, c_1, c_2, \ldots$;*

(ii) *$S(c_i, c_{i+1})$ holds for each node c_i;*

(iii) *for each i if $i \equiv j \pmod p$ (where $j \leq p-1$) then $COL(colour_j, c_i)$ holds. $COL(red, joker)$ holds;*

(iv) *$COND_1(run, c_i, joker)$ holds for each i;*
$COND_1(jump, c_i, c_{i+j})$ holds for each $i \equiv 0 \pmod p$ and for each $0 \leq j \leq p-1$;

(v) *$COND_2(run, joker, c_i)$ holds for each i;*
$COND_2(jump, c_i, c_i)$ holds for each i,
$COND_2(run, joker, joker)$ holds;

(vi) *$GRULE_1(run, colour_j, colour_k, c_i, c_{i+a_j/q_j})$ holds if $i \equiv k \pmod p$ and $p(a_j/q_j) < p-k$,*
$GRULE_1(run, colour_j, colour_k, c_i, c_{i+p-k})$ holds if $i \equiv k \pmod p$ and $p(a_j/q_j) \geq p-k$,
$GRULE_1(jump, colour_j, colour_k, c_i, c_{i+d_j})$ holds if $i \equiv k \pmod p$ and $d_j < p-k$,
$GRULE_1(jump, colour_j, colour_k, c_i, c_{i+p-k})$ holds if $i \equiv k \pmod p$ and $d_j \geq p-k$,
$GRULE_1(run, red, colour_k, c_i, joker)$ holds for all $i \equiv k \pmod p$,
$GRULE_1(jump, red, colour_k, c_i, joker)$ holds for all $i \equiv k \pmod p$,

(vii) *$GRULE_2(run, colour_j, colour_k, c_i, c_i)$ holds if $p(a_j/q_j) < p-k$,*
$GRULE_2(run, colour_j, colour_k, c_i, c_{i+p(a_j/q_j)-p+k})$
holds if $i \equiv 0 \pmod p$, $p(a_j/q_j \geq p-k$ but $p(a_j/q_j) < 2p-k$,
$GRULE_2(run, colour_j, colour_k, c_i, c_{i+p})$ holds if $i \equiv 0 \pmod p$, $p(a_j/q_j) \geq 2p-k$,
$GRULE_2(jump, colour_j, colour_k, c_i, c_i)$ holds if $k + d_j < p$,
$GRULE_2(jump, colour_j, colour_k, c_i, c_{i+d_j-p+k})$ holds if $i \equiv 0 \pmod p$, $d_j \geq p-k$ but $d_j < 2p-k$,
$GRULE_2(jump, colour_j, colour_k, c_i, c_{i+p})$ holds if $i \equiv 0 \pmod p$, $d_j \geq 2p-k$,
$GRULE_2(run, red, colour_k, joker, joker)$ holds for all k,
$GRULE_2(jump, red, colour_k, joker, joker)$ holds for all k,

(viii) *$GRULE_3(run, colour_j, colour_k, c_i, c_i)$ holds if $p(a_j/q_j) < 2p-k$,*
$GRULE_3(run, colour_j, colour_k, c_i, c_{i+p(a_j/q_j)-2p+k})$
holds if $i \equiv 0 \pmod p$, $p(a_j/q_j) \geq 2p-k$,
$GRULE_3(jump, colour_j, colour_k, c_i, c_i)$ holds if $d_j < 2p-k$,
$GRULE_3(jump, colour_j, colour_k, c_i, c_{i+d_j-2p+k})$
holds if $i \equiv 0 \pmod p$, $d_j \geq 2p-k$,
$GRULE_3(run, red, colour_k, joker, c_i)$ holds for all i,
$GRULE_3(jump, red, colour_k, joker, c_i)$ holds for all i,

(ix) *$TRULE(run, colour, c_i, c_i)$ holds for each i and for each colour,*
$TRULE(run, red, c_i, joker)$ holds for each i,
$TRULE(jump, colour, c_i, joker)$ holds for each i and for each colour,
$TRULE(run, red, joker, joker)$ holds,

(x) *$GTRULE(jump, c_i, c_i)$ holds for each i,*
$GTRULE(run, c_i, joker)$ holds for each i,
$GTRULE(run, joker, joker)$ holds.

The set of the nodes c_i, $i = 0, 1, \ldots$ of the graph can be in a natural way understood as a tape of Achilles-Turtle machine. Notice that all the facts from the definition are "local" in the sense that if some elements c_i and c_j are directly connected by a fact then $|j-i| \leq p$ and there is no white node between c_i and c_j .

Now we are going to use the relations of the Achilles-Turtle graph to encode all the rules of the machine in only one transition rule:
$CONF(A_p, T', G')$:–
$BODY$,
$S(A, A_1), S(A_1, A_2), \ldots S(A_{p-1}, A_p)$,
$CONF(A, T, G)$.

where $BODY$ is the conjunction of the following facts:

$COND_1(COND, A, X_0)$,
$COND_2(COND, X_0, T)$,
$COL(TCOLOR, T)$,
$COL(GCOLOR, G)$
$GRULE_1(COND, TCOLOR, GCOLOR, G, X_1)$,
$GRULE_2(COND, TCOLOR, GCOLOR, X_1, X_2)$,
$GRULE_3(COND, TCOLOR, GCOLOR, X_2, G')$,
$TRULE(COND, TCOLOR, T, X_3)$,
$TRULE(COND, TCOLOR, T', X_3)$,
$GTRULE(COND, G', X_4)$,
$GTRULE(COND, T', X_4)$.

LEMMA 4.2. *Suppose T is not red. Then $CON(A', T', G')$ can be computed from $CON(A, T, G)$ in a single computation step of the Achilles-Turtle machine if and only if $CONF(A', T', G')$ can be reached from $CONF(A, T, G)$ in a single step of the Achilles-Turtle graph game.*

Proof. It is clear that the move of Achilles is performed in the same way by the machine and the game (he simply moves p cells ahead). We should check that this is also the case with the Turtle and the Guide.

The "only if" direction is easier: if the transition of the machine has been done according to the *run* rule then substitute *run* for the variable $COND$. Otherwise substitute *jump*. For the variable $TCOLOR$ substitute the colour of the Turtle's cell. For the variable $GCOLOR$ substitute the colour of the Guide's cell. Now, if there is no white node between G and G', substitute $X_1 = X_2 = G'$. If there is exactly one such white node, then substitute the node for X_1 and $X_2 = G'$. If there are two such nodes, then substitute the first of them for X_1 and the second for X_2 (notice that by condition (ii) from Theorem 2.3(1) there are at most two white nodes between G and G'). The "GRULE" goals in the lines (v)–(vii) of the BODY are satisfied in this way. If $COND$ is *run*, then substitute T for X_3 (notice that in this case $T = T'$). If $COND$ is *jump*, then substitute *joker* for X_3. The two "TRULE" goals of the BODY are satisfied in this way. To satisfy the last two goals substitute *joker* for X_4 if the $COND$ is *run*, and if the $COND$ is *jump*, then substitute G' for X_4.

For the "if" direction first notice that if the conjunction of
$COND_1(jump, A, X_0)$,
$COND_2(jump, X_0, T)$,
can be satisfied, then really the distance between Achilles and the Turtle, before the transition, is smaller than p; jump is allowed according to the Achilles-Turtle machine

rules. The rules for the Guide (defined by claims (vi), (vii), and (viii) of Definition 4.1) ensure that if only COND, TCOLOR, and GCOLOR are chosen in a fair way, then the Guide of the game moves in the same way as the one from the machine. As for the Turtle, if the COND is *run*, then he should not move, and in fact the conjunction
$TRULE(run, TCOLOR, T, X_3)$,
$TRULE(run, TCOLOR, T', X_3)$
can be satisfied only if $T = T'$.

If the COND is *jump* then the Turtle should go to the same node where Guide does. The conjunction
$TRULE(jump, TCOLOR, T, X_3)$,
$TRULE(jump, TCOLOR, T', X_3)$
can be satisfied for every choice of T, T' then, if *joker* is substituted for X_3, but the conjunction
$GTRULE(jump, G', X_4)$,
$GTRULE(jump, T', X_4)$
is satisfied only if $X_4 = G' = T'$. □

The following lemma is much easier to prove than Lemma 4.2 and is left as an exercise for the reader.

LEMMA 4.3. *If c_t is red, then each configuration of the form $CONF(c_{i+p}, c_{t'}, c_{g'})$ can be reached from $CONF(c_i, c_t, c_g)$ in a single step of the graph game.*

Hint: Put $COND$ equal to *run* and $TCOLOR$ equal to *red*.

4.3. Single linear rule program with initialization. In this section we will use the Achilles-Turtle game to construct a DATALOG program with one linear recursive rule and one initialization, which is uniformly bounded if and only if $C(g, 2)$ holds.

The EDBs of the program will be the same as used in Definition 4.1.

It will be one ternary IDB CONFIG. The variables A, T, G in a fact of the form $CONFIG(A, T, G)$ should be, as usual, understood as Achilles, Turtle, and Guide.

Consider a database $\mathcal{D}$. We suppose that $colour_0, colour_1, \ldots, colour_{p-1}$, *jump*, *run* and *joker* occur in $\mathcal{D}$. A Motorway will be a sequence of elements of the database which can be used for playing the Achilles-Turtle game.

DEFINITION 4.4. *Suppose $X_0, X_1, \ldots, X_n$ are elements of $\mathcal{D}$. We say that the sequence $X_0, X_1, \ldots X_n$ is a Motorway if $\mathcal{G} \cap \{jump, run, joker, colour_0, colour_1, \ldots,$ $colour_{p-1}, c_0, c_1, \ldots, c_n\}$ is a subgraph of $\mathcal{D} \cap \{jump, run, joker, colour_0, colour_1, \ldots,$ $colour_{p-1}, X_0, X_1, \ldots, X_n\}$,*
where ordered sets of elements are considered. $\mathcal{G}$ is the database from Definition 4.1.

So, for example, we require that $S(X_3, X_4)$ and $COL(pink, X_3)$ hold in $\mathcal{D}$, if $X_0, X_1, \ldots X_n$ is a Motorway.

Now we are ready to write the following.

Linear recursive rule of program $\mathcal{T}$:
$CONFIG(A_p, T', G')$:–
$CONFIG(A, T, G)$,
Motorway$(A, A_1 \ldots A_p)$,
$BODY$,
$TRULE(run, red, T', joker)$,
$GRULE_3(run, red, colour_0, joker, G')$,
$GRULE_3(run, red, colour_1, joker, G')$,
...

$GRULE_3(run, red, colour_{p-1}, joker, G')$,
$GTRULE(run, G', joker)$,
$GTRULE(run, T', joker)$,

where Motorway$(A_, A_1, \ldots, A_p)$ is the conjunction of facts needed for the sequence $(A_, A_1, \ldots, A_p)$ to be a Motorway. Notice that the last $p+3$ lines the rule are exactly the literals of $BODY$ in which T' or G' occurs, with run substituted for $COND$ and red substituted for $TCOLOR$.

Initialization of program $\mathcal{T}$:

$CONFIG(A_0, A_2, A_2)$:–
$COL(white, A_0), S_(A_0, A_1), S(A_1, A_2)$.

Thanks to the last $p+3$ lines of the recursive rule, we can be sure that if the fact $CONFIG(A_p, T', G')$ can be proved in one step from $CONFIG(A, T, G)$, then it can also be proved in one step from each fact of the form $CONFIG(A, T'', G'')$ where T'' is red (see the proof of Lemma 4.8, case 1).

Our next goal is to show that if a long proof using the recursive rule is possible in some database $\mathcal{D}$, then there is a long Motorway in $\mathcal{D}$.

Lemma 4.5. *Consider a sequence: $A_0, A_1, A_2, \ldots, A_{xp}$ of elements of a database. If, for each $0 \leq k \leq x-1$ the subsequence $A_{kp}, A_{kp+1}, \ldots, A_{(k+1)p}$ is a Motorway, then also the whole sequence is a Motorway.*

Proof. Conditions (i)–(x) of Definition 4.1 are "local": if some elements c_i and c_j occur in a condition, then $|j-i| \leq p$ and there is no white node between c_i and c_j . □

Lemma 4.6. *Suppose that*
$CONFIG(A^0, T^0, G^0)$,
$CONFIG(A^1, T^1, G^1)$,
$CONFIG(A^2, T^2, G^2)$,
...

$CONFIG(A^l, T^l, G^l)$
is a sequence of facts, such that if $0 \leq i \leq l-1$, then
$CONFIG(A^{i+1}, T^{i+1}, G^{i+1})$
can be derived from $CONFIG(A^i, T^i, G^i)$ by a single use of the recursive rule. Then there exists a sequence
$A^0, A^0_1, \ldots A^0_{p-1}$,
$A^1, A^1_1, \ldots A^1_{p-1}$,
$A^2, A^2_1, \ldots A^2_{p-1}$,
...
$A^{l-1}, A^{l-1}_1, \ldots A^{l-1}_{p-1}, A^l$

of elements of the database which is a Motorway.

Proof. That follows from Lemma 4.5 and from the construction of the recursive rule. □

Definition 4.7. *If A, T, G is a tuple of nodes of the database $\mathcal{D}$, then we say that $CONFIG(A, T, G)$ is a fact about A.*

Now we are ready to prove that if $C(g, 2)$ holds then the program $\mathcal{T}$ is uniformly bounded.

LEMMA 4.8. *If $C(g,2)$ holds then there exists a constant C such that in every database $\mathcal{D}$ if the program $\mathcal{T}$ proves some fact, then the fact can be proved in no more than C derivation steps.*

Proof. If $C(g,2)$ holds then the Achilles-Turtle game can reach the configuration $CONF(X, S(X), S(X))$ for some white X. $S(X)$ is red then. Suppose the K moves are needed to reach this configuration. $X = S^{pK}(Y)$ for some Y then, and the nodes of the machine graph left of Y or right of $S(X)$ are not visited during the computation. We are going to prove that $K{+}2$ is a good candidate to be C.

Consider an element A of $\mathcal{D}$. There are two possibilities.

Case 1. There is a Motorway of length $(K+1)p$ in the database, such that A is its last node.

Suppose

$$A_{-(K+1)p}, A_{-(K+1)p+1}, \dots, A$$

is the Motorway. By the initialization rule

$$CONFIG(A_{-(K+1)p}, A_{-(K+1)p+2}, A_{-(K+1)p+2})$$

can be proved in one derivation step. During the next K derivation steps one can simulate K steps of Achilles-Turtle game, and so after $K{+}1$ steps we derive

$$CONFIG(A_{-p}, A_{-p+1}, A_{-p+1}).$$

Since A_{-p+1} is red one can argue as in the proof of Lemma 4.3, to see that in the next derivation step we can prove

$CONFIG(A, T', G')$

for each T' and each G' such that

$TRULE(run, red, T', joker)$,
$GRULE_3(run, red, colour_0, joker, G')$,
$GRULE_3(run, red, colour_1, joker, G)$,
...
$GRULE_3(run, red, colour_{p-1}, joker, G')$,
$GTRULE(run, G', joker)$,
$GTRULE(run, T', joker)$.

Because of the last $p+3$ lines of the recursive rule, no other facts can be proved about A.

Case 2. There is no such Motorway. Then, by Lemma 4.6, every proof has less than K+3 steps. □

We still need to show that if $C(g,2)$ does not hold, then the program is unbounded.

LEMMA 4.9. *If $C(g,2)$ does not hold, then for each constant C there exists a database $\mathcal{D}$, with empty input IDB relation, and a fact*

$$CONFIG(A, T, G)$$

which can be proved in the database but the proof requires more than C steps.

Proof. It's enough to show that arbitrarily long proofs are needed in the Achilles-Turtle game graph (we suppose that there are no IDB input facts). So start with

$$CONFIG(c_0, c_2, c_2)$$

(which can be done by initialization) and use $2C$ times the run rule for Turtle in a pink-colored cell. Notice that the position of the Turtle will remain unchanged during the computation and the final configuration will be

$$CONFIG(c_{2C}, c_2, c_{2+2C(a_2/q_2)}).$$

The shortest proof of the fact requires $2C+1$ steps (including initialization). □

To summarize, we have the following.

THEOREM 4.10. *Uniform boundedness and program boundedness are undecidable for programs consisting of one linear rule and one initialization.*

Proof. The problem *for given Conway function g, does $C(g, 2)$ hold?* is undecidable, even for functions satisfying conditions (ii) and (iii) of Theorem 2.3(1). For each such function we can construct a DATALOG program, with one linear rule and one initialization which is not program bounded if $C(g, 2)$ does not hold (Lemma 4.9) and which is uniformly bounded if $C(g, 2)$ holds (Lemma 4.8). □

4.4. Single rule program: How one cannot construct it. Now we would like to modify the construction of the previous section and get a single rule program. The only problem is how to initialize the predicate $CONFIG$. The simplest solution would be not to initialize it at all, but just check, in the same way as we use the "Motorway" goal in the body of the rule, that the needed EDB facts hold. So the rule should look like this:

$CONFIG(A_p, T', G')$:–
  $CONFIG(A, T, G)$,
  Motorway$(A, A_1 \dots A_p, A_p)$,
  $CONFIG(A, A_2, A_2)$,
  $BODY$,
  $TRULE(run, red, T', joker)$,
  $GRULE_3(run, red, colour_0, joker, G')$,
  $GRULE_3(run, red, colour_1, joker, G')$,
  ...
  $GRULE_3(run, red, colour_{p-1}, joker, G')$,
  $GTRULE(run, G', joker)$,
  $GTRULE(run, T', joker)$.

In this way, one could think, we secure that it is possible to start the computation of the Achilles-Turtle machine in each place, where any derivation step is made. But it is not enough to go in the footsteps of the proof of Lemma 4.8. We there require that the initial configuration is not only provable—which is really secured by the would-be rule above—but that it is provable in a bounded number of steps (in fact, just one step, in the previous section). We are to think of a new trick to ensure that.

4.5. Single rule program: How to construct it. The single recursive rule $\mathcal{S}$ is

$CONFIG(run, Z, A_p, T', G')$:–

$CONFIG(W, run, A, T, G)$, *main premise*

$CONFIG(jump, run, A, A_2, A_2)$, *initialization premise*

$CONFIG(jump, jump, joker, joker, joker)$, *jump=run premise*

Motorway$(A, A_1, \dots A_p)$,
Motorway$(joker, joker, \dots joker, A_p)$,

$BODY$,
$TRULE(run, red, T', joker)$,
$GRULE_3(run, red, colour_0, joker, G')$,
$GRULE_3(run, red, colour_1, joker, G')$,
$\dots$
$GRULE_3(run, red, colour_{p-1}, joker, G')$,
$GTRULE(run, G', joker)$,
$GTRULE(run, T', joker)$,

where the constant *joker* occurs p times in the "predicate" Motorway. We have added two additional arguments to the recursive predicate here. The rule asserts that if something can be derived then its first argument is *run*. Thus, if only the constants *run* and *jump* are not interpreted in the same way in the database, then the fact

$CONFIG(jump, run, A, A_2, A_2)$

cannot be proved by the program; if it is provable, then it is provable in 0 steps (is given as a part of the input).
Also the fact

$CONFIG(jump, jump, joker, joker, joker)$

does not require a deep proof: if any proof at all is possible, then the fact is given in the input.

The "jump=run premise" is normally useless as the main or the initialization premise of a derivation step: it has "jump" as the second argument. But if *run* and *jump* are equal in the database, then we use it to show that if anything can be proved about A, then everything can be proved about it in one step. That is why Motorway$(joker, joker, \dots, joker, A_p)$ must hold and why *joker* is *red*.

We use the methods of section 4.3 to prove that the constructed single rule program is uniformly bounded if and only if $C(g, 2)$ holds.

LEMMA 4.11. *If $C(g, 2)$ holds, then there exists a constant C such that in every database $\mathcal{D}$ if some fact can be proved with the rule $\mathcal{S}$, then it has a proof no deeper than C.*

Proof. Let K be like in the proof of Lemma 4.8. We need to consider two cases.

Case 1. *jump* and *run* are different elements of the database.

Suppose that for some A there is a fact about it which has a proof of length at least $K{+}2$. Then, we follow the proof of Lemma 4.8: we use the fact that the needed initialization has been given in the input, so it has a short (0-step) proof, and show that everything can be proved about A in no more than $K+2$ derivation steps.

Case 2. *jump* and *run* are interpreted as the same element of the database.

Suppose that anything can be proved about some A_p. Then

$$CONFIG(jump, jump, joker, joker, joker)$$

holds in the database. Since Motorway$(joker, joker, \ldots, joker, A_p)$ holds and since *joker* is *red*, every fact of the form

$$CONFIG(run, Z, A_p, T', G')$$

can be proved in one derivation step if only
$TRULE(run, red, T', joker)$,
$GRULE_3(run, red, colour_0, joker, G')$,
$GRULE_3(run, red, colour_1, joker, G')$,
...
$GRULE_3(run, red, colour_{p-1}, joker, G')$,
$GTRULE(run, G', joker)$,
$GTRULE(run, T', joker)$. □

LEMMA 4.12. *If $C(g, 2)$ does not hold, then for each constant C there exist a database $\mathcal{D}$, and a fact*

$$CONFIG(run, run, A_0, T, G)$$

which can be proved, with the rule $\mathcal{S}$, in the database $\mathcal{D}$, but the proof requires more than C steps.

Proof. We proceed in a similar way as in the proof of Lemma 4.9, with the following differences:

(i) We no longer assume that the IDB input is empty. Instead, we require that there are the following $CONFIG$ facts in the input:

$$CONFIG(jump, jump, joker, joker, joker),$$

and, for each $x \leq C$,

$$CONFIG(jump, run, c_{px}, c_{px+2}, c_{px+2}).$$

(ii) We require that for each $x \leq C$,

$$\text{Motorway}(joker, joker, \ldots, joker, A_{px})$$

holds. This ends the proof of the following. □

THEOREM 4.13. *Uniform boundedness of single rule DATALOG programs is undecidable.*

REFERENCES

[1] Aristotle, *Physics*, VI, 239 b 5-240 a 18.

[2] S. Abiteboul, *Boundedness is undecidable for datalog programs with a single recursive rule*, Inform. Process. Lett., 32 (1989) pp. 281–287.

[3] M. Ajtai and Y. Gurevich, *DATALOG versus First Order Logic*, in Proceedings of 30th FOCS, 1989.

[4] J.H. Conway, *Unpredictable Iterations*, in Proceedings of 1972 Number Theory Conference, University of Colorado, 1972, pp. 49–52.

[5] S.S. Cosmadakis and P. C. Kanellakis, *Parallel evaluation of recursive rule queries*, in Proceedings of 5th ACM PODS, ACM, New York, 1986, pp. 280–293.

[6] S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis, and M.Y. Vardi, *Decidable optimization problems for database logic programs*, in Proceedings of 20th ACM STOC, 1988, pp. 477–490.

[7] P. Devienne, P. Lebègue, and J.C. Routier, *Halting problem of one binary recursive horn clause is undecidable*, in Proceedings of STACS '93, Springer-Verlag, New York, 1993.

[8] H. Gaifman, H. G. Mairson, Y. Sagiv, and M.Y. Vardi, *Undecidable optimization problems for database logic programs*, J. ACM, 40 (1993), pp. 683–713.

[9] G.G. Hillebrand, P.C. Kanellakis, H.G. Mairson, and M.Y. Vardi, *Undecidable boundedness problems for datalog programs*, in Proceedings of 10th PODS, 1991.

[10] Y.E. Ioanidis, *A time bound on the materialization of some recursively defined views*, in Proceedings of the 85 VLDB.

[11] P.C. Kanellakis, *Logic programming and parallel complexity*, in Foundations of Deductive Databases and Logic Programming, J. Minker, ed., Morgan Kaufman, San Francisco, 1988, pp. 547–586.

[12] P.C. Kanellakis, *Elements of relational database theory*, in Handbook of Theoretical Computer Science, B, J. van Leeuven, ed., North-Holland, Amsterdam, 1990.

[13] D.C. Kozen, *Automata and Computability*, Springer-Verlag, New York, 1997, p. 224.

[14] J.C. Lagarias, *The $3x + 1$ problem and its generalizations*, Amer. Math. Monthly, 92 (1985), pp. 3–23.

[15] J. Marcinkowski, *The 3 Frenchmen method proves undecidability of the uniform boundedness for single recursive rule ternary DATALOG programs*, in Proceedings of 13th STACS, Lecture Notes in Comput. Sci. 1046, Springer-Verlag, New York, pp. 427–438.

[16] J. Marcinkowski, *Undecidability of uniform boundedness for single rule datalog programs*, in Proceedings of the 11th IEEE Symposium on Logic in Computer Science, pp. 13–24.

[17] J. Minker and J.M. Nicolas, *On recursive axioms in relational databases*, Information Systems, 8 (1982), pp. 1–13.

[18] J. Naughton, *Data independent recursion in deductive databases*, in Proceedings of 5th PODS, 1986.

[19] J. Naughton and Y. Sagiv, *A decidable class of bounded recursions*, in Proceedings of 6th PODS, 1987.

[20] Y. Sagiv, *On computing restricted projections of representative instances*, in Proceedings of 4th PODS, 1985.

[21] Y. Sagiv, *Optimizing datalog programs*, in Foundations of Deductive Databases and Logic Programming, J. Minker, ed., Morgan Kaufman, San Francisco, 1988, pp. 659–698.

[22] O. Shmueli, *Decidability and expressiveness aspects of logic queries*, in Proceedings of 7th PODS, 1988.

[23] K. Wang, *Some positive results for boundedness of multiple recursive rules*, in 95 International Conference on Database Theory, 1995.

[24] M.Y. Vardi, *Decidability and undecidability results for boundedness of linear recursive queries*, in Proceedings of 8th PODS, ACM, New York, 1988, pp. 341–351.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 258–273

TIGHT BOUNDS ON THE SIZE OF FAULT-TOLERANT MERGING AND SORTING NETWORKS WITH DESTRUCTIVE FAULTS[*]

TOM LEIGHTON[†] AND YUAN MA[‡]

Abstract. We study networks that can sort n items even when a large number of the comparators in the network are faulty. We restrict our attention to networks that consist of registers, comparators, and replicators. (*Replicators* are used to copy an item from one register to another, and they are assumed to be fault free.) We consider the scenario of both random and worst-case comparator faults, and we follow the general model of *destructive* comparator failure proposed by Assaf and Upfal [*Proc. 31st IEEE Symposium on Foundations of Computer Science*, St. Louis, MO, 1990, pp. 275–284] in which the two outputs of a faulty comparator can fail independently of each other.

In the case of random faults, Assaf and Upfal showed how to construct a network with $O(n \log^2 n)$ comparators that (with high probability) can sort n items even if a constant fraction of the comparators are faulty. Whether the bound on the number of comparators can be improved (to, say, $O(n \log n)$) for sorting (or merging) has remained an interesting open question. We resolve this question in this paper by proving that any n-item sorting or merging network which can tolerate a constant fraction of random failures has $\Omega(n \log^2 n)$ comparators.

In the case of worst-case faults, we show that $\Omega(kn \log n)$ comparators are necessary to construct a sorting or merging network that can tolerate up to k worst-case faults. We also show that this bound is tight for $k = O(\log n)$. The lower bound is particularly significant since it formally proves that the cost of being tolerant to worst-case failures is very high.

Both the lower bound for random faults and the lower bound for worst-case faults are the first nontrivial lower bounds on the size of a fault-tolerant sorting or merging network.

Key words. merging, sorting, circuits, comparator networks, fault-tolerance, lower bounds, probabilistic analysis of algorithms

AMS subject classifications. 68Q22, 68Q25, 94C99

PII. S0097539796305298

1. Introduction.

In the classic model of a *sorting circuit* analyzed by Knuth [6], Batcher [3], and Ajtai, Komlós, and Szemerédi [1], the circuit consists of n registers and a collection of comparators, where n is the number of items to be sorted. Each *register* holds one of the items to be sorted, and each *comparator* is a 2-input, 2-output device that outputs the two input items in sorted order. The comparators are partitioned into *levels* so that each register is involved in at most one comparison in each level. The *depth* of the circuit is defined to be the number of levels in the circuit, and the *size* of the circuit is defined to be the number of comparators in the circuit. For example, a 4-item sorting circuit with depth 5 and size 6 is shown in Figure 1.1.

Sorting circuits have numerous applications in the context of message routing and switching [9] and they have been intensively studied for several decades. In the past

[*]Received by the editors June 18, 1996; accepted for publication (in revised form) February 19, 1998; published electronically September 22, 1999. This research was supported by AFOSR contract F49620-92-J-0125, DARPA contract N00014-91-J-1698, and DARPA contract N00014-92-J-1799. A preliminary version of this paper appeared in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 30–41.

http://www.siam.org/journals/sicomp/29-1/30529.html

[†]Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (ftl@math.mit.edu).

[‡]Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. The research of this author was also supported by an NSF Mathematical Sciences Postdoctoral Research Fellowship while he was at Stanford University. Current address: Haas School of Business, University of California, Berkeley, CA, 94720 (yuan@haas.berkeley.edu).

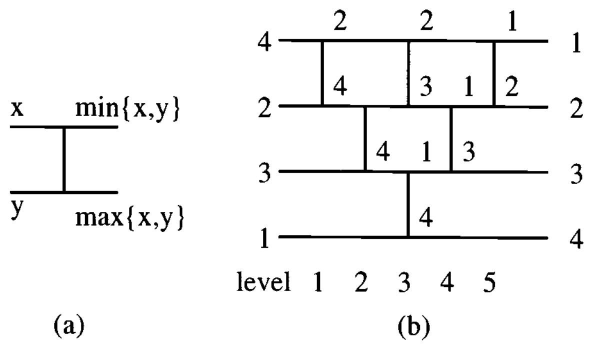


FIG. 1.1. (a) *A comparator.* (b) *A sorting circuit.*

several years, issues involving the fault-tolerance properties of sorting circuits have gained increased importance and attention (see [2, 4, 7, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]).

Thus far, the work on developing sorting circuits that are tolerant to faults has focussed on three basic types of failures: passive comparator faults, reversal comparator faults, and destructive comparator faults. In the *passive fault model*, a faulty comparator outputs its two input items in the same order in which they are input (i.e., without comparison). In the *reversal fault model*, a faulty comparator always outputs its input items in the wrong order but never loses any of the items.

In this paper, we consider the stronger "destructive" model of comparator failure proposed by Assaf and Upfal [2]. In the *destructive fault model*, a faulty comparator with inputs x and y can output $f(x,y)$ and $g(x,y)$, where f and g can be any of the following functions: x, y, $\min(x,y)$, or $\max(x,y)$. The destructive model of failure allows for any form of miswiring or short circuit. The only restriction is that each output be one of the inputs. (Without this constraint, it would not be possible to sort with high probability since for any circuit a single fault immediately before an output could make the output incorrect.)

In the destructive model of comparator failure, it is possible for one of the two inputs to a faulty comparator to be destroyed during the comparison. (For example, this happens to input y when $f(x,y) = g(x,y) = x$.) Hence, Assaf and Upfal made use of replicators when constructing networks that are tolerant to destructive faults. A *replicator* simply copies the content of one register to another. For example, a 2-item sorting network with replicators that is tolerant to any single fault is illustrated in Figure 1.2. In this paper (as in [2]), we will assume that the replicators are fault free. This is not a particularly unreasonable assumption since replicators can be hardwired and they do not contain any logic elements.

In [2], Assaf and Upfal described a general method for converting any sorting circuit into a sorting network (with replicators) that (with high probability) is tolerant to random faults in the comparators. In particular, given an n-item sorting circuit with depth d and size s, the fault-tolerant network produced by the Assaf–Upfal transformation has depth $O(d)$ and size $O(s \log n)$, and it is able to sort (with probability at least $1 - \frac{1}{n}$) even if each comparator independently suffers a destructive failure with some constant probability.[1] (The *size* of a network is defined to be the number of comparators. Asymptotically it makes no difference whether the replica-

[1]In this paper, all the logarithms are base 2 unless specified otherwise.

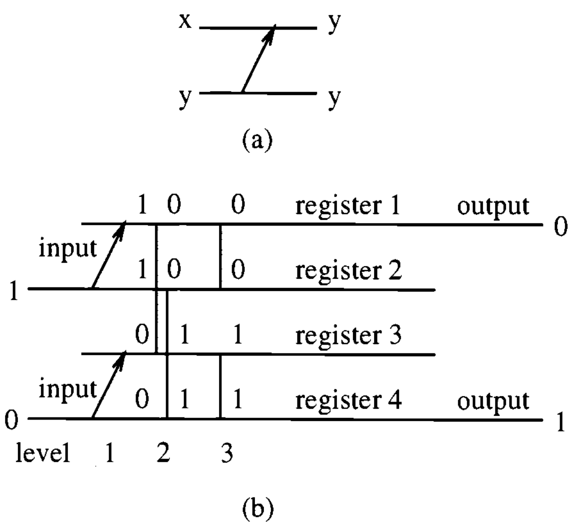


FIG. 1.2. (a) *A replicator.* (b) *A 2-item sorting network that tolerates any single fault.*

tors are counted since an optimal network should make copies of an item only if that item will be input to a comparator.) The network will also have width $O(n \log n)$, where the *width* of a network is defined to be the number of registers it contains. When used in conjunction with the AKS sorting circuit of [1], this provides a sorting network with depth $O(\log n)$, width $O(n \log n)$, and size $O(n \log^2 n)$ that is tolerant to random destructive comparator failures.

The Assaf–Upfal method proceeds by making $\Theta(\log n)$ copies of each item and replacing each comparator with $\Theta(\log n)$ comparators, followed by a majority-enhancing device that is constructed from an expander. As a consequence, the width and size of the resulting network are increased by a factor of $\Theta(\log n)$. Whether there is an alternative approach to fault tolerance that can avoid the $\Theta(\log n)$ factor blowup in width and/or size has remained an interesting open question. We resolve this question by showing that the width in the Assaf–Upfal network can be decreased, but only at the expense of a proportional increase in depth. In particular, for any $n \le w \le n \log n$, we show how to construct an n-item sorting network with width $O(w)$ and depth $O(\frac{n \log^2 n}{w})$ that can tolerate random destructive faults with high probability. More important, we show that any n-item sorting network that can tolerate random destructive faults (with any reasonable probability) has size $\Omega(n \log^2 n)$. (Somewhat surprisingly, we also show that the same lower bound holds for fault-tolerant merging networks.) These results provide tight bounds on the size and width of fault-tolerant sorting (and merging) networks, and they provide the first nontrivial lower bounds on the size of fault-tolerant sorting (and merging) networks. Compared with the recent results on passive and reversal faults [10, 13], our $\Omega(n \log^2 n)$ lower bound separates the size complexity of sorting and merging networks with destructive faults from those with passive or reversal faults.

We also consider the scenario of worst-case destructive faults. In particular, we prove a $\frac{(k+1)n \log n}{4}$ lower bound on the size of sorting and merging networks that can tolerate up to k worst-case destructive faults. This is the first nontrivial lower bound on sorting or merging networks that can tolerate worst-case faults, and it is tight (up to constant factors) for $k = O(\log n)$. Unfortunately, the lower bound means that building sorting and merging networks that are tolerant to worst-case faults is very costly. In particular, we must build the equivalent (in terms of size) of $\Omega(k)$ copies of an optimal sorting network in order to withstand k worst-case faults.

The remainder of the paper is organized into sections as follows. We start in section 2 by proving the $\Omega(kn \log n)$ lower bound on the size of merging networks that can tolerate up to k worst-case faults. We then extend this result in section 3 to prove the $\Omega(n \log^2 n)$ lower bound on the size of merging networks that are tolerant to random faults. The material in section 3 represents the most difficult and important contribution of the paper. The upper bound result on the width of fault-tolerant sorting networks is presented in section 4.

2. The lower bound on size for worst-case faults.

In this section, we prove a lower bound on the size of merging and sorting networks that are tolerant to worst-case faults. For ease of reference, we define a *k-destructive-fault-tolerant* sorting (or merging) network to be a network that is still a sorting (or merging) network even if any k (or fewer) comparators suffer any form of destructive failure. Our main result in this section is a proof of the following theorem.

THEOREM 2.1. *The size of any k-destructive-fault-tolerant merging (or sorting) network is at least* $\frac{(k+1)n \log n}{4}$.

We will prove Theorem 2.1 by showing a lower bound for merging networks. Without loss of generality, we will assume that n is an exact power of two. In what follows, we will use $\mathcal{M}$ to denote a k-destructive-fault-tolerant merging network that takes two sorted lists $X = (x_1 \le x_2 \le \cdots \le x_{n/2})$ and $Y = (y_1 \le y_2 \le \cdots \le y_{n/2})$ as input and that outputs the merged list. Without loss of generality, this means that $\mathcal{M}$ sorts lists of the form $(x_1, y_1, x_2, y_2, \ldots, x_{n/2}, y_{n/2})$ where $x_1 \le x_2 \le \cdots \le x_{n/2}$ and $y_1 \le y_2 \le \cdots \le y_{n/2}$.

To show that $\mathcal{M}$ has $\frac{(k+1)n \log n}{4}$ comparators, we need the following definitions and lemmas, some of which are extensions of those used by Floyd to prove that a fault-free merging network needs $\Omega(n \log n)$ comparators (see pages 230–232 of [6]).

Given a merging network $\mathcal{M}$ with *fault pattern* F (i.e., F specifies which comparators, if any, are faulty and how they fail) and a list of integer inputs $\Pi = (\pi_1, \pi_2, \ldots, \pi_n)$ to $\mathcal{M}$, we denote the *content* of a register r immediately after level t by $\mathcal{C}(r, t)$. For example, $\mathcal{C}(3, 1) = 0$ and $\mathcal{C}(3, 2) = \mathcal{C}(3, 3) = 1$ in Figure 1.2. We define the *history* of $\mathcal{M}$ given F and Π to be the collection of register contents $\mathcal{C}(r, t)$ taken over all registers and all levels. The following lemma shows how the history of a network computation can be influenced by a fault at a single output of a comparator.

LEMMA 2.2. *Given any* $\mathcal{M}$, Π, F, *and* F', *let* $\mathcal{C}(r, t)$ *denote the content of register* r *immediately after level* t *for* $\mathcal{M}$ *given* F *and* Π, *and let* $\mathcal{C}'(r, t)$ *denote the content of register* r *immediately after level* t *for* $\mathcal{M}$ *given* F' *and* Π. *If* F' *is identical to* F *except that one comparator* C *on level* l *with output registers* p *and* q *is modified in* F' *so that* $\mathcal{C}'(q, l) = \mathcal{C}(q, l)$ *and* $\mathcal{C}'(p, l) \ne \mathcal{C}(p, l)$, *then for all* r *and* t

(1) *if* $\mathcal{C}(r, t) < \mathcal{C}(p, l)$, *then* $\mathcal{C}'(r, t) \le \mathcal{C}(r, t)$,

(2) *if* $\mathcal{C}(r, t) > \mathcal{C}(p, l)$, *then* $\mathcal{C}'(r, t) \ge \mathcal{C}(r, t)$.

Proof. For simplicity, let $s = \mathcal{C}(p, l)$ and $s' = \mathcal{C}'(p, l)$. We will only prove the lemma for the case in which $s' < s$. The case in which $s' > s$ can be proved similarly.

We will think of F' as the result of modifying $\mathcal{C}$ in F and prove that some properties hold when such a modification is made. The network can be divided into two parts: the part before (and including) level l and the part after level l. The former part clearly remains unchanged after C is modified, and the latter part is changed as if one input item to that part were modified. Hence, it suffices to show that the following properties (a) and (b) hold when some input item is changed from s to some $s' < s$ but no comparator is modified. (We will use the term *register segment* to denote the part of a register that is between two consecutive levels.)

(a) The content of each register segment whose content was less than s before the modification is not increased.

(b) The content of each register segment whose content was greater than s before the modification remains unchanged.

In a given history, if an input item is changed from s to $s-1$ (but no comparator is modified), then all the input items that are neither s nor $s-1$ move in the network as they did before. Hence, all the register segments containing neither s nor $s-1$ before the modification now contain the same values as before. All the register segments containing $s-1$ before the modification now contain $s-1$, while some register segments containing s before the modification may contain $s-1$ now. This means that properties (a) and (b) hold when an input s is changed to $s-1$. If the new input item $s-1$ (the one that was s before) is further changed to $s-2$, then the only new change in the history is that some register segments containing $s-1$ before the modification may contain $s-2$ now. Overall, an input item has been changed from s to $s-2$ and properties (a) and (b) still hold. Since both s and s' are integers, this process can be continued until an input has been changed from s to s'. During the whole process, properties (a) and (b) are never violated. $\square$

Consider the history of $\mathcal{M}$ on a particular 0–1 input sequence π. A *crossing comparator* of $\mathcal{M}$ with respect to π is defined to be a comparator whose input contents are $\{0,1\}$ (i.e., they are not both 0 or both 1).[2] $\mathcal{M}_0(\pi)$, a subnetwork of $\mathcal{M}$ with respect to π, is constructed as follows. (The subscript 0 is used to denote that $\mathcal{M}_0(\pi)$ is the part of $\mathcal{M}$ that contains 0 on input sequence π.) Take all the register segments and replicators that contain 0 and all the comparators with both inputs containing 0. Replace each crossing comparator of $\mathcal{M}$ with respect to π by connecting directly its (unique) input containing 0 and its (unique) output containing 0. $\mathcal{M}_1(\pi)$ can be constructed in a similar fashion. For example, when $\mathcal{M}$ and π are as shown in Figure 1.2(b), $\mathcal{M}_0(\pi)$ and $\mathcal{M}_1(\pi)$ are as illustrated in Figure 2.1.

Unless specified otherwise, we will be particularly interested in the history of $\mathcal{M}$ (when there is *no* fault) on the input sequence

$$(\underbrace{0,0,\ldots,0}_{n/2},\underbrace{1,1,\ldots,1}_{n/2}). \tag{2.1}$$

Therefore, when we talk about crossing comparators, $\mathcal{M}_0$, and $\mathcal{M}_1$ without specifying π, π should be interpreted as the 0–1 sequence given in (2.1). In particular, to construct $\mathcal{M}_0$ and $\mathcal{M}_1$, we input the smallest $\frac{n}{2}$ items to the top half of $\mathcal{M}$ and the largest $\frac{n}{2}$ items to the bottom half of $\mathcal{M}$. Hence, when there is no fault, the smallest $\frac{n}{2}$ items should be contained in $\mathcal{M}_0$ and the largest $\frac{n}{2}$ items should be contained in $\mathcal{M}_1$. The motivation for defining crossing comparators, $\mathcal{M}_0$, and $\mathcal{M}_1$ can be found in the following lemmas.

LEMMA 2.3. *If $\mathcal{M}$ is an n-input k-destructive-fault-tolerant merging network, then both $\mathcal{M}_0$ and $\mathcal{M}_1$ are $\frac{n}{2}$-input k-destructive-fault-tolerant merging networks.*

[2]Here, the notion of a crossing comparator has nothing to do with any fault pattern. In the next section, however, the notion of a crossing comparator will be slightly modified so that it will be dependent on a fault pattern.

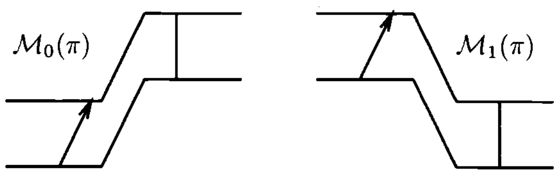


FIG. 2.1. *The decomposition of the network $\mathcal{M}$ in Figure* 1.2*, with respect to the input sequence therein, into $\mathcal{M}_0(\pi)$ and $\mathcal{M}_1(\pi)$. The sloping lines represent direct connections that replace crossing comparators.*

Proof. If we input to $\mathcal{M}$ any sequence

$$(x_1, y_1, x_2, y_2, \ldots, x_{n/4}, y_{n/4}, \underbrace{+\infty, \ldots, +\infty}_{n/2})$$

such that $x_1 \le x_2 \le \cdots \le x_{n/4}$ and $y_1 \le y_2 \le \cdots \le y_{n/4}$ and if none of the crossing comparators of $\mathcal{M}$ are faulty, then, by the definition of $\mathcal{M}_0$, the x's and y's should move within $\mathcal{M}_0$ only and $\mathcal{M}_1$ has no impact on them. Hence, $\mathcal{M}_0$ is a k-destructive-fault-tolerant merging network. (Note that an adversary can put no faults at crossing comparators and put up to k faults into $\mathcal{M}_0$.) By a similar argument, we can show $\mathcal{M}_1$ is also a k-destructive-fault-tolerant merging network. □

LEMMA 2.4. *If $\mathcal{M}$ is a k-destructive-fault-tolerant merging network, then $\mathcal{M}$ has at least $\frac{(k+1)n}{4}$ crossing comparators.*

Proof. We will focus on the history of $\mathcal{M}$ on input sequence

$$(1, +\infty, 2, +\infty, \ldots, n/2, +\infty).$$

According to the definition of $\mathcal{M}_0$, $1, 2, \ldots, \frac{n}{2}$ should all be output in the outputs of $\mathcal{M}_0$. In particular, $\frac{n}{4}+1, \ldots, \frac{n}{2}$ should be moved from $\mathcal{M}_1$ to $\mathcal{M}_0$. By definition, each crossing comparator has exactly one $\mathcal{M}_0$-input and one $\mathcal{M}_1$-input. (If an input (output) of a comparator is in $\mathcal{M}_0$, then we call it an $\mathcal{M}_0$-input (output); if an input (output) of a comparator is in $\mathcal{M}_1$, then we call it an $\mathcal{M}_1$-input (output).) Label each crossing comparator by its $\mathcal{M}_1$-input.

Assume for the purposes of contradiction that $\mathcal{M}$ contains less than $\frac{(k+1)n}{4}$ crossing comparators. Then there exists an integer s such that $\frac{n}{4}+1 \le s \le \frac{n}{2}$ and the total number of crossing comparators labeled by s is at most k.

Let C_1 be a crossing comparator labeled by s that is at the lowest level in $\mathcal{M}$. Since s cannot have gotten into $\mathcal{M}_0$ without using a crossing comparator labeled by s, we know that the $\mathcal{M}_0$-input to C_1 does not contain s. Moreover, since C_1 is not faulty in the fault-free network $\mathcal{M}$, one of its outputs contains s and the other output does not contain s. Hence, if we make C_1 be faulty by forcing the $\mathcal{M}_0$-input of C_1 to appear in both outputs of C_1, this will have the effect of replacing the value of s in one of the output registers with a value other than s, which is exactly the scenario described by Lemma 2.2. In addition, C_1 can no longer be used to move s from $\mathcal{M}_1$ to $\mathcal{M}_0$.

For any r and t, by Lemma 2.2 we know that if $C(r,t) \ne s$ before C_1 is made faulty, then $C(r,t) \ne s$ after C_1 is made faulty. Hence, the number of working crossing comparators labeled by s decreases by at least 1 when C_1 is made faulty.

We next relabel crossing comparators based on the new history when C_1 is faulty and proceed inductively (i.e., we select the next crossing comparator labeled by s, make it faulty, and so forth). The proceeding process terminates when there are no longer any functioning crossing comparators labeled by s. This is accomplished by making at most k crossing comparators faulty. Since there are no longer any crossing comparators that can move s from $\mathcal{M}_1$ to $\mathcal{M}_0$, the network does not successfully complete the merge. Hence, we can conclude that $\mathcal{M}$ has at least $\frac{(k+1)n}{4}$ crossing comparators. $\square$

Proof of Theorem 2.1. For any fixed k, let $S(n)$ denote the size of the smallest n-input k-destructive-fault-tolerant merging network. From the definition of crossing comparators, the construction of $\mathcal{M}_0$ and $\mathcal{M}_1$, and the fact that no crossing comparator in $\mathcal{M}$ appears as a comparator in either $\mathcal{M}_0$ or $\mathcal{M}_1$, we know that

$$\begin{aligned}\text{size}(\mathcal{M}) \geq\ & \text{size}(\mathcal{M}_0) + \text{size}(\mathcal{M}_1)\\ & + |\{\text{crossing comparators in } \mathcal{M}\}|.\end{aligned}$$

By Lemmas 2.3 and 2.4, this means that

$$S(n) \geq 2S(\frac{n}{2}) + \frac{(k+1)n}{4}$$

for $n \geq 4$. Solving the recurrence, we find that

$$S(n) \geq \frac{n}{2} \cdot S(2) + \frac{(k+1)n\log(n/2)}{4}.$$

Since $S(2) \geq k+1$, this means that $S(n) \geq \frac{(k+1)n\log n}{4}$, as claimed. $\square$

3. The lower bound on size for random faults. In this section, we prove a lower bound on the size of merging and sorting networks that are tolerant to random faults. For ease of reference, we define a (ρ, ϵ)-*destructive-fault-tolerant* sorting (or merging) network to be a network that is still a sorting (or merging) network with probability at least $1-\epsilon$ even if each comparator independently suffers a destructive failure with fixed probability ρ. When a destructive fault does occur, we will assume that each form of failure is equally likely. In other words, the probability that a particular form of failure appears at a comparator is $\rho_0 = \frac{\rho}{\alpha}$, where the constant α denotes the total number of possible ways that a comparator can fail. The main result in this section, which is also the main result in this paper, is a proof of the following theorem.

THEOREM 3.1. *The size of any (ρ, ϵ)-destructive-fault-tolerant merging (or sorting) network is*

$$\Omega\left(n\log n \frac{\log\frac{1}{\epsilon} + \log\sqrt{n} - \log\log_e\frac{1}{1-\epsilon}}{1-\log\rho}\right). \tag{3.1}$$

This theorem gives a good lower bound for a large range of ρ and ϵ and it does not require either ρ or ϵ to be a constant. The most interesting case, which people have studied most often, is that ρ is a nonzero constant and $\epsilon = 1/\text{poly}(n)$. In this case, the theorem gives a lower bound of $\Omega(n\log^2 n)$, which is tight [2]. Somewhat surprisingly, however, the theorem gives the same $\Omega(n\log^2 n)$ lower bound even when ϵ is not small. In particular, when ρ is a nonzero constant, the theorem implies the $\Omega(n\log^2 n)$ lower bound even for some extremely small success probability like $1-\epsilon = e^{-n^{\frac{1}{4}}}$. Hence, even networks that have a tiny chance of surviving the faults have $\Omega(n\log^2 n)$ comparators.

In fact, we will show a lower bound of the form

$$\Omega\left(n\log n\frac{\log\frac{1}{\epsilon}+\log\sqrt{n}-\log\log_e\frac{1}{1-\epsilon}}{\log\frac{1}{\rho_0}}\right), \tag{3.2}$$

where ρ_0 is the probability that a comparator suffers a particular form of destructive failure. (One can easily check that lower bound (3.2) implies lower bound (3.1). The term "1" in the denominator of (3.1) prevents the lower bound from going to $+\infty$ when ρ goes to 1.) We will prove lower bound (3.2) by showing a lower bound of

$$\frac{n\log n\log\epsilon}{4\log\rho_0} \tag{3.3}$$

and a lower bound of

$$\frac{n\log n(\log\sqrt{n}-\log\log_e\frac{1}{1-\epsilon})}{4\log\frac{1}{\rho_0}}. \tag{3.4}$$

When $\epsilon = o(1/\text{poly}(n))$, lower bound 3.4 is stronger, and when $\epsilon = \Omega(1/\text{poly}(n))$, lower bound 3.3 is stronger.

For worst-case faults, we have shown that a factor of $\Omega(k)$ in redundancy is necessary to tolerate k faults. In fact, if the size of the network is not large enough, then there exists a set of at most k comparators that are critical. If all these comparators fail in a particular way (and all other comparators work correctly), then the network will not work correctly. At first glance, one might think that this immediately implies that a factor of $\Omega(\frac{\log\epsilon}{\log\rho_0})$ in redundancy is required for the random fault case, since any set of $\Omega(\frac{\log\epsilon}{\log\rho_0})$ comparators will all fail in the necessary way with probability ϵ. If this were true, lower bound 3.3 would follow immediately. However, the problem for random faults is that we cannot assume that all "other" comparators work correctly. Indeed, it is likely that a large fraction of those "other" comparators will be faulty, and these faulty comparators might inadvertently help the network work correctly. One such interesting example is the butterfly-based fault-tolerant sorting circuit designed in [11] in which a fixed fraction of the comparator failures are crucial for the circuit to work! (To understand why fixed failure probability can be helpful, one can imagine that each comparator flips a coin to decide its behavior. Then the failures can be used as a source of randomness, which might be exploited in the network.)

In general, there are two types of lower bound results for random-fault-tolerant computation: one for the case of fixed failure probability and the other one for the case that only an upper bound is enforced on the failure probability. As argued by Pippenger [15] in the context of fault-tolerant Boolean circuits, the first type of lower bound is always stronger than the second type. Our proof technique for worst-case faults implies a lower bound of the second type for random faults. In order to get the stronger lower bound (of the first type) stated in Theorem 3.1, we need to do much more work.

As a direct application of the proof technique for worst-case faults, if the size of $\mathcal{M}$ does not satisfy lower bound 3.3, then, for any faulty $\mathcal{M}$, we can always produce another faulty network that does not work correctly by forcing at most $\log_{\rho_0}\epsilon = \frac{\log\epsilon}{\log_{\rho_0}}$ comparators to be faulty in a particular way. It is natural to ask if this property is strong enough to show that there is a large fraction of faulty networks that do not work correctly. This can be formulated as the following question. Let S be the set of all sequences with fixed length l and S_0 be a subset of S. (S will correspond to all fault patterns for a network $\mathcal{M}$, and S_0 will correspond to those fault patterns that keep $\mathcal{M}$ from being a merging network.) If the union of all the hamming balls with origins in S_0 and radius $\log_{\rho_0} \epsilon$ covers S, can we prove $\frac{|S_0|}{|S|} = \Omega(\epsilon)$? That is, if every fault pattern is within $\log_{\rho_0} \epsilon$ faults of a bad fault pattern, is the density of bad fault patterns $\Omega(\epsilon)$? Unfortunately, the answer to this question is in general "No." Hence, to prove our lower bound, we need a better understanding of the structure of the bad fault patterns.

Before proving the theorem, we first clarify some terminology to be used in the proof. We will use the terms *fault pattern* and *history* as defined in section 2. The notation $\mathcal{M}(F)$ will be used to denote the faulty network derived from $\mathcal{M}$ by setting the behavior of each comparator in $\mathcal{M}$ according to fault pattern F. In the proof, we will focus on a particular merging network $\mathcal{M}$ only. Hence there is a one-to-one correspondence between each fault pattern F and the faulty network $\mathcal{M}(F)$. We call a fault pattern F *good* if $\mathcal{M}(F)$ functions correctly as a merging network, and we call F *bad* otherwise. As in section 2, we want to decompose $\mathcal{M}$ into smaller networks and analyze the behavior of the comparators that connect these small networks. However, unlike in section 2, where we use the fault-free $\mathcal{M}$ to define these terminologies, we need to deal with different decompositions and different sets of crossing comparators for different fault patterns. We will use the history of $\mathcal{M}(F)$ on input sequence

$$(\underbrace{0, 0, \ldots, 0}_{n/2}, \underbrace{1, 1, \ldots, 1}_{n/2})$$

to redefine these terminologies as follows. A *crossing comparator* of $\mathcal{M}(F)$ is defined to be a comparator whose input contents are $\{0, 1\}$ (i.e., they are not both 0 or both 1). $\mathcal{M}(F)_0$ is constructed as follows. Take all the register segments and replicators that contain 0 and all the comparators with both inputs containing 0. For each crossing comparator with one output containing 0 and the other output containing 1, replace it in $\mathcal{M}(F)_0$ by directly connecting the (unique) input containing 0 to the (unique) output containing 0. For each crossing comparator with both outputs containing 0 (because of a fault), replace it in $\mathcal{M}(F)_0$ by a replicator that copies from the (unique) input register containing 0 to the other (output) register. (We do not include anything in $\mathcal{M}(F)_0$ to represent crossing comparators with both outputs containing 1.) $\mathcal{M}(F)_1$ can be constructed in a similar fashion.

Given $\mathcal{M}(F)_0$, we can construct $\mathcal{M}(F)_{00}$ and $\mathcal{M}(F)_{01}$, and given $\mathcal{M}(F)_1$, we can construct $\mathcal{M}(F)_{10}$ and $\mathcal{M}(F)_{11}$. Working recursively, we can construct $\mathcal{M}(F)_i$ in a similar way for any binary number i with less than $\log n$ bits.

For a fixed $\mathcal{M}$, we can define a partial order on the set of all the comparators in $\mathcal{M}$ by the following rule:

$$C_1 \prec C_2 \quad \text{if and only if} \quad \text{depth}(C_1) < \text{depth}(C_2).$$

We can extend this partial order into a total order. It does not matter how we extend, but we will stick to a fixed extension in the proof.

Proof of Theorem 3.1. We start by showing that for any (ρ, ϵ)-fault-tolerant merging network $\mathcal{M}$,

$$\text{size}(\mathcal{M}) \geq \frac{n \log n \log \epsilon}{4 \log \rho_0}. \tag{3.5}$$

Assume for the purposes of contradiction that (3.5) is not true for $\mathcal{M}$. Then we will prove that

$$\text{Prob}[F \text{ is bad for } \mathcal{M}] > \epsilon, \tag{3.6}$$

where F denotes a randomly generated fault pattern for $\mathcal{M}$. We will prove (3.6) by partitioning the space of all possible fault patterns into small groups and then showing that, within each group, a bad fault pattern will be generated with probability more than ϵ.

Now we will do the most difficult part of the proof, which is to find an appropriate partitioning of the space of the fault patterns. On any fault pattern F, we will use the following 3-step procedure to choose a characteristic set of F, which is denoted as *Char*(F). The set consists of comparators and it will be the basis for us to define the partition.

Step 1. List all the $\mathcal{M}(F)_i$'s as follows:

$$\mathcal{M}(F), \mathcal{M}(F)_0, \mathcal{M}(F)_1, \mathcal{M}(F)_{00}, \mathcal{M}(F)_{01}, \mathcal{M}(F)_{10}, \mathcal{M}(F)_{11}, \ldots. \tag{3.7}$$

That is, for any binary numbers i and j with strictly less than $\log n$ bits, we list $\mathcal{M}(F)_i$ before $\mathcal{M}(F)_j$ if i has fewer bits than j or if i and j have the same number of bits and $i < j$. Take the first network $\mathcal{M}(F)_i$ in list (3.7) that has strictly less than $\frac{1}{4} n_i \frac{\log \epsilon}{\log \rho_0}$ crossing comparators where $n_i = \frac{n}{2^{l(i)}}$ (where $l(i)$ is the length of i) is the total number of input items to $\mathcal{M}(F)_i$. (Such an integer i does exist since we have assumed that (3.5) is not true.)

By the way we create list (3.7) and the definition of n_i, we know that $n_i \geq 2$. In what follows, we will assume $n_i \geq 4$. The case when $n_i = 2$ is easily handled as a special case (or it can be ignored by replacing $\log n$ with $\log n - 1$ in the lower bound).

Step 2. Compute the history of $\mathcal{M}(F)_i$ on input

$$(1, +\infty, 2, +\infty, \ldots, i, +\infty, \ldots, n_i/2, +\infty),$$

and label each crossing comparator by its (unique) $\mathcal{M}(F)_{i1}$-input item (i.e., the item that is input from the subnetwork $\mathcal{M}(F)_{i1}$). Let $S(j)$ be the set of the crossing comparators labeled by j. Go through the list

$$S\left(\frac{n_i}{4} + 1\right), S\left(\frac{n_i}{4} + 2\right), \ldots, S\left(\frac{n_i}{2}\right)$$

and choose the first set $S(s)$ such that

$$\sum_{\frac{n_i}{4}+1 \leq j \leq s} |S(j)| < \left(s - \frac{n_i}{4}\right) \frac{\log \epsilon}{\log \rho_0}. \tag{3.8}$$

(Here, our assumption that $n_i \geq 4$ ensures that $\frac{n_i}{4}$ is an integer.) Such an s does exist, since, by the choice of i, $\mathcal{M}(F)_i$ contains less than $\frac{1}{4} n_i \frac{\log \epsilon}{\log \rho_0}$ crossing comparators, and therefore $s = \frac{n_i}{2}$ satisfies (3.8). (Note that our assumption $n_i \geq 4$ implies $\frac{n_i}{4} + 1 \leq \frac{n_i}{2}$.) By the minimality of s, we can conclude the following.

CLAIM 1. $|S(s)| < \frac{\log \epsilon}{\log \rho_0}$.

Step 3. We will continue to work on the history of $\mathcal{M}(F)_i$ on input

$$(1, +\infty, 2, +\infty, \ldots, i, +\infty, \ldots, n_i/2, +\infty),$$

and we will choose a characteristic set for F, $\mathit{Char}(F)$, from $S(s)$. List all the comparators in $S(s)$ as follows:

$$C_1 \prec C_2 \prec \cdots \prec C_t, \tag{3.9}$$

where $\prec$ is the depth-respecting total order described earlier. We first put comparator C_1 into $\mathit{Char}(F)$. Then we modify the behavior of C_1 (thereby making it faulty in a particular way) so as to make C_1 directly output its $\mathcal{M}(F)_{i0}$-input item to all its $\mathcal{M}(F)_{i0}$-outputs (if any), without changing any $\mathcal{M}(F)_{i1}$-output of C_1.

Before this modification, the $\mathcal{M}(F)_{i1}$-input of C_1 contained s and the $\mathcal{M}(F)_{i0}$-input of C_1 did not contain s since s could not have moved into $\mathcal{M}(F)_{i0}$ without using a comparator labeled by s. Moreover, if an $\mathcal{M}(F)_{i0}$-output of C_1 did not contain s (which is the $\mathcal{M}(F)_{i1}$-input content of C_1), it had to contain the $\mathcal{M}(F)_{i0}$-input content of C_1. Therefore, this modification has the effect of changing some output content of C_1 from s to non-s. By Lemma 2.2, this modification cannot cause any new crossing comparator to be labeled by s. Now C_1 is no longer capable of moving s from $\mathcal{M}(F)_{i1}$ to $\mathcal{M}(F)_{i0}$. Then we update the history accordingly. In the new history, $C_2, C_3, \ldots, C_t$ are the only remaining comparators that might move s from $\mathcal{M}(F)_{i1}$ to $\mathcal{M}(F)_{i0}$. In this remaining part of list (3.9), we choose the first comparator labeled by s in the new history. We put this comparator into $\mathit{Char}(F)$ and modify its behavior as we did for C_1. We then update the history again and continue in this fashion until all comparators in list (3.9) have been dealt with.

This completes the 3-step procedure and the construction of $\mathit{Char}(F)$. The final history, in which s is never moved into $\mathcal{M}(F)_{i0}$, corresponds to another fault pattern, and we will call this fault pattern $\tilde{F}$.

LEMMA 3.2. *For any fault pattern F,* (1) $\mathit{Char}(F) = \mathit{Char}(\tilde{F})$*,* (2) $\tilde{F}$ *is bad, and* (3)$|\mathit{Char}(F)| < \frac{\log \epsilon}{\log \rho_0}$.

Proof. To prove (1), we use the 3-step procedure to determine $\mathit{Char}(\tilde{F})$ and to show that it is the same as $\mathit{Char}(F)$.

We first note that $\mathcal{M}(\tilde{F})$ and $\mathcal{M}(F)$ have the same history on the input list

$$\Pi = (\underbrace{0, \ldots, 0}_{n/2}, \underbrace{1, \ldots, 1}_{n/2}).$$

This is because the changes made in comparators to produce $\tilde{F}$ from F do not affect the performance of $\mathcal{M}(F)$ on Π. In particular, the only changes from F to $\tilde{F}$ are made in Step 3 and they do not affect the history of $\mathcal{M}(F)$ on input Π, although they do affect the history of $\mathcal{M}(F)$ on input $(1, +\infty, 2, +\infty, \ldots, i, +\infty, \ldots, n_i/2, +\infty)$, as designed. (Recall that Step 3 lets an $\mathcal{M}(F)_{i0}$-output (or $\mathcal{M}(F)_{i1}$-output) remain so after the changes.) Hence, for all j such that $\mathcal{M}(F)_j$ is listed before or at $\mathcal{M}(F)_i$ in list (3.7), the structure of $\mathcal{M}(\tilde{F})_j$ is the same as the structure of $\mathcal{M}(F)_j$, although the functionality of some individual comparators may differ.[3] In addition, for all j such that $\mathcal{M}(F)_j$ is listed before or at $\mathcal{M}(F)_i$ in list (3.7), the crossing comparators for $\mathcal{M}(\tilde{F})_j$ are the same as the crossing comparators for $\mathcal{M}(F)_j$. Hence, the value of i for $\mathcal{M}(\tilde{F})$ selected in Step 1 is equal to that for $\mathcal{M}(F)$.

[3]To be precise, we say two networks have the same structure if and only if they look identical when they are drawn out in a picture like Figure 1.1(b), where we do not really care how the comparators work in presence of faults. That is, we care only about the topology of the network, not the functionalities of individual comparators.

Next, we show that we pick the same value of s for $\mathcal{M}(\tilde{F})_i$ and $\mathcal{M}(F)_i$ in Step 2. From the construction of $\tilde{F}$, we know that $\tilde{F}$ differs from F (at most) only in comparators that are labeled s in the history of $\mathcal{M}(F)_i$. More precisely, some outputs of these comparators (i.e., the $\mathcal{M}(F)_{i0}$-outputs) that contain s in the history of $\mathcal{M}(F)_i$ may contain non-s values in the history of $\mathcal{M}(\tilde{F})_i$. By Lemma 2.2, the effect of these changes is to make (possibly) some values that were s or less in the history of $\mathcal{M}(F)_i$ be smaller in the history of $\mathcal{M}(\tilde{F})_i$ and to make (possibly) some values that were s or greater in the history of $\mathcal{M}(F)_i$ be larger in the history of $\mathcal{M}(\tilde{F})_i$. Hence, the same value of s will be chosen in Step 2 for $\tilde{F}$ as for F. (The reason that we have used the cumulative threshold in Step 2 instead of simply selecting the smallest $S(s)$ should now be apparent.)

According to the description of the 3-step procedure, we can see that the starting history for $\tilde{F}$ at the beginning of the 3-step procedure is the same as the final history at the termination of Step 3 for F. (This can be shown inductively from the lowest level to the highest level.) Since the first two steps do not change any comparators, at the beginning of Step 3 for $\tilde{F}$ we have all the comparators in $\mathit{Char}(F)$ to start with. As we move along in the history of $\mathcal{M}(\tilde{F})_i$ on input $(1, +\infty, 2, +\infty, \ldots, n_i/2, +\infty)$, we will not make any real change on any comparator in $\mathit{Char}(F)$ since, at the termination of the 3-step procedure for F, all the comparators in $\mathit{Char}(F)$ have already output their $\mathcal{M}_{i0}$-inputs directly to their $\mathcal{M}_{i0}$-outputs. Furthermore, these comparators are all labeled by s in the history for $\tilde{F}$. Therefore, we have to put all the comparators in $\mathit{Char}(F)$ into the characteristic set for $\tilde{F}$. Hence, $\mathit{Char}(F) = \mathit{Char}(\tilde{F})$, as claimed.

To prove (2), we assume for the purposes of contradiction that $\tilde{F}$ is good. Then the proof technique of Lemma 2.3 implies that $\mathcal{M}(\tilde{F})_i$ functions correctly as a merging network. In particular, it should work correctly on both input

$$(\underbrace{0, \ldots, 0}_{n/2}, \underbrace{1, \ldots, 1}_{n/2})$$

and input $(1, +\infty, 2, +\infty, \ldots, n_i/2, +\infty)$. Hence, $\mathcal{M}(\tilde{F})_i$ successfully moves s from $\mathcal{M}(\tilde{F})_{i1}$ to $\mathcal{M}(\tilde{F})_{i0}$. However, in the history for $\tilde{F}$, no s can be moved from $\mathcal{M}(\tilde{F})_{i1}$ to $\mathcal{M}(\tilde{F})_{i0}$. This is a contradiction, which means that $\tilde{F}$ is bad.

The correctness of (3) follows from Claim 1. $\square$

We are now ready to describe the partition of the space of fault patterns. We group all the fault patterns by the following rule. We put F and F' in the same group if and only if (1) $\mathit{Char}(F) = \mathit{Char}(F')$ and (2) the fault patterns F and F' are identical on all the comparators not in $\mathit{Char}(F)$.

For any group G, take a fault pattern $F \in G$. By (1) in Lemma 3.2 and the construction of $\tilde{F}$, we know that $\tilde{F} \in G$. By (2) in Lemma 3.2, we know that $\tilde{F}$ is bad. By (1) and (3) in Lemma 3.2, we know that the probability that $\tilde{F}$ occurs is greater than ${\rho_0}^{\frac{\log \epsilon}{\log \rho_0}} = \epsilon$ times the probability that a fault pattern in G occurs. In other words, if a fault pattern in G occurs, there is a better than ϵ chance that it is $\tilde{F}$. Since this is true for all groups, we can thus conclude that the probability that a random fault pattern is bad is greater than ϵ.

This completes the proof of lower bound (3.5). We next show that

$$\mathit{size}(\mathcal{M}) \geq n \log n \frac{\log \sqrt{n} - \log \log_e \frac{1}{1-\epsilon}}{8 \log \frac{1}{\rho_0}}. \tag{3.10}$$

We will divide the network into blocks of size $\sqrt{n}$ and "pump up" the failure probability by showing that most of the blocks behave well in order for the overall circuit to work. In particular, we will partition the space of fault patterns into groups and use a conditional probabilistic argument.

For each fault pattern F, we can decompose $\mathcal{M}(F)$ into $\sqrt{n}$ networks such that each of them has $\sqrt{n}$ inputs and is of the form

$$\mathcal{M}(F)_i,$$

where $i \leq \sqrt{n}$ is a binary number with $\frac{\log n}{2}$ bits. By doing so, we have removed many comparators from $\mathcal{M}$. These comparators that we have removed are crossing comparators of many different networks that are larger than the networks with $\sqrt{n}$ inputs that we are currently interested in. We use *Cross*(F) to denote the set of all these removed crossing comparators. We put F and F' in the same group if and only if (1) *Cross*$(F) =$ *Cross*(F') and (2) F and F' are the same on all the comparators in *Cross*(F).

If, within each group G, the probability that a fault pattern $F \in G$ is good is less than $1-\epsilon$, then a randomly generated fault pattern will be good with probability less than $1-\epsilon$. This is a contradiction to the fact that a randomly generated F is good with probability at least $1-\epsilon$. Therefore, there exists a group G_0 such that

$$\text{Prob}[F \text{ is good} \mid F \in G_0] \geq 1-\epsilon.$$

Notice that the decomposition of $\mathcal{M}(F)$ is determined by the information about F in *Cross*(F) only. Hence, the fault patterns in the same group have the same decomposition. In particular, we can assume that, for any fault pattern $F \in G_0$ and any $i \leq \sqrt{n}$,

$$size(\mathcal{M}(F)_i) = \frac{\sqrt{n}\log\sqrt{n}x_i}{4\log\frac{1}{\rho_0}}, \tag{3.11}$$

where x_i depends on the group G_0 only (it does not depend on the individual F). By the proof technique of Lemma 2.3, we can see that each $\mathcal{M}(F)_i$ functions correctly as a merging network if F is good. Using lower bound (3.5) and (3.11), we have

$$\text{Prob}[\mathcal{M}(F)_i \text{ is good } \mid F \in G_0] \leq 1 - \frac{1}{2^{x_i}} \qquad \text{for } i \leq \sqrt{n}.$$

On the other hand, for a fault pattern F in group G_0, the behaviors of the $\mathcal{M}_i$'s ($i \leq \sqrt{n}$) are mutually independent. Hence,

$$\begin{aligned} 1-\epsilon &\leq \text{Prob}[F \text{ is good} \mid F \in G_0] \\ &\leq \text{Prob}[\mathcal{M}(F)_i \text{ is good } \forall i \mid F \in G_0] \\ &\leq \prod_{i\leq\sqrt{n}}\left(1-\frac{1}{2^{x_i}}\right) \\ &\leq e^{-\sum_{i\leq\sqrt{n}}(\frac{1}{2})^{x_i}}. \end{aligned}$$

Therefore,

$$\begin{aligned} -\log_e(1-\epsilon) &\geq \sum_{i\leq\sqrt{n}}\left(\frac{1}{2}\right)^{x_i} \\ &\geq \sqrt{n}\left(\frac{1}{2}\right)^{\frac{\sum_{i\leq\sqrt{n}}x_i}{\sqrt{n}}}, \end{aligned}$$

where the last inequality holds due to the concavity of function $(\frac{1}{2})^x$. Hence

$$\sum_i x_i \geq \sqrt{n}\left(\log\sqrt{n} - \log\log_e \frac{1}{1-\epsilon}\right).$$

Finally, we have

$$\begin{aligned} size(\mathcal{M}) &\geq \sum_{i\leq\sqrt{n}} size(\mathcal{M}(F)_i) \\ &\geq \frac{\sqrt{n}\log\sqrt{n}\sum_{i\leq\sqrt{n}} x_i}{4\log\frac{1}{\rho_0}} \\ &= \frac{n\log n(\log\sqrt{n} - \log\log_e\frac{1}{1-\epsilon})}{8\log\frac{1}{\rho_0}}. \end{aligned}$$

This proves (3.10) and completes the proof of Theorem 3.1. □

4. Trading depth for width. In this section, we show that there is a nice trade-off between the width and depth of fault-tolerant sorting networks. As in [2], we need to assume in this section that ρ is less than a constant strictly less than $\frac{1}{2}$. The proofs combine the COLUMN-SORT algorithm in [8] with the fault-tolerant sorting networks in [2].

THEOREM 4.1. *For any $n \leq w \leq n\log n$ and any ρ less than a constant strictly less than $\frac{1}{2}$, there exists an explicit construction of a $(\rho, 1/\text{poly}(n))$-destructive-fault-tolerant sorting network with width $O(w)$ and depth $O(\frac{n\log^2 n}{w})$.*

Proof. The COLUMN-SORT algorithm in [8] arranges all the items in an $r \times s$ matrix, where $r \geq 2s^2$, and it consists of 8 phases. Phases 1, 3, 5, 7 sort each column of the matrix and phases 2, 4, 6, 8 permute the matrix in a fixed manner. The only property of COLUMN-SORT that we need here is the fact that if we can sort all the columns of the matrix in T steps, then by applying COLUMN-SORT we can sort all the items in the matrix in $O(T)$ steps.

Assaf and Upfal [2] have shown how to build a $(\rho, 1/\text{poly}(n))$-destructive-fault-tolerant sorting network with width $O(n\log n)$ and depth $O(\log n)$. Hence, for a given w between n and $n\log n$, we can use a network with width w and depth $O(\log(\frac{w}{\log n})) = O(\log n)$ to sort $\frac{w}{\log n}$ items first. (At the same time, we need to keep all other items in some other registers. This can be done as long as we keep enough, say, $2n$, registers.) Then keep this sorted list of $\frac{w}{\log n}$ items in some registers and work on the next group of $\frac{w}{\log n}$ items, etc. We will have worked on all the groups after $O(\frac{n}{w/\log n}) = O(\frac{n\log n}{w})$ rounds. This finishes the first phase of COLUMN-SORT with depth $O(\frac{n\log^2 n}{w})$. To implement the second phase of COLUMN-SORT, we can hardwire in a permutation. We similarly finish the remaining phases. The overall depth is $O(\frac{n\log^2 n}{w})$. □

THEOREM 4.2. *For any $n \leq w \leq nk$ and $k = O(\log n)$, there exists an explicit construction of a k-destructive-fault-tolerant sorting network with width $O(w)$ and depth $O(\frac{nk\log n}{w})$.*

Proof sketch. In [2], Assaf and Upfal did not address the issue of worst-case faults. However, by following their method it is possible to construct a k-destructive-fault-tolerant sorting network with width $O(kn)$ and depth $O(\log n)$ when $k = O(\log n)$. Therefore we can use COLUMN-SORT, as we did in the proof of Theorem 4.1, to prove the theorem. □

5. Concluding remarks. We have shown lower bounds for merging and sorting networks that can tolerate destructive faults. Our lower bound for random faults is tight in the most interesting case where ρ is a constant and $1/\epsilon$ is a nonzero constant or a polynomial in n. Our lower bound for worst-case faults is tight when $k = O(\log n)$. In joint work with Kleitman [5], we have shown that there are better lower bounds for worst-case faults when k is very large, say, exponential in n.

Acknowledgment. We thank the anonymous referee for an outstanding job.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *An $O(n \log n)$ sorting network*, in Proc. 15th ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 1–9.

[2] S. Assaf and E. Upfal, *Fault-tolerant sorting network*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 275–284.

[3] K. E. Batcher, *Sorting networks and their applications*, in Proc. AFIPS Spring Joint Computer Conference, 32, Atlantic City, NJ, 1968, pp. 307–314.

[4] P. Donejko, K. Diks, A. Pelc, and M. Piotrów, *Reliable minimum finding comparator networks*, in Proc. 19th Symposium on Mathematical Foundations of Computer Science, I. Prívara, B. Rovan, and P. Ružička, eds., Košice, Slovakia, 1994, Lecture Notes in Comput. Sci. 841, Springer-Verlag, Berlin, 1994, pp. 306–315.

[5] D. Kleitman, T. Leighton, and Y. Ma, *On the design of Boolean circuits that contain partially unreliable gates*, in Proc. 35th IEEE Symposium on Foundations of Computer Science, Santa Fe, NM, 1994, pp. 332–346.

[6] D. E. Knuth, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, Addison–Wesley, Reading, MA, 1973.

[7] S.-Y. Kuo and S.-C. Liang, *Defect-tolerant hierarchical sorting networks for wafer-scale integration*, IEEE J. Solid-State Circuits, 26 (1991), pp. 1212–1222.

[8] T. Leighton, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput., C-34 (1985), pp. 326–335.

[9] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan-Kaufmann, San Mateo, CA, 1992.

[10] T. Leighton, Y. Ma, and C. G. Plaxton, *Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults*, J. Comput. System Sci., 54 (1997), pp. 265–304.

[11] T. Leighton, Y. Ma, and C. G. Plaxton, *Highly fault-tolerant sorting circuits*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1991, pp. 458–469.

[12] Y. Ma, *Fault-Tolerant Sorting Networks*, Ph.D. thesis, MIT, Cambridge, MA, 1994.

[13] Y. Ma, *An $O(n \log n)$-size fault-tolerant sorting network*, in Proc. 28th ACM Symposium on the Theory of Computing, Philadelphia, PA, 1996, pp. 266–275.

[14] M. Piotrów, *Depth optimal sorting networks resistant to k passive faults*, in Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, Atlanta, GA, 1996, pp. 242–251.

[15] N. Pippenger, *On networks of noisy gates*, in Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 30–36.

[16] L. Rudolph, *A robust sorting network*, IEEE Trans. Comput., 34 (1985), pp. 344–354.

[17] M. R. Samatham and D. K. Pradhan, *The De Bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI*, IEEE Trans. Comput., 38 (1989), pp. 567–581.

[18] M. Schimmler and C. Starke, *A correction network for N-sorters*, SIAM J. Comput., 18 (1989), pp. 1179–1187.

[19] J. Sun, E. Cerny, and J. Gecsei, *Design of a fault-tolerant sorting network*, in Proc. Canadian Conference on Very Large Scale Integration, Ottawa, Canada, 1989, pp. 235–242.

[20] J. Sun, E. Cerny, and J. Gecsei, *Sorting networks without critical stages*, in Proc. Canadian Conference on Very Large Scale Integration, Montreal, Canada, 1990, pp. 5.4/1–8.

[21] J. Sun, E. Cerny, and J. Gecsei, *Fault tolerance in a class of sorting networks*, IEEE Trans. Comput., 43 (1994), pp. 827–837.

[22] J. Sun and J. Gecsei, *A multiple-fault tolerant sorting network*, in Proc. 21st International Symposium on Fault-Tolerant Computing, Montreal, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 274–281.

[23] J. Sun, J. Gecsei, and E. Cerny, *An improvement in robust sorting networks*, in Proc. 23rd ASILOMAR Conference on Signals, Systems and Computers, Pacific Grove, CA, 1989, pp. 1036–1040.

[24] J. Sun, J. Gecsei, and E. Cerny, *Fault-tolerance in balanced sorting networks*, J. Electronic Testing Theory Appl., 1 (1990), pp. 31–41.

[25] A. C. Yao and F. F. Yao, *On fault-tolerant networks for sorting*, SIAM J. Comput., 14 (1985), pp. 120–128.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 274–287

ROUTING WITH MINIMUM WIRE LENGTH IN THE DOGLEG-FREE MANHATTAN MODEL IS $\mathcal{NP}$-COMPLETE[*]

TIBOR SZKALICZKI[†]

Abstract. The present article concentrates on the dogleg-free Manhattan model where horizontal and vertical wire segments are positioned on different sides of the board and each net (wire) has at most one horizontal segment. While the minimum width can be found in linear time in the single row routing, apparently there was no efficient algorithm to find the minimum wire length. We show that there is no hope to find such an algorithm because this problem is $\mathcal{NP}$-complete even if each net has only two terminals. The results on dogleg-free Manhattan routing can be connected with other application areas related to interval graphs. In this paper we define the minimum value interval placement problem. There is given a set of weighted intervals and w rows and the intervals have to be placed without overlapping into rows so that the sum of the interval values, which is the value of a function of the weight and the row number assigned to the interval, is minimum. We show that this problem is $\mathcal{NP}$-complete. This implies the $\mathcal{NP}$-completeness of other problems including the minimum wire length routing and the sum coloring on interval graphs.

Key words. single row routing, VLSI, $\mathcal{NP}$-complete problems, minimum wire length, interval graph

AMS subject classifications. 68Q25,68Q35

PII. S0097539796303123

1. Introduction.

The *routing problem* is to decide whether the problem instance is routable under specified restrictions and, if yes, to determine the routes of the wires that optimize certain criteria. The points to be interconnected are called *terminals*. Routing within a rectangle is a basic problem of the VLSI design. In case of *single row routing* all terminals appear only on one side of the rectangle. This is a special case of the *channel routing* where all terminals are located either at the upper or the lower boundary of the routing region. A *net* is a collection of terminals. An instance of the problem is a set of pairwise disjoint nets. The solution of a routing problem is a set of subgraphs (wires) where each subgraph connects all the terminals of the corresponding net under the conditions of the wiring model. In the *Manhattan model* wires run on a rectangular grid and horizontal and vertical wire segments are positioned on different sides of the board. In a restricted version of the Manhattan model each wire could occupy only one horizontal row (track). This model is called the *dogleg-free* model. We are interested in the complexity of finding the minimum wire length solution of the given routing problem in the dogleg-free model. Lengauer [5] presents a detailed exposition of the routing in the Manhattan model.

The minimum width can be found in linear time in the single row routing in the Manhattan model (Gallai [1]; see also Recski [7]). Szkaliczki [8] found an algorithm for the minimum wire length whose running time is linear in the length and super-polynomial in the width of the channel even in case of the channel routing. LaPaugh [4] proved that the channel routing problem is $\mathcal{NP}$-complete. We shall prove that

[*]Received by the editors May 6, 1996; accepted for publication (in revised form) September 13, 1998; published electronically September 22, 1999. This work was supported by Hungarian National Foundation for Scientific Research grant OTKA T 017181 and by Hungarian Ministry of Education grant FKFP 409/1997.

http://www.siam.org/journals/sicomp/29-1/30312.html

[†]Computer and Automation Research Institute, H-1111 Budapest, Lágymányosi u. 11., Hungary (sztibor@ilab.sztaki.hu).

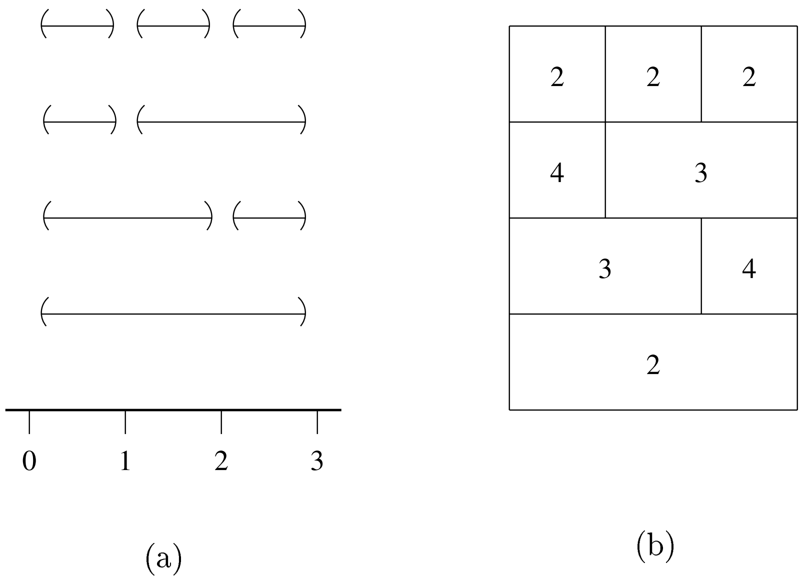


FIG. 1. *An instance of the interval placement problem:* (a) *interval representation,* (b) *rectangle representation.*

the minimum wire length single row routing in the dogleg-free Manhattan model is $\mathcal{NP}$-complete so there is no hope to find an algorithm which is polynomial both in length and width. This holds even if each net has only two terminals. Our result is that the minimum wire length routing is computationally difficult even in one of the simplest cases.

In this paper we define the minimum value interval placement problem. We prove that it is $\mathcal{NP}$-complete. We use this result to prove that finding the minimum wire length is $\mathcal{NP}$-complete in the single row routing in the dogleg-free Manhattan model. The minimum value interval placement can be applied to other areas such as graph coloring.

2. Interval placement problem.

The *interval placement problem* is as follows. Assume a finite set of intervals on a line and w rows. Each interval has to be placed into one of the rows in such a way that two intervals can be placed into the same row if and only if they have no common point. Several intervals can have the same end point. Figure 1(a) depicts an instance of the interval placement problem ($w = 4$). This figure not only gives the specification of the problem instance but also shows one of the solutions.

We often need an optimum solution. The minimum width, that is, the minimum number of necessary rows, can be found in linear time (Gallai [1]; see also Recski [7]). We define the value of a solution of an interval placement problem in the following way. A weight is assigned to each interval. The interval j has the weight l_j and in a solution it is placed into row r_j. The value of an interval is $r_j \cdot l_j$. The value (v) of a solution of the interval placement problem is the sum of the values of all intervals:

$$v = \sum r_j \cdot l_j. \tag{1}$$

If l'_i is the sum of the weights of the intervals in the ith row, then $v = \sum_{i=1}^{w} i \cdot l'_i$.

The *minimum value interval placement problem* is as follows: Is there a solution for the interval placement problem for which the value is at most k?

For simplicity, we assume that each interval is open, that is, two intervals placed into the same row can have common end points.

Figure 1(b) depicts the weighted version of the interval placement problem shown in Figure 1(a) using another notation. A rectangle corresponds to an interval. There are w rows and they have width in contrast with the notation in Figure 1(a). The height of each rectangle is equal to the width of the rows and their length is equal to the length of the corresponding interval. The rectangle is placed into the row assigned to the interval. The weights are denoted by the numbers written in the rectangles. We will use this notation because it is clear and it is suitable for the description of the solution as well as the specification of the interval placement problem.

We will deal with the saturated interval placement problem. We call an interval placement problem *saturated* if each point except the boundaries of intervals is inside either none of the intervals or exactly w intervals. Thus the intervals have to be placed continuously, without an empty place in each row. Therefore two intervals can be placed into the same row if and only if at the end of one of them is the starting point of the other or the section between them can be filled up with other intervals without an empty place.

3. Construction. We will reduce an $\mathcal{NP}$-complete problem to the minimum value interval placement problem in order to prove that this problem is $\mathcal{NP}$-complete as well. We shall show a transformation that translates an instance of a satisfiability problem of Boolean formulas (SAT; see Garey and Johnson [2]) into a saturated instance of the interval placement problem which has a solution with value k if and only if the original instance is satisfiable. The instance of SAT consists of n variables and m clauses. Without loss of generality, we assume that no clause contains the same variable more than once.

In this section, we consider each of the construction elements that correspond to the constituents of Boolean formulas (occurrence of a Boolean variable in a clause, Boolean variable, and clause). Using these elements we make a construction corresponding to the whole Boolean formula. We determine the proper weights of some additional intervals and the threshold value k for the instance of the minimum value interval placement problem. At last, we prove the $\mathcal{NP}$-completeness of the problem.

3.1. Occurrence of a variable. Figure 2 shows the part of the construction corresponding to an occurrence of a variable. Let us call it the *variable-occurrence element.* There are four different ways to place the intervals in four rows, as shown in Figure 2. If the rows in each of these figures are permuted, we do not consider the new solution to be essentially different. Obviously, in the optimal realization the sums of the weights of intervals in the same row are in decreasing order. Figure 2 depicts the four different realizations. The sum of the weights of the intervals in the rows and the value of the realization can be seen beside the realization. A variable-occurrence element has two different realizations with the minimum value. Let the realization Figure 2(a) (A) correspond to true and Figure 2(b) ($\overline{A}$) correspond to false.

3.2. Variable. Four adjacent rows correspond to a variable. Let us call them the *variable element*. They contain as many variable-occurrence elements as the number of occurrences of the variable in clauses. The variable in Figure 3 occurs in two clauses. We will determine the weights of the first intervals of the rows in section 3.5. The intervals with weight 2 located between two adjacent variable-occurrence elements

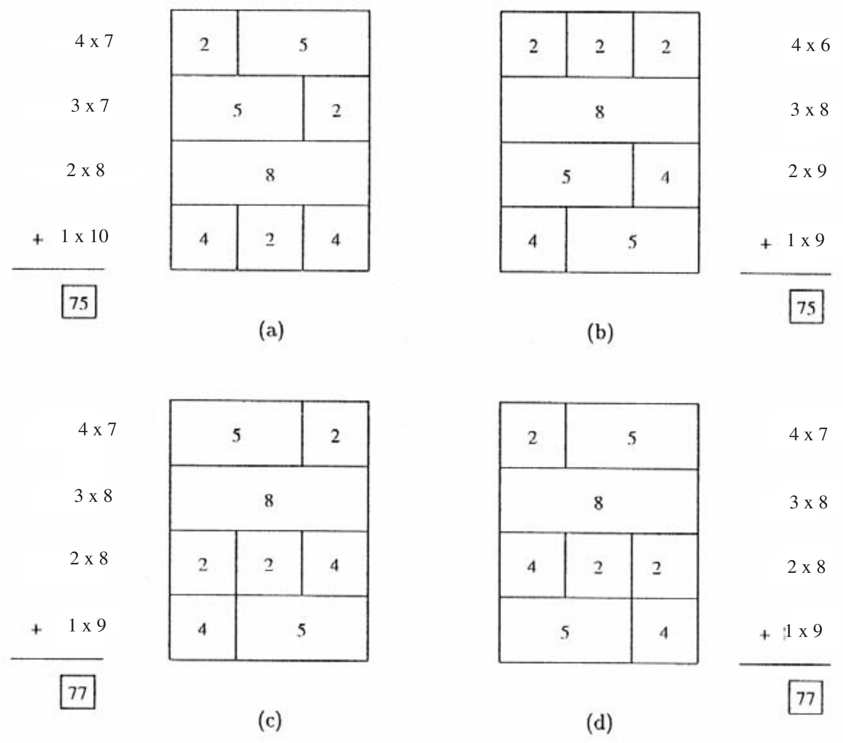


FIG. 2. *Element corresponding to an occurrence of a variable.*

and at the end of the rows are called *variable-connecting intervals*. An interval with weight 2 in the variable-occurrence element is lengthened because its exchange with an interval belonging to a clause will be permitted (see section 3.3 below), but further exchanges should be prevented. All the intervals with weight 8 belonging to the same variable and the variable-connecting intervals between them are merged into one long interval.

LEMMA 1. *The value of the variable element is minimum if and only if either realization A or $\overline{A}$ occurs at each element corresponding to an occurrence of the same variable.*

Proof. The realizations A and $\overline{A}$ of a variable-occurrence element are its minimum value realizations. The merged interval forces each element belonging to the same variable to have the same minimum value realization. □

3.3. Clause. The element corresponding to a clause is called the *clause element*. A specific example of a clause element is shown in Figure 4. The lower two rows are the *clause rows* and the rows above them are the *variable rows*. The intervals with weight t_1 and t_2 are placed between clause elements. They are called *clause-connecting intervals*. Notice that the two clause rows cannot be exchanged on a section within

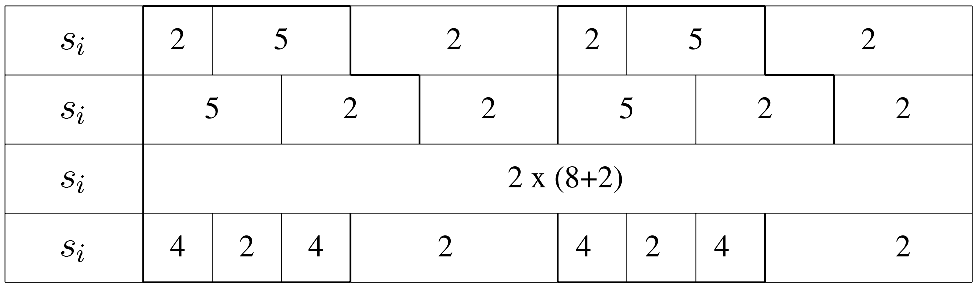


FIG. 3. *An element corresponding to a variable.*

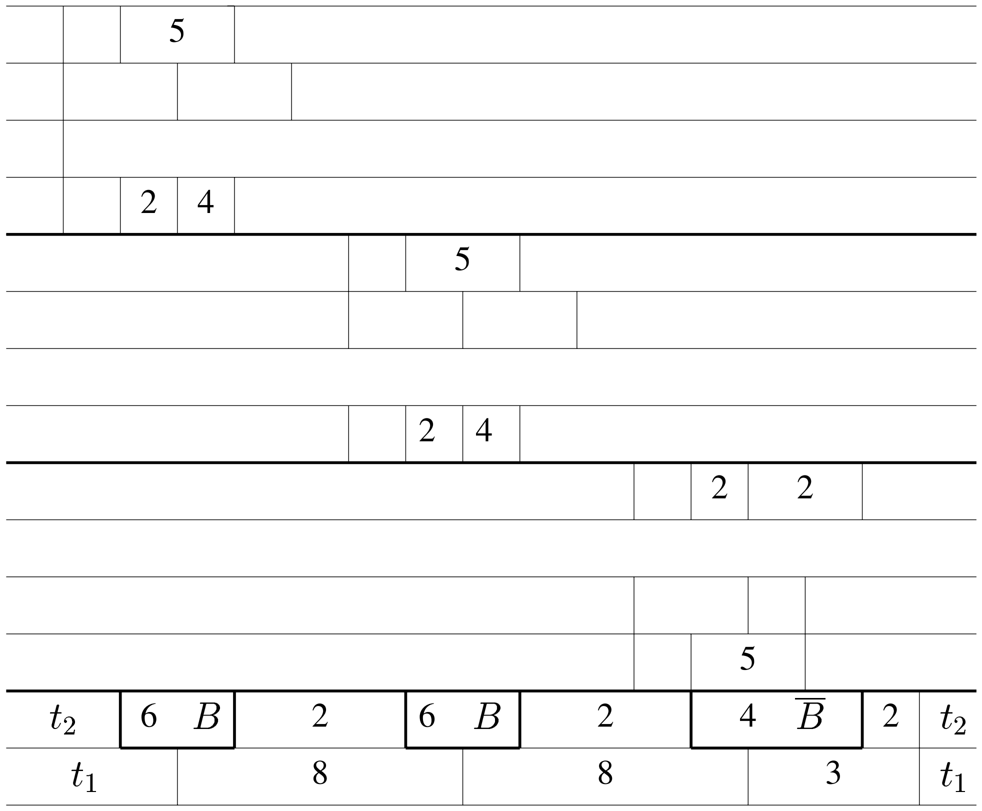


FIG. 4. *An element corresponding to the clause* $x_3 \vee x_2 \vee \overline{x}_1$.

a clause element because the intervals in different rows have no common end point. There are two types of intervals in clause rows that can be exchanged with the intervals in variable rows: B corresponds to a nonnegated variable and $\overline{B}$ to a negated variable. These intervals are marked with a thick border in the figure. The weight of B is 6 and its length is 2. The weight of $\overline{B}$ is 4 and its length is 3. There are as many variable-occurrence elements on the section of a clause element as there are variables included in the clause: each interval B and $\overline{B}$ is inside the section of exactly one variable-occurrence element corresponding to the variable included in the clause.

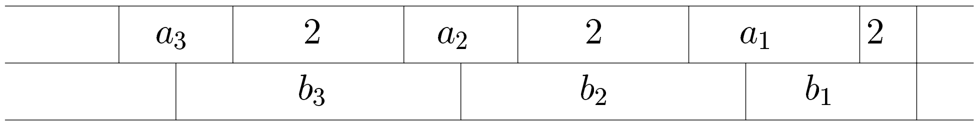


FIG. 5. *Weights of the intervals of a clause element.*

LEMMA 2. *If the clause-connecting intervals are in the lowest two rows, then among the intervals belonging to clauses only the intervals B and $\overline{B}$ can be placed into a higher row.*

Proof. Each interval belonging to a clause except B and $\overline{B}$ has a section where a variable-connecting interval is situated in each variable row. If one of these intervals is placed into a variable row, then a variable-connecting interval has to be placed into one of the clause rows. Each variable-connecting interval overlaps clause-connecting intervals. Thus at least one of the clause-connecting intervals has to be placed into a variable row, a contradiction. ▯

Figure 5 shows the clause element in the general case. a_i and b_i denote weights whose values depend on the concrete clause. We know that a_i (the weight of B or $\overline{B}$) is 4 or 6. Let

$$b_i = \begin{cases} a_i + 2 & \text{if } i > 1, \\ a_i - 1 & \text{if } i = 1. \end{cases}$$

There are two kinds of clause-connecting intervals. Let the longer interval with weight t_1 be placed into the lowest row and the shorter one with weight t_2 be placed into the second row from below. The clause rows can be exchanged with one another within a section containing clause elements together with clause-connecting intervals. The sum of weights of the intervals of a clause element in the first clause row is three plus that in the second row. Thus if $t_1 = t_2 + 4$, then the sum of weights of the intervals in the first row is greater than one in the second row, so it is not worth changing these rows in the minimum value solution. This is true even if some intervals of the clause rows are exchanged with intervals of the variable rows. The value of $t_2 = t$ will be determined in section 3.6.

Depending on the realization of the variable-occurrence element overlapping the intervals B and $\overline{B}$, we may say that realization A or $\overline{A}$ belongs to the intervals B and $\overline{B}$.

LEMMA 3. *Let us assume that the clause-connecting intervals are placed into the lowest two rows and the variable-occurrence elements together with the variable-connecting intervals belonging to the same variable are placed into adjacent rows. Then the placement of intervals belonging to them has the minimum value if and only if realization A belongs to interval B or realization $\overline{A}$ belongs to $\overline{B}$ at least at one occurrence of a variable at each clause.*

Proof. Let us suppose that each interval belonging to the clause is in the lowest two rows and each interval belonging to the occurrences of variables is in the corresponding variable rows, as Figure 4 shows. Now we examine how the value of this placement can be reduced. By Lemma 2, only intervals B or $\overline{B}$ can be placed into variable rows from clause rows.

Let us consider the minimum value placement if an interval B or $\overline{B}$ is placed into a variable row. If A is the realization of the variable, then the intervals with weight 2 and 4 can be exchanged with B in the clause row; however, there is no exchange

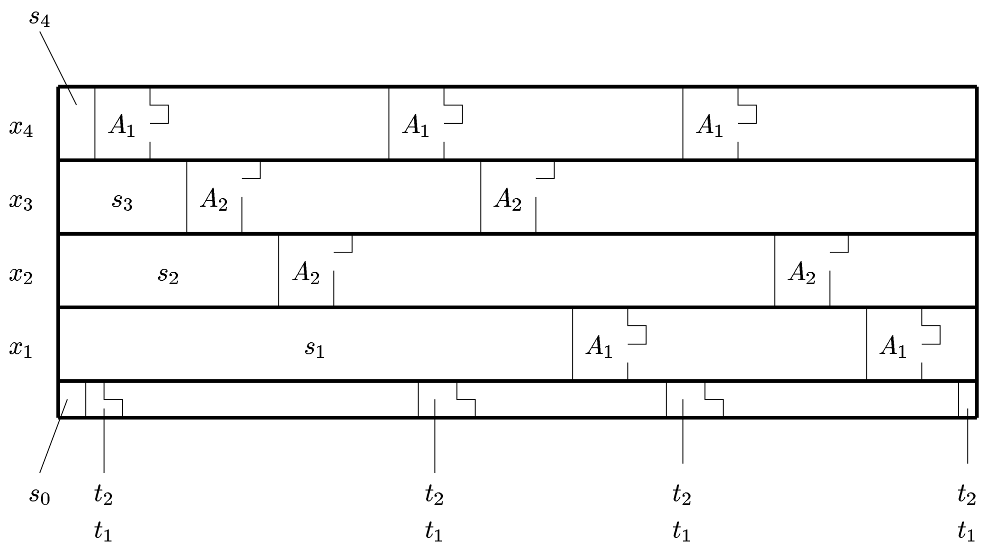


FIG. 6. *Outline of the construction corresponding to the formula* $(x_4 \vee x_3 \vee x_2) \wedge (x_4 \vee \overline{x}_3 \vee x_1) \wedge (\overline{x}_4 \vee \overline{x}_2 \vee \overline{x}_1)$.

with $\overline{B}$. If $\overline{A}$ is the realization of the variable, then two intervals with weight 2 can be exchanged with $\overline{B}$. The interval with weight 5 can be exchanged with B, but this is not worth doing because the total value increases and this exchange does not enable an additional exchange. The exchange of B or $\overline{B}$ with the appropriate intervals belonging to variables does not modify the total value of the placement but it enables the exchange of the two clause rows on a section, which reduces the total value by one. Let us call this exchange of clause rows an *improving exchange*. It does not matter whether one or more intervals belonging to the same clause are exchanged with intervals of variable rows because the total value can be reduced by one in each case.

It can easily be proved that this is the only way to reduce the value. For this reason, the value of a clause element cannot be reduced by more than one. Thus the value of a clause element can be reduced by one if and only if realization A belongs to interval B or realization $\overline{A}$ belongs to $\overline{B}$ at one or more occurrences of variables. The total value is minimum if this holds at each clause. □

3.4. Boolean formula. Now we know all the necessary elements to construct an instance of the interval placement problem that can be realized with a certain value if and only if the original Boolean formula is satisfiable. The structure of the whole construction is essentially the repetition of the block shown in Figure 4. Figure 6 depicts a construction in outline.

LEMMA 4. *If the clause-connecting intervals are in the lowest two rows and if the first intervals belonging to variables are in the corresponding variable rows, then none of the intervals belonging to variables are in rows belonging to other variables.*

Proof. The proof is indirect. Assume that some of the intervals belonging to variables are placed into rows belonging to other variables. A variable-connecting interval has to be moved in any case because each interval of a variable-occurrence element overlaps a variable-connecting interval or a fixed first interval in each row belonging to other variables. Let i denote the variable-connecting interval which is placed into a row belonging to another variable v, and its first point is the leftmost among the points of such variable-connecting intervals. Interval i cannot be placed into a clause row because each variable-connecting interval overlaps clause-connecting intervals fixed in the lowest two rows. There is an interval j which overlaps the beginning of interval i, belongs to variable v, and is placed into a row belonging to another variable. Notice that the beginning of each variable-connecting interval is overlapped only by variable-connecting intervals or by fixed first intervals in the rows belonging to other variables. If j is a variable-connecting interval, then this contradicts the assumption that i is the leftmost variable-connecting interval which is placed into a row belonging to another variable. If j is a first interval, then this contradicts the fact that the first intervals are fixed. $\square$

3.5. The weight of the first interval of a row. Let s_i be the weight of the first intervals belonging to the ith variable $(1 \le i \le n)$ and let s_0 belong to the clause rows. Let the weights of the first intervals belonging to the same variable be the same. Let l_{ia} $(1 \le i \le n, a = 1,2,3,4)$ be the sum of the weights of the intervals in the ath row of the ith variable and l_{0a} $(a = 1,2)$ be the sum of the weights of the intervals in the ath clause row. Let $l'_{ia} = l_{ia} - s_i$. Let l' be greater than any l'_{ia}, $i \ge 1$.

LEMMA 5. *If $s_i = (n+1-i)\cdot 12m$ $(0 \le i \le n)$ and the clause-connecting intervals are in the lowest two rows, then the first intervals belonging to the clauses are in the lowest two rows and the first intervals belonging to the ith variable are in the rows $4k-1$, $4k$, $4k+1$, $4k+2$ in the minimum value solution. (m is the number of the clauses and n is the number of the variables in the Boolean formula.)*

Proof. The sum of the weights of the intervals of a variable-occurrence element in one row and the weight of a variable-connecting interval is at most 12. By Lemma 2, if the clause-connecting intervals are in the lowest two rows, then only intervals B and $\overline{B}$ can be placed from the clause rows into the variable rows and these intervals do not increase the maximum sum of weights of a variable row. Thus the value $12m$ is appropriate for l'.

Let now $s_i = (n+1-i)\cdot 12m$. In this case if $j < i$, then $s_j > l' + s_i$ and so $l_{ja} > l_{ib}$ for each a and b. Thus the first interval with weight s_i is assigned to a higher row than that with weight s_j in the minimum value solution. Therefore the first intervals are placed into rows in decreasing order of their weights. $\square$

3.6. The weights of the clause-connecting intervals. Let the total value of the first intervals in all rows be v_1, the total value of the variable-occurrence elements be v_2, the total value of the variable-connecting intervals be v_3, the total value of the clause elements be v_4, and the total value of the clause-connecting intervals be v_5.

v'_i $(i = 1,2,3,4,5)$ denotes the corresponding values belonging to minimum value placement assuming that each clause-connecting interval is placed into the lowest two rows; v''_i $(i = 1,2,3,4,5)$ denotes the same values assuming that at least one of the clause-connecting intervals is not placed into the lowest two rows.

LEMMA 6. *If the weights $(t_1 = t+4, t_2 = t)$ of the clause-connecting intervals are at least $v'_2 + v'_3 + v'_4$, then there is a minimum value solution where these intervals are placed into the lowest two rows.*

Proof. If an interval is placed higher by one or more rows, then the value is increasing by at least the weight of the interval. Thus $v''_5 \ge v'_5 + t$. By Lemma 5, if the clause-connecting intervals are in the lowest two rows, then the first intervals are placed into rows in decreasing order of their weights. The value of their placement cannot be reduced; thus $v'_1 \leq v''_1$. Since $v''_i > 0$,

$$v'' = v''_1 + v''_2 + v''_3 + v''_4 + v''_5 \geq v''_1 + v''_5 \geq v'_1 + v'_5 + t \geq v'_1 + v'_5 + v'_2 + v'_3 + v'_4 = v'.$$

Thus if the clause-connecting intervals are placed into the lowest two rows, then the minimum value of the placement is not greater than the value of any placement containing at least one clause-connecting interval in the third or higher rows. ∎

3.7. Minimum value and $\mathcal{NP}$-completeness. We have shown a construction to assign a weighted interval set to each instance of SAT. We have to calculate an appropriate value for k which is contained by the instance of the minimum value interval placement problem as well and it is equal to the minimum value which can be achieved if and only if an improving exchange can be realized at each clause. Details are omitted, but it is necessary to note that k can be determined in polynomial time, because each weight and the value of each component of the optimum solution can be calculated in polynomial time. The equation for k is quite complicated:

$$k = 36(n+1)m + 4mn(n+1)(8n+19) + \sum_{i=1}^{n} j_i \cdot 95 \tag{2}$$

$$+ \sum_{i=1}^{n} j_i(4i-2)\cdot 40 + 24\sum_{i=1}^{m} p_i + 18\sum_{i=1}^{m} q_i - 4m + (m+1)(600mn^2+4).$$

Theorem 1. *The saturated minimum value interval placement problem is $\mathcal{NP}$-complete.*

Proof. The instance of SAT consists of n variables and m clauses, and o is the total number of occurrences of the variables. The dimensions of the instance of the interval placement problem are the following:

The number of rows: $4n+2$.

The number of variable-occurrence elements: o.

The number of clause elements: m.

The number of intervals: $12o + 2m + 5n + 4$.

The number of different interval boundaries: $5o + m + 2$.

The maximum weight: $200mn^2 + 4$.

$k < c(n^3m + n^2m^2)$, where c is an appropriate constant.

Each element of the construction can be determined in polynomial time in the size of the Boolean formula, and the whole construction applies a polynomial number of building elements. We can conclude that each instance of SAT can be translated into an instance of the minimum value interval placement problem in polynomial time.

We shall prove that the interval placement problem has a solution with value k if and only if the Boolean formula is satisfiable. By Lemma 6, the clause-connecting intervals are placed into the lowest two rows in the minimum value solution. By Lemma 5, the first intervals belonging to the variables are placed into fixed adjacent rows in the minimum value solution. Thus Lemmas 2 and 4 guarantee that each interval belonging to a variable except the intervals interchanged with B or $\overline{B}$ is placed into the corresponding variable row. By Lemma 1, the same realization belongs to each element corresponding to the occurrence of the same variable in the minimum value solution. By Lemma 3, the solution of the instance of the interval placement problem has the least value if and only if realization A belongs to interval B or realization $\overline{A}$ belongs to interval $\overline{B}$ at least at one occurrence of a variable in each clause element.

Suppose first that the placement of intervals can be realized with value k. If the variable element has a realization A or $\overline{A}$, let the corresponding Boolean variable be true or false, respectively. In this case each clause and the entire Boolean formula is satisfiable. Conversely, if the Boolean formula is satisfiable, apply realization A or $\overline{A}$ to the variable element depending on the value of the Boolean variable. In this case we can make an improving exchange in each clause element and we can achieve the placement value k. $\square$

4. Extensions and further results.

4.1. Unweighted case. The problem remains $\mathcal{NP}$-complete in the unweighted case, when the weight of each interval is 1 so the value of the interval j placed into the row r_j is simply r_j and the value of the solution is $v = \sum_j r_j$.

The saturated weighted case can be reduced to the unweighted case in the following way. Each interval with weight l_j has to be divided by $l_j - 1$ new points into l_j intervals in such a way that none of the new points coincides with an interval end point in the original problem instance and none of the new points coincides with another. In this case, all the intervals derived from the same weighted interval must be placed into the same row. Using this it can be shown that the unweighted interval placement problem is $\mathcal{NP}$-complete, too.

We still have to show that the transformation can be done in polynomial time. The only critical point is the number of divisions to be done. This is less than the product of the number of intervals in the original instance and the maximum weight. Unfortunately, the value of a weight is exponential in its length. However, the minimum value interval placement problem remains $\mathcal{NP}$-complete if all weights can be bounded from above by a polynomial in the number of intervals (see the expressions for the maximum weight and for the number of intervals in the proof of Theorem 1). For this, the number of divisions to be done can be bounded from above by a polynomial in the input length.

4.2. Maximization. Obviously, if we exchange the order of the rows of the minimum value solution, then we get the maximum value solution. Hence, this latter problem is $\mathcal{NP}$-complete, too.

4.3. Arbitrary width. We have assumed that the number w of the rows is given. We distinguish two new variants of the minimum value interval placement problem.

Let $\Pi(S, x)$ be the problem of deciding whether the weighted intervals in set S can be placed without overlapping with the total value at most x and with the minimum width.

Let $\Pi_0(S, x)$ be the problem of deciding whether the weighted intervals in set S can be placed without overlapping with the total value at most x and with arbitrary width.

Since the width in the construction in the proof of Theorem 1 is equal to the minimum width, $\Pi(S, x)$ is $\mathcal{NP}$-complete. $\Pi_0(S, x)$ is obviously in the class $\mathcal{NP}$. We shall prove that $\Pi_0(S, x)$ is $\mathcal{NP}$-complete, too. For this, we reduce $\Pi(S, x)$ to $\Pi_0(S, x)$ in polynomial time. We show that for each instant S, x there exists an instance S', x' so that $\Pi(S, x)$ is true if and only if $\Pi_0(S', x')$ is true.

This is not trivial because the minimum value placement may use more rows than the necessary minimum (see Figure 7). The minimum value is at most the value belonging to the situation when each interval would be placed into the highest row. Thus the total value v of a placement can be bounded from above by $k = y \cdot w + 1$,

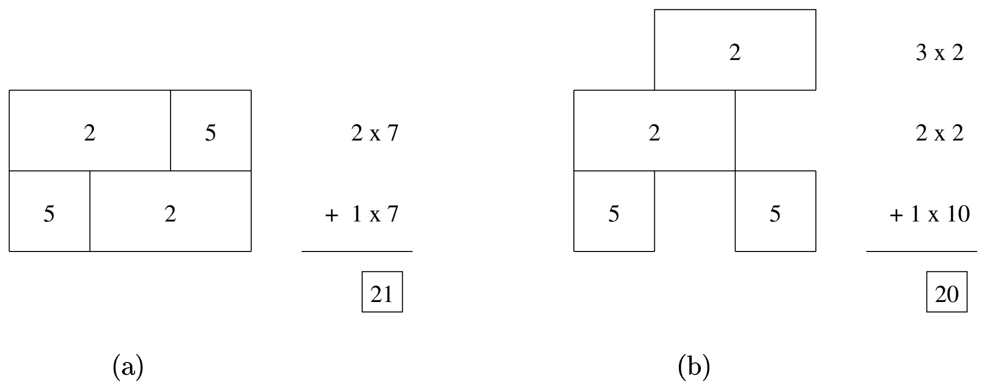


FIG. 7. *The minimum value and the minimum width placement problems are different.* (a) *A minimum value placement.* (b) *A minimum width placement.*

where y is the sum of the weights and w is the number of the rows (width). Therefore $v < k$. We get S' from S by adding k intervals with weight 1 which cover all the intervals in S. Let v' be $v + k \cdot (2w + 1 + k)/2$.

LEMMA 7. *The minimum value interval placement problem Π_0 with arbitrary width is $\mathcal{NP}$-complete.*

Proof. We have shown how to translate an instance of Π into an instance of Π_0. Now we show that the intervals in S' can be placed with the total value at most v' in any number of rows if and only if the intervals in S can be placed with the total value at most v and with minimum width w.

(if) The intervals from S are placed into the first w rows with the total value at most v. The new intervals are placed into the rows $w + 1$ through $w + k$. The total value of the new intervals is $k \cdot (2w + 1 + k)/2$.

(only if) Let us suppose that the intervals in S' can be placed with the total value at most v'. Let us note that the new intervals with weight 1 are placed into separate rows and these rows are the weakest ones; thus they are the highest rows. Let us suppose that the other intervals occupy at least $w + 1$ rows. Then the total value of the additional intervals with weight 1 is at least $k\cdot(2w+3+k)/2 = k+k\cdot(2w+1+k)/2 > v + k \cdot (2w + 1 + k)/2 = v'$. Consequently, the intervals from S use only the first w rows if the value is at most v' and the new intervals use the rows $w+1$ through $w+k$. In this case the total value of the new intervals is $k \cdot (2w + 1 + k)/2$. Since the total value of the intervals in S' is at most $v' = v + k \cdot (2w + 1 + k)/2$, the total value of the original intervals from S is at most v.

We have to check that this reduction can be done in polynomial time. For this, we show that k can be bounded from above by a polynomial in the length of the instance of Π. This length is proportional to the number i of the intervals. Clearly, $w \leq i$. Let us notice that our $\mathcal{NP}$-completeness result for problem Π holds even if all weights can be bounded from above by a polynomial in the number of the intervals. In this case, the sum y of all weights and k, which is calculated from the product of y and w, are less than a polynomial in the length of the problem instance. ∎

4.4. Graph coloring. The interval placement problem can be associated with interval graphs in the following way. The vertices of the interval graph correspond to the intervals and there is an edge between two vertices exactly if the corresponding intervals overlap. The weights of the intervals are assigned to the vertices.

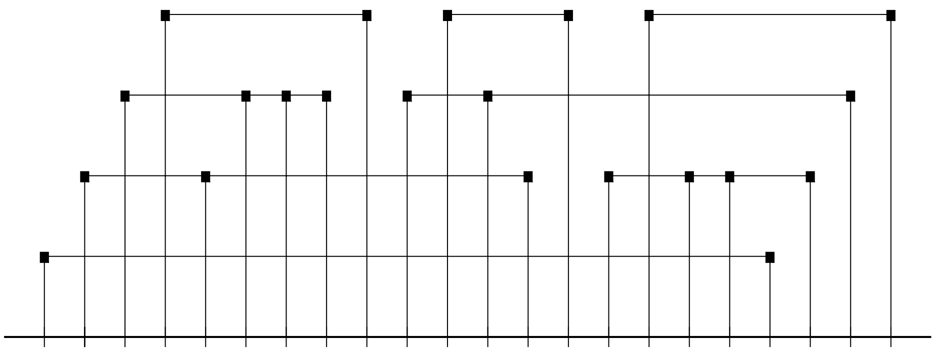


FIG. 8. *Single row routing corresponding to the interval placement in Figure* 1.

The coloring of the graph corresponds to the placement of the intervals if the colors correspond to the rows. The colors are numbered from 1 to w. The value of a vertex is the product of the color number and the weight of the vertex. We define the *minimum value graph coloring problem* to be the minimization of the sum of the values of each vertex. The minimum value coloring of an interval graph is tantamount to the minimum value interval placement problem, so it is also $\mathcal{NP}$-complete while the original coloring problem of an interval graph is polynomial.

We can apply our result to the chromatic sum problem. The chromatic sum of a graph is the minimum sum of the color numbers. The problem of determining the chromatic sum is known as the *chromatic sum problem* and the problem of producing a coloring where the sum of color numbers is equal to the chromatic sum is known as the *sum coloring problem.* These problems on arbitrary graphs are $\mathcal{NP}$-complete [3]. There is an approximation result on interval graphs [6]. Now, we can determine the complexity of these problems on interval graphs. The chromatic sum problem can be regarded as an unweighted version of the minimum value graph coloring problem. Similarly to the weighted case, the $\mathcal{NP}$-completeness of the chromatic sum problem and the sum coloring problem on interval graphs follows from the $\mathcal{NP}$-completeness of the unweighted version of the minimum value interval placement problem. If the number of colors is not fixed in the chromatic sum problem, first we reduce the unweighted version to the minimum value interval placement problem with arbitrary width, then the later problem is reduced to the chromatic sum problem.

5. Application to VLSI routing. The previous result implies the $\mathcal{NP}$-completeness of other problems. The constraint graph of an instance of the single row routing problem in the Manhattan model is an undirected graph such that its vertices correspond to the nets and two vertices are adjacent if and only if the nets overlap. This is an interval graph and can be used to describe the single row routing problems in the dogleg-free Manhattan model. An assignment of tracks to nets that represents a legal routing is a coloring of the constraint graph. The colors are track numbers and the vertices represent nets. Each instance of the coloring problem of the interval graphs can be transformed into an instance of the single row routing problem in the dogleg-free Manhattan model and vice versa. Lengauer [5] shows a transformation of a coloring problem into a routing problem. Figure 8 shows the routing problem constructed by translating the instance of the interval placement problem in Figure 1.

The minimum width (w) can be determined in linear time. Now we are interested in the minimum wire length. The total length of the horizontal segments is the same in any realization of the routing. Thus the minimization of the wire length is equivalent to the minimization of the vertical segment length.

How can the total vertical segment length be determined? If net j is placed into track r_j and it has l_j terminals ($l_j \geq 2$), then the vertical wire length of this net is $r_j \cdot l_j$. If there are t nets, then the total vertical wire length is

$$v = \sum_{i=1}^{t} r_j \cdot l_j. \tag{3}$$

If altogether l'_i terminals belong to the nets in the ith track, then $v = \sum_{i=1}^{w} i \cdot l'_i$.

The minimum wire length single row routing problem is equivalent to the following: Is there a realization of the given single row routing with width w in the dogleg-free Manhattan model for which the value defined in (3) is at most k?

Using this it is easy to see that each instance of the minimum value coloring problem of the interval graphs can be transformed into an instance of the minimum wire length single row routing in the dogleg-free Manhattan model if the weights of the vertices are integers greater than 1. The minimum value interval placement problem can be reduced to the minimum value coloring problem and it remains $\mathcal{NP}$-complete even if each weight is greater than 1 because the construction shown in the proof of the $\mathcal{NP}$-completeness uses such weights. Thus we managed to reduce an $\mathcal{NP}$-complete problem to the minimum wire length single row routing problem in the dogleg-free Manhattan model so it is $\mathcal{NP}$-complete.

THEOREM 2. *Single row routing with minimum wire length is $\mathcal{NP}$-complete in the dogleg-free Manhattan model.*

If each net has the same number of terminals, then the number of terminals can be eliminated from the expression in (3). Thus, the unweighted minimum value interval placement problem can be reduced to the restricted version of the minimum wire length single row routing problem when each net has the same number of terminals. Consequently, the minimum wire length single row routing problem is $\mathcal{NP}$-complete even if each net has only two terminals.

Similarly, an obvious consequence of Lemma 7 is $\mathcal{NP}$-completeness of the minimum wire length single row routing problem with arbitrary width in the dogleg-free Manhattan model.

Theorem 2 implies that routing with minimum wire length is $\mathcal{NP}$-complete in the dogleg-free Manhattan model in case of any shape of the routing region. This has been proved previously in cases of channel and switchbox routing [4]. However, this is a new result in cases of routing around a rectangle as well as in cases of gamma routing where terminals appear on two adjacent sides of a rectangular. For example, LaPaugh [4] presented a polynomial algorithm which finds a layout with minimum area in the Manhattan model if terminals are on four sides of a rectangular and the wires lie on the outside of the rectangle. However, there is no polynomial algorithm for optimizing the wire length. By Theorem 2, this problem is $\mathcal{NP}$-complete and thus there is no hope to find such an algorithm.

6. Conclusions. In this paper we assigned a value to a placement of intervals and proved that the minimum value interval placement problem is $\mathcal{NP}$-complete. This result could be applied to the interval graphs which are among the most useful mathematical structures for modeling real-world problems. Thus we proved that the minimum value coloring of interval graphs is $\mathcal{NP}$-complete. The routing problem is known to be $\mathcal{NP}$-complete in many cases. In this paper we have shown that finding the minimum wire length is computationally difficult even in one of the simplest cases—single row routing in the dogleg-free Manhattan model. This completes the previous results on complexity of routing with minimum wire length in the dogleg-free Manhattan model.

Acknowledgments. Useful conversations with A. Recski are gratefully acknowledged. I thank Sara Nicoloso for her comments.

REFERENCES

[1] T. Gallai, unpublished; cited in A. Hajnal and J. Surányi, Über die auflösung von graphen in vollständige teilgraphen, Ann. Univ. Sci. Budapest Eötvös, 1 (1958), pp. 115–123.
[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1978.
[3] E. Kubicka and A. J. Schwenk, *An introduction to chromatic sums*, in Proceedings, ACM Computer Science Conference, Louisville, KY, 1989, pp. 39–45.
[4] A. S. LaPaugh, *A polynomial time algorithm for optimal routing around a rectangle*, in Proceedings, 21st Symposium on Foundations of Computer Science, Syracuse, NY, 1980, pp. 282–293.
[5] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, in Applicable Theory in Computer Science, John Wiley, Chichester, Teubner, Stuttgart, 1990, pp. 534–551.
[6] S. Nicoloso, M. Sarrafzadeh, and X. Song, *On the sum coloring problem on interval graphs*, Algorithmica, 23 (1999), pp. 109–126.
[7] A. Recski, *Minimax results and polynomial algorithms in VLSI routing*, Ann. Discrete Math., 44 (1992), pp. 261–273.
[8] T. Szkaliczki, *Optimal routing on narrow channels*, Period. Polytech. Ser. Elec. Engrg., 38 (1994), pp. 191–196.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 288–301

ON REGULAR TREE EMBEDDINGS*

WEIMIN CHEN† AND VOLKER TURAU‡

Abstract. Regular trees are a natural extension of finite trees, which have many applications. The path-embedding problem is to determine whether a regular tree S can be obtained from another regular tree T by deleting (probably infinitely many) subtrees of T. This paper explores efficient algorithms for the path-embedding problem in ordered and unordered trees. Given two regular trees S and T represented by rational graphs, our algorithms solve the ordered version of path-embedding problem in $O(|E_S||E_T|)$ time and the unordered version in $O(|E_S||E_T|D_SD_T)$ time. Here $|E_S|$ denotes the number of edges in the rational graph for S, and D_S denotes the maximum outdegree of a vertex in S. We also demonstrate that our approach can be applied to pattern matching problems for regular trees recently studied by Fu [*J. Algorithms*, 22 (1997), pp. 372–391].

Key words. regular trees, directed graph pattern matching, embedding, boolean equations

AMS subject classifications. 68Q25, 68Q20, 68R10

PII. S0097539797324308

1. Introduction.

A *labeled infinite tree* is a directed tree in which each node is assigned a unique label and each node has a finite number of children but the number of nodes in the tree may be infinite. Given a node v in a labeled infinite tree T, a *subtree* of T rooted at v is the tree induced by v and its descendants in T. A labeled infinite tree is called *regular* if it contains only finitely many distinct subtrees (two trees are regarded as distinct if they are not isomorphic). Further, a regular tree is called *ordered* (*unordered*) if the left-to-right order among siblings at each node is significant (insignificant, resp.). Figure 1 shows two examples of regular trees.

Regular trees are the basic tools for investigating tree automata, flow graphs [4], and type systems [2, 5]. In this paper we pose the following problem with respect to regular trees:

Given two regular trees S and T, can we obtain S from T by deleting (finitely or infinitely many) subtrees of T?

We call this problem the *ordered (unordered) regular tree path-embedding* problem if the regular trees S and T are ordered (unordered, resp.). The regular tree path-embedding problems are an extension of the finite tree path-embedding problems [7]. For ordered finite tree path-embedding problem, there is a simple $O(|S||T|)$ time algorithm [7], where S and T are the ordered finite trees. For the unordered finite tree path-embedding problem, efficient algorithms can be found in [8, 9, 11]. These algorithms run in $O(|S|^{3/2}|T|)$ time, where S and T both are unordered finite trees.

This paper explores efficient algorithms for the ordered and unordered regular tree embedding problems. The main idea of our approach is to treat the path-embedding algorithms as optimized procedures for finding a special solution of systems of boolean equations. By representing each regular tree as a finite rational graph [4], we solve the ordered version of the path-embedding in $O(|E_S||E_T|)$ time, and the unordered version in $O(|E_S||E_T|D_SD_T)$ time. Here, $|E_S|$ denotes the number of edges in the

*Received by the editors July 14, 1997; accepted for publication (in revised form) August 19, 1998; published electronically September 22, 1999.

http://www.siam.org/journals/sicomp/29-1/32430.html

†GMD—Integrated Publication and Information Systems Institute, Dolivostr. 15, 64293 Darmstadt, Germany (chen@darmstadt.gmd.de).

‡FH Wiesbaden, Fachbereich Informatik, Kurt Schumacher Ring 18, 65197 Wiesbaden, Germany (turau@fh-wiesbaden.informatik.de).

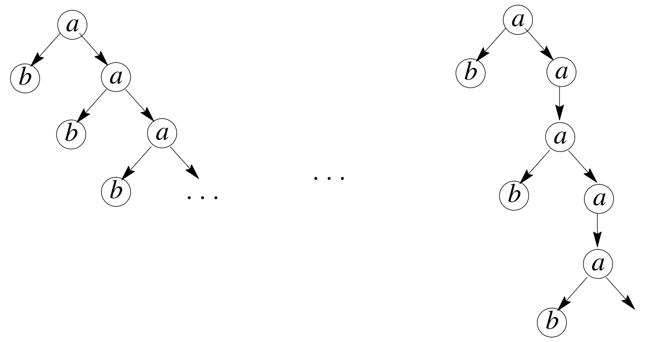


FIG. 1. *Two regular trees.*

rational graph for S, and D_S denotes the maximum outdegree of a vertex in S.

This paper is organized as follows. In section 2 we formally define the regular tree path-embedding problems and describe the data structure to represent regular trees. In sections 3 and 4, resp., we describe the ordered and unordered versions of our algorithms. Section 5 contains a comparison with related work. Finally, section 6 gives a short conclusion.

2. Basic definitions and rational graphs.

We give a formal definition for the path-embedding of ordered and unordered regular trees.

DEFINITION 2.1. *Let S and T be ordered regular trees. An injection $f : S \longrightarrow T$ is called an* ordered path-embedding *from S to T if*

(a) *f preserves roots, i.e., if r is the root of S, then $f(r)$ is the root of T;*
(b) *f preserves labels, i.e., $label(s) = label(f(s))$ for any node s of S;*
(c) *f preserves parents, i.e., if s_p is the parent of s, then $f(s_p)$ is the parent of $f(s)$;*
(d) *f preserves the order among siblings, i.e., if s_1 is a left sibling of s_2, then $f(s_1)$ is a left sibling of $f(s_2)$.*

Let $S \sqsubseteq^f T$ denote the fact that f is an ordered path-embedding of S into T.

For unordered regular trees S and T, an injective function $f : S \longrightarrow T$ is called an unordered path-embedding *of S into T if the above conditions* (a)–(c) *are satisfied. Let $S \sqsubseteq^f_{\mathrm{u}} T$ denote the fact that f is an unordered path-embedding of S into T.*

Further, let $S \sqsubseteq T$ and $S \sqsubseteq_{\mathrm{u}} T$, resp., denote that there exists a path-embedding f such that $S \sqsubseteq^f T$ and $S \sqsubseteq^f_{\mathrm{u}} T$.

In this paper we only consider the path-embedding problem and simply refer to it as the "embedding" problem.

Our first question is how to represent regular trees? Infinite regular trees can be expressed by finite systems. The natural approach is by means of rational graphs [4], defined as follows.

DEFINITION 2.2. *A* rational graph *$G = (V, E, r)$ is a finite directed graph, where V is the set of vertices, E is the set of edges, and $r \in V$ is the root such that*

(a) *all vertices of G are labeled over some alphabet;*
(b) *G may contain multiedges and loops;*
(c) *every vertex of G can be reached from the root r along a path of edges of G.*

A rational graph is called ordered (unordered)*, if the order of out-going edges at each vertex is significant (insignificant, resp.).*

Rational graphs are connected graphs. For completeness we show that an ordered regular tree can be expressed as an ordered rational graph, and vice versa. Let T be

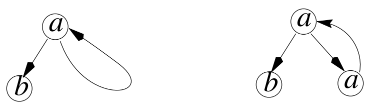


FIG. 2. *Rational graphs for regular trees in Figure* 1 *(top nodes are the roots).*

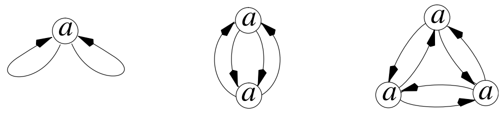


FIG. 3. *Rational graphs with multiedges for a full binary tree.*

an ordered regular tree and $S = \{T_1, \dots, T_s\}$ be the set of all distinct subtrees of T. The ordered rational graph $G = (V, E, r)$ of T can be constructed as follows. Let $V = \{v_1, \dots, v_s\}$, and let $label(v_i)$ equal to the label of the root of T_i. If $T_{i_1}, \dots, T_{i_l}$ (in S) are the directed subtrees of T_i from left to right, then $v_{i_1}, \dots, v_{i_l}$ are the successors of v_i from left to right. The root r is a vertex v_i such that T_i is equal to T. Conversely, given a rational graph $G = (V, E, r)$, a regular tree T can be constructed, level by level, by unfolding G starting at the root r (so T contains at most $|V|$ distinct subtrees). For example, the rational graphs in Figure 2 represent the regular trees in Figure 1.

Similarly, an unordered regular tree can be expressed as an unordered rational graph, and vice versa.

Note that a rational graph always represents a unique regular tree, but a regular tree may be represented by different rational graphs. Figure 3 depicts three different rational graphs (with multiedges) for the same regular tree.

We adopt the following notation to represent intervals of integers: Given integers a and b, let $[a..b]$ denote the set of integers n with $a \le n \le b$, and let $(a..b] = [a+1..b]$, $[a..b) = [a..b-1]$, and $(a..b) = [a+1..b-1]$.

3. Ordered embedding. Let S and T be ordered regular trees. Throughout this section let $G_S = (V_S, E_S, r_S)$ and $G_T = (V_T, E_T, r_T)$ denote the rational graphs for S and T. Given G_S and G_T, the task of the algorithm is to determine whether the relationship $S \sqsubseteq T$ holds.

For $(s, t) \in V_S \times V_T$, let $s \preceq t$ denote the following relation:

$$label(s) = label(t) \;\wedge\; d(s) \le d(t), \tag{1}$$

where $d(s)$ denotes the outdegree of vertex s. Furthermore, let $s \dot{\preceq} t$ be the abbreviation for

$$s \preceq t \;\wedge\; 0 < d(s).$$

The ith successor of s is denoted by $s[i]$. Let $\mathcal{B} = \{\mathbf{0}, \mathbf{1}\}$ denote the set of boolean values (represented in bold font).

Given a vertex v in rational graph G, let $tree(v)$ denote the regular tree obtained by unfolding G starting at v.

Before discussing the algorithm for the ordered regular tree embedding problem, let us look at the ordered finite tree embedding problem, which is also called the *ordered path inclusion* problem [7]. Suppose S and T both are ordered finite trees with roots r_S and r_T (so S and T can also be regarded as rational graphs). Let s and t be internal nodes in S and T (thus $\mathit{tree}(s)$ and $\mathit{tree}(t)$ are the subtrees rooted at s and t). To check whether $\mathit{tree}(s)$ can be embedded in $\mathit{tree}(t)$, the main idea behind the algorithm in [7] is to find a sequence of numbers $1 \le j_1 < j_2 < \cdots < j_{d(s)} \le d(t)$ such that $\mathit{tree}(s[i])$ can be embedded into $\mathit{tree}(t[j_i])$ for all $i \in [1..d(s)]$. Such a sequence can be found by scanning the children of s and t from left to right and thereby trying to find an embedding of the corresponding subtrees in a bottom-up fashion. The following function, *emb*, performs this operation, and a call of $\mathit{emb}(s,t)$ returns the value of the expression $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$.

```
(e1)    function $\mathit{emb}(s,t)$ returns $\mathcal{B}$
        { if $s \not\preceq t$ then return 0
          $i \leftarrow j \leftarrow 1$
          while $i \le d(s) \wedge j \le d(t)$ do
(e2)      { if $\mathit{emb}(s[i], t[j])$ then $i \leftarrow i+1$
            $j \leftarrow j+1$
          }
          return $(i = d(s)+1)$
        }
```

Let $\mathit{tree}(s,i)$ denote the tree obtained from $\mathit{tree}(s)$ by removing subtrees $\mathit{tree}(s[k])$, for $k \in (i..d(s)]$. Thus, $\mathit{tree}(s,d(s)) = \mathit{tree}(s)$. To check the correctness of the above function *emb*, notice that the invariant of the while-loop is

$$\mathit{tree}(s,i-1) \sqsubseteq \mathit{tree}(t,j-1) \ \wedge\ \mathit{tree}(s,i) \not\sqsubseteq \mathit{tree}(t,j-1).$$

We attempt to extend the above algorithm to general rational graphs. The main obstacle is that in rational graphs the ancestorship of vertices does not form a partial order due to the cycles in the graphs, and, consequently, the above bottom-up matching approach cannot be applied to rational graphs.

For the pattern matching problem presented in [5], a similar obstacle occurs. However, pattern matching preserves degrees; the path-embedding problem relaxing this constraint further complicates the algorithm. A detailed comparison is given in section 5.

3.1. Transformation. The main idea of our approach is to transform the tree embedding problem into the problem of finding a specific solution in a system of boolean equations. First, it is easy to derive the following lemma.

LEMMA 3.1. *For $(s,t) \in V_S \times V_T$, $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$ iff $s \preceq t$ and at least one of the following conditions holds:*

(a) $d(s) = 0$.

(b) *There exists a sequence of numbers $1 \le l_1 < l_2 < \cdots < l_{d(s)} \le d(t)$ such that $\mathit{tree}(s[i]) \sqsubseteq \mathit{tree}(t[l_i])$ for $i \in [1..d(s)]$.*

Let x_s^t be a boolean variable for each $(s,t) \in V_S \times V_T$. Consider the following system of boolean equations: $\forall (s,t) \in V_S \times V_T$,

$$x_s^t \;=\; s \preceq t \wedge \left((d(s)=0) \vee \bigvee_{1 \le l_1 < \cdots < l_{d(s)} \le d(t)} \bigwedge_{i=1}^{d(s)} x_{s[i]}^{t[l_i]} \right). \tag{2}$$

Lemma 3.1 guarantees that

$$x_s^t \;=\; (\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)),\ \ (s,t) \in V_S \times V_T \tag{3}$$

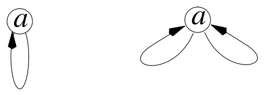


FIG. 4. *Two rational graphs.*

is a solution of (2). Hence we can ask how to find the solution (3) from (2). Note that (2) may have multiple solutions. For example, (2) corresponding to rational graphs shown in Figure 4 are $x_s^t = x_s^t \vee x_s^t$, for which there are two different solutions $x_s^t = \mathbf{0}$ and $x_s^t = \mathbf{1}$.

To obtain solution (3) from (2), we need the following concepts. Let $x_1, \ldots, x_n$ be boolean variables and $f_i : \mathcal{B}^n \longrightarrow \mathcal{B}$ for $i \in [1..n]$ boolean functions. Consider the following system of equations:

$$x_i = f_i(x_1, \ldots, x_n),\ \ i \in [1..n]. \tag{4}$$

DEFINITION 3.2. *A solution* $(a_1, \ldots, a_n)$ *for* (4) *is called the* maximum solution *if for any other solution* $(b_1, \ldots, b_n)$*,* $a_i \geq b_i$ *for all* $i \in [1..n]$*.*

Note that the maximum solution of (4) may not exist, but it is unique if it exists. Note that the above definition can be applied to (2) since they have the required form. This leads to the following result.

LEMMA 3.3. *The values of* x_s^t *given by expression* (3) *are the maximum solution of* (2)*.*

Proof. Lemma 3.1 proves that the values given in (3) are a solution of (2). Now we prove that they are the maximal solution. Let a_s^t with $(s,t) \in V_S \times V_T$ be any solution of (2). It suffices to show that $a_s^t \leq (\mathit{tree}(s) \sqsubseteq \mathit{tree}(t))$, i.e., if $a_s^t = \mathbf{1}$ then $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$. We prove this by constructing a mapping f from $\mathit{tree}(s)$ to $\mathit{tree}(t)$ as follows. Let $f(s) = t$. The condition $a_s^t = \mathbf{1}$ implies that $s \preceq t$. If $d(s) = 0$, then $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$. Now let $d(s) > 0$. Equations (2) guarantee the existence of numbers $1 \leq l_1 < l_2 < \cdots < l_{d(s)} \leq d(t)$ such that $a_{s[i]}^{t[l_i]} = \mathbf{1}$ for $i \in [1..d(s)]$. Applying the above argument with s and t replaced with $s[i]$ and $t[l_i]$, we define $f(s[i]) = t[l_i]$ for each $i \in [1..d(s)]$. Repeating this procedure level by level results in a mapping f from $\mathit{tree}(s)$ to $\mathit{tree}(t)$. Clearly, f satisfies all the conditions of Definition 2.1. Hence $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$. ◻

The question is how to find the maximum solution of (2). It turns out that the answer to this question can be found by considering a more general problem. Consider the following definition.

DEFINITION 3.4. *A boolean function* $f : \mathcal{B}^n \longrightarrow \mathcal{B}$ *is called* monotonic*, if* $f(x_1, \ldots, x_n) \leq f(y_1, \ldots, y_n)$ *for all* $(x_1, \ldots, x_n)$ *and* $(y_1, \ldots, y_n) \in \mathcal{B}^n$ *with* $x_i \leq y_i$*,* $i \in [i..n]$*. Equations* (4) *are called* monotonic*, if all functions* f_i *are monotonic.*

The following theorem provides a key foundation of our algorithms for the ordered and unordered embedding problems.

THEOREM 3.5. *For monotonic* (4)*, the maximum solution exists and can be effectively found.*

Proof. We show that the maximum solution can be constructed by the following algorithm.

```
**algorithm R**
{ **foreach** $i \in [1..n]$ **do** $x_i \leftarrow \mathbf{1}$
  **while** $\exists i \in [1..n]$ **such that** $x_i > f_i(x_1, \ldots, x_n)$ **do** $x_i \leftarrow \mathbf{0}$
}
```

Suppose that the above while-loop takes k iterations; as each iteration of the while-loop changes the value of one variable x_i from $\mathbf{1}$ to $\mathbf{0}$, it follows that $k \leq n$. Below we show that $\mathbf{R}$ generates the maximum solution of (4). Without loss of generality, assume the while-loop sequentially updates the variables $x_1, x_2, \ldots, x_k$ in that order to $\mathbf{0}$. For simplicity, let $\bar{x}$ denote the vector $(x_1, \ldots, x_n)$, $f_i(\bar{x})$ denote the function $f_i(x_1, \ldots, x_n)$, and $\bar{e}_i$ denote the vector $(\mathbf{0}, \ldots, \mathbf{0}, \mathbf{1}, \ldots, \mathbf{1})$ of length n (the i leading entries are $\mathbf{0}$, and the remaining $n-i$ entries are $\mathbf{1}$). Thus, $\bar{x} = \bar{e}_k$ when $\mathbf{R}$ terminates.

First we show that $\bar{x} = \bar{e}_k$ is a solution of (4). Algorithm $\mathbf{R}$ guarantees that

$$f_i(\bar{e}_{i-1}) = \mathbf{0}, \ i \in [1..k]. \tag{5}$$

As f_i is monotonic, it follows that $f_i(\bar{e}_k) \leq f_i(\bar{e}_{i-1}) = \mathbf{0}$ for $i \in [1..k]$ and hence $x_i = f_i(\bar{e}_k) = \mathbf{0}$. On the other hand, for $i \in (k..n]$ we have $x_i = f_i(\bar{e}_k) = \mathbf{1}$.

Next it is shown that $\bar{x} = \bar{e}_k$ is the maximum solution. Suppose that $\bar{a} = (a_1, \ldots, a_n)$ is another solution of (4). Let i be the minimal subscript of $\bar{a}$ with $a_i = \mathbf{1}$ (i.e., $\bar{a} \leq \bar{e}_{i-1}$). It suffices to prove that $k < i$. This can be inferred from relations (5) and $f_i(\bar{e}_{i-1}) \geq f_i(a_1, \ldots, a_n) = a_i = \mathbf{1}$. $\square$

3.2. Algorithms. It is easy to verify that (2) are monotonic. Thus, algorithm $\mathbf{R}$ can be applied to (2) for finding the maximum solution (3). The cost for this application is as follows. Let $F(s,t)$ denote the function expressing the right-hand side of (2). Using an approach similar to function *emb*, $F(s,t)$ can be computed in $O(d(t))$ time (more exactly, the procedure to compute $F(s,t)$ can be obtained from function *emb* by changing the term *emb*(s,t) at line (e1) into $F(s,t)$ and term *emb*$(s[i], t[j])$ at line (e2) into $x_{s[i]}^{t[j]}$). As the while-loop of $\mathbf{R}$ iterates at most $|V_S||V_T|$ times, the total time-cost of $\mathbf{R}$ is thus $O(|V_S||V_T| \sum_{(s,t)\in V_S \times V_T} d(t)) = O(|V_S|^2|V_T||E_T|)$ and the required space is bounded by $O(|V_S||V_T|)$.

This section gives a refinement of $\mathbf{R}$ specifically for (2). The algorithm runs in $O(|E_S||E_T|)$ time and requires $O(|E_S||V_T| + |E_T|)$ space. First we introduce the following concepts.

DEFINITION 3.6. *Let x_s^t be boolean variables for $(s,t) \in V_S \times V_T$. Given $(s,t) \in V_S \times V_T$ with $s \preceq t$, a list $(l_1, \ldots, l_{d(s)})$ is called a* positive list *for (s,t), if*

- $1 \leq l_1 < \cdots < l_{d(s)} \leq d(t)$;
- $x_{s[i]}^{t[j]} = \mathbf{0}$ *for $i \in [1..d(s)]$ and $j \in (l_{i-1}..l_i)$ with $l_0 = 0$.*

A list $(l_1, \ldots, l_k)$ with $k \in [1..d(s)]$ is called a negative list *for (s,t), if*

- $1 \leq l_1 < \cdots < l_k = d(t) + 1$;
- $x_{s[i]}^{t[j]} = \mathbf{0}$ *for $i \in [1..k]$ and $j \in (l_{i-1}..l_i)$ with $l_0 = 0$.*

Clearly, for $s \dot{\preceq} t$, $F(s,t) = \mathbf{1}$ implies the existence of a positive list, but the converse is not true, because we don't make any assumption on the values $x_{s[i]}^{t[l_i]}$ of the positive list. For negative lists we have the following result.

LEMMA 3.7. *Let $(s,t) \in V_S \times V_T$ with $s \dot{\preceq} t$. If there exists a negative list for (s,t) then $F(s,t) = \mathbf{0}$.*

Proof. Assume $F(s,t) = \mathbf{1}$. Then there exists a list $(a_1, \ldots, a_{d(s)})$ such that $1 \leq a_1 < \cdots < a_{d(s)} \leq d(t)$ and $x_{s[i]}^{t[a_i]} = \mathbf{1}$ for $i \in [1..d(s)]$. Let $(l_1, \ldots, l_k)$ be a negative list for (s,t) and $i \in [1..k]$ minimal such that $a_i < l_i$ (the relation $a_k \leq d(t) < l_k$ implies the existence of such i). Thus, $a_i > a_{i-1} \geq l_{i-1}$, where $a_0 = l_0 = 0$. Therefore, $a_i \in (l_{i-1}..l_i)$ and hence $x_{s[i]}^{t[a_i]} = \mathbf{0}$, which leads to a contradiction. $\square$

The new algorithm $\mathbf{R}_o$ for ordered tree embeddings uses the following variables:

- x_s^t of type $\mathcal{B}$ for each $(s,t) \in V_S \times V_T$, to indicate the truth value for relation $\mathit{tree}(s) \sqsubseteq \mathit{tree}(t)$.
- $\mathit{pred}[s]$ of type **set of** $(V_S \times \mathbb{N})$ for each $s \in V_S$, to represent the set of predecessors of s in S.
- $\mathit{pred}[t]$ of type **set of** $(V_T \times \mathbb{N})$ for each $t \in V_T$, to represent the set of predecessors of t in T.
- l_s^t of type **array**$[1..d(s)]$ **of** $\mathbb{N}$ for each $(s,t) \in V_S \times V_T$, storing a positive list (if any existed) for (s,t).

The above variables are initialized by the following routine:

```
        routine $init()$
        { foreach $s \in V_S$ do $pred[s] \leftarrow \{(u,i) \mid u[i] = s\}$
          foreach $t \in V_T$ do $pred[t] \leftarrow \{(v,j) \mid v[j] = t\}$
          foreach $(s,t) \in V_S \times V_T$ do
          { $x_s^t \leftarrow \mathbf{1}$
(i1)        if $s \stackrel{\cdot}{\preceq} t$ then $(l_s^t[1], \ldots, l_s^t[d(s)]) \leftarrow (1, \ldots, d(s))$
          }
        }
```

The main body of algorithm $\mathbf{R}_o$ is as follows:

```
        algorithm $\mathbf{R}_o$
        { $init()$
          foreach $(s,t) \in V_S \times V_T$ such that $s \not\preceq t \wedge x_s^t = \mathbf{1}$ do $set\_zero(s,t)$
        }
```

The above algorithm needs to invoke the subroutine $\mathit{set\_zero}(s,t)$ that assigns x_s^t to $\mathbf{0}$ and recursively propagates this change of this value.

```
        routine $set\_zero(s,t)$
(s1)    { $x_s^t \leftarrow \mathbf{0}$
          foreach $((u,i),(v,j)) \in pred[s] \times pred[t]$ such that
(s2)              $u \preceq v \wedge x_u^v = \mathbf{1} \wedge l_u^v[i] = j$ do
(s3)        if $find(u,i,v) = \mathbf{0}$ then $set\_zero(u,v)$
        }
```

The routine *set_zero* needs to call function $\mathit{find}(u,i,v)$, which attempts to find a new positive list for (u,v) to reflect the change of the value of x_s^t (i.e., $x_{u[i]}^{v[l_i]}$) and to store this list in l_u^v. The function returns **1** if this task can be accomplished or returns **0** otherwise; in the latter case there exists a negative list for (u,v).

In the body of function $\mathit{find}(u,i,v)$, for simplicity, let l_j be the short-hand for the variable $l_u^v[j]$.

```
        function $find(u,i,v)$ returns $\mathcal{B}$
        { while 1 do
          { repeat $l_i \leftarrow l_i + 1$ until $l_i > d(v) \vee x_{u[i]}^{v[l_i]} = \mathbf{1}$
            if $l_i > d(v)$ then return 0
            if $i < d(u) \wedge l_i \geq l_{i+1}$ then $\{l_i \leftarrow l_{i+1};\ i \leftarrow i + 1\}$
            else return 1
          }
        }
```

Below we prove the correctness of algorithm $\mathbf{R}_o$. For convenience let $\langle l_u^v, +\rangle$ denote the fact that $(l_1, \dots, l_{d(u)})$ is a positive list (recall that l_j is the short-hand for $l_u^v[j]$), and let $\langle l_u^v, -\rangle$ denote that there exists $j \in [1..d(u)]$ such that $(l_1, \dots, l_j)$ is a negative list. Furthermore, we introduce global boolean variables ρ_s^t for $(s,t) \in V_S \times V_T$, which are only used in the proofs. Initially, assume $\rho_s^t = \mathbf{1}$ for all $(s,t) \in V_S \times V_T$. The value of ρ_u^v is set in line (s3) to the value of *find*(u,i,v). The following lemma analyzes function *find*.

LEMMA 3.8. *Let $(u,v) \in V_S \times V_T$ with $u \dot{\preceq} v$. Consider the call of function find(u,i,v) for some $i \in [1..d(u)]$. Assume that before the call $\langle l_u^v, +\rangle$ holds. The call of the function causes the following effect to the list $(l_1, \dots, l_{d(s)})$:* (I) *The values of $l_{i_0}, \dots, l_{i_1}$ are increased, where i_0 and i_1 denote the initial and final values of variable i;* (II) *the values of $l_j, j \notin [i_0..i_1]$ are not changed;* (III) *one of the following properties holds:*

(a) $\rho_u^v = \mathbf{1}$, $\langle l_u^v, +\rangle$, *and* $x_{u[j]}^{v[l_j]} = \mathbf{1}$ *for all increased* l_j*;*

(b) $\rho_u^v = \mathbf{0}$ *and* $\langle l_u^v, -\rangle$.

Proof. Notice that the function *find* is invoked only at line (s3), where the value of $x_{u[i]}^{v[l_i]}$ is set to $\mathbf{0}$ at line (s1).

The following invariant of the while-loop in the function *find* is easy to prove. When $i > i_0$,

- the values of $l_{i_0}, \dots, l_{i-1}$ are increased and the value of l_i is not decreased;
- the values of $l_j, j \notin [i_0..i_1]$ are not changed;
- $l_{i_0} < \cdots < l_{i-1} = l_i \le d(v)$;
- $\forall j \in [i_0..i) : x_{u[j]}^{v[p]}$ equals $\mathbf{1}$ if $p = l_j$ and equals $\mathbf{0}$ if $p \in (l_{i-1}..l_i)$.

The last round of the loop always increases the value of l_i. Properties (I) and (II) are obvious. For property (III), if the function returns $\mathbf{1}$, then (a) holds; if the function returns $\mathbf{0}$ then (b) holds and in this case $(l_1, \dots, l_i)$ forms a negative list. □

For the next lemma suppose that the execution of $\mathbf{R}_o$ produces a series of timestamps $\omega_1, \omega_2, \dots, \omega_n$, each of which corresponds either to the end of an invocation of function *find* or to the end of an execution of line (s1). Furthermore, let ω_0 denote the timestamp produced at the beginning of $\mathbf{R}_o$.

LEMMA 3.9. *At time ω_k, $k \in [0..n]$, for each $(u,v) \in V_S \times V_T$ with $u \dot{\preceq} v$ one of the following properties holds:*

(a) $\rho_u^v = \mathbf{1}$ *and* $\langle l_u^v, +\rangle$;

(b) $\rho_u^v = \mathbf{0}$ *and* $\langle l_u^v, -\rangle$.

Proof. The proof is by induction on k. Property (a) holds in the initial case $k = 0$. Now let $k > 0$. If ω_k is related to the execution of line (s1), then in the time interval between ω_{k-1} and ω_k, nothing is changed except that x_u^v is set to $\mathbf{0}$. However, the change of the value of x_u^v does not affect properties (a) and (b) in this time interval. Now assume ω_k is related to the invocation of *find*(s,i,t). If $(s,t) \neq (u,v)$, then ρ_u^v and l_u^v are not changed during the time interval between ω_{k-1} and ω_k. Now let $(s,t) = (u,v)$. By Lemma 3.8 it suffices to show that $\langle l_u^v, +\rangle$ at time ω_{k-1}. By induction hypothesis it is enough to show that $\rho_u^v = \mathbf{1}$ at time ω_{k-1}. Assume contrapositively that this is not true. Then ρ_u^v must be assigned to $\mathbf{0}$ by a call of *find*(u,i',v) which terminates at time $\omega_{k'}$ with $k' < k$. Consequently, x_u^v is set to $\mathbf{0}$ at time $\omega_{k'+1}$. Note that $k'+1 = k$ is impossible as ω_k is related to function *find*(u,i,v). Thus, $k'+1 < k$. Hence, $x_u^v = \mathbf{0}$ after ω_{k-1}, and by line (s2) the function *find*(u,i,v) cannot be invoked after ω_{k-1}. This leads to a contradiction. □

Now we are in a position to prove the correctness of algorithm $\mathbf{R}_o$.

THEOREM 3.10. *Algorithm* $\mathbf{R}_\text{o}$ *is correct. That is, once* $\mathbf{R}_\text{o}$ *terminates,* $x_s^t = (\mathit{tree}(s) \sqsubseteq \mathit{tree}(t))$ *for each* $(s,t) \in V_S \times V_T$.

Proof. By Lemma 3.3 and Theorem 3.5, it suffices to prove the equivalence of algorithms $\mathbf{R}$ and $\mathbf{R}_\text{o}$ for (2). Recall that $F(s,t)$ denotes the right-hand side of (2). Thus we need to prove

(a) the precondition to the call of *set_zero*(s,t) is $x_s^t > F(s,t)$;

(b) at time ω_n there exists no (s,t) such that $x_s^t > F(s,t)$.

Proof of (a). The routine *set_zero*(s,t) must be invoked either by the main routine or by another invocation of *set_zero*. In any case $x_s^t = \mathbf{1}$ holds before the call of *set_zero*(s,t). Below we show that $F(s,t) = \mathbf{0}$ before the call of *set_zero*(s,t). If *set_zero*(s,t) is invoked by the main routine, then $F(s,t) = \mathbf{0}$ is obvious as $s \not\preceq t$. Now assume that *set_zero*(s,t) is invoked by another invocation of *set_zero* and that *set_zero*(s,t) is invoked at time between ω_{k-1} and ω_k. Thus, time ω_{k-1} is related to an invocation of *find*(s,i,t) that returns $\mathbf{0}$, i.e., $\rho_s^t = \mathbf{0}$ at time ω_{k-1}. Lemma 3.9 shows that at time ω_{k-1}, $\langle l_s^t, -\rangle$ and hence, by Lemma 3.7, $F(s,t) = \mathbf{0}$.

Proof of (b). Let $x_s^t = \mathbf{1}$ at time ω_n. It suffices to show that $F(s,t) = \mathbf{1}$ at ω_n. Notice that in the course of $\mathbf{R}_\text{o}$ the following condition always holds:

$$s \preceq t \text{ and } \rho_s^t = \mathbf{1}, \tag{6}$$

as otherwise *set_zero*(s,t) would be invoked either at line (r1) or line (s3), and hence $x_s^t = \mathbf{0}$ at time ω_n.

In condition (6), if $s \preceq t$ and $d(s) = 0$, then $F(s,t) = \mathbf{1}$ is obvious. Below we assume $d(s) > 0$ (i.e., replace $\preceq$ by $\dot{\preceq}$ in (6)).

Now Lemma 3.9 with condition (6) guarantees that $\langle l_s^t, +\rangle$ at ω_n. To derive $F(s,t) = \mathbf{1}$ it suffices to further prove that for each $i \in [1..d(s)]$,

$$x_{s[i]}^{t[l_i]} = \mathbf{1} \text{ at time } \omega_n, \tag{7}$$

where $l_s^t[i]$ is abbreviated by l_i. Now fix $i \in [1..d(s)]$. Assume ω_k is the earliest timestamp such that l_i is not updated since ω_k. Surely *set_zero*$(s[i], t[l_i])$ is never called since ω_k, as otherwise the call of it would lead to a call of *find*(s,i,t) (due to condition (6)), in which the value of l_i would be increased again, by property (I) of Lemma 3.8. Thus, to prove (7) it suffices to prove that

$$x_{s[i]}^{t[l_i]} = \mathbf{1} \text{ at time } \omega_k. \tag{8}$$

If $k = 0$ (i.e., l_i is not updated since the initialization) then (8) is obvious. Now let $k > 0$. Surely ω_k is related to an invocation of *find*(s,i',t) in which the value of l_i is increased. By condition (6), Lemma 3.9 ensures that $\langle l_s^t, +\rangle$ holds at any time ω_p. Thus at time ω_k the property (a) of Lemma 3.8 holds and hence (8) holds (as l_i is increased). ◻

THEOREM 3.11. *Algorithm* $\mathbf{R}_\text{o}$ *runs in* $O(|E_S||E_T|)$ *time and requires* $O(|E_S||V_T| + |E_T|)$ *space.*

Proof. We express all set-typed variables *pred*$[s]$, *pred*$[t]$ by lists that take total $O(|E_S|+|E_T|)$ space. On the other hand, all arrays l_s^t use $O(|E_S||V_T|)$ space. Further, notice that the depth of the stack for (recursive) routine invocations *set_zero* never exceeds $|V_S||V_T|$.

For the time-cost, the routine *init* is accomplished in $O(|E_S||E_T|)$ time. Not counting the time for the invocations of *set_zero*, the main routine of $\mathbf{R}_\text{o}$ runs in time $O(|E_S||E_T|)$. Excluding the cost for the invocations of *find*, all invocations of *set_zero*(s,t) for all $(s,t) \in V_S \times V_T$ need time $O(\sum_{(s,t)\in V_S\times V_T}(d^-(s)d^-(t)+1)) = O(|E_S||E_T|)$, where $d^-(v)$ denotes the indegree of a vertex v.

It remains to count the cost for all invocations of the function *find*. Note that a call *find*(u,i,v) is performed at most once for a fixed triple (u,i,v). On the other hand, the function *find*(u,i,v) runs in time $O(\delta)$, where δ is the overall increment of entries $l_u^v[k]$, for $k \in [1..d(u)]$, caused by the function itself. Notice that during the course of the algorithm, every entry $l_u^v[k]$ is never decreased and its upper bound is $d(v)+1$. Thus, for a fixed pair (u,v), all calls of *find*(u,i,v) run in overall $O(d(u)d(v))$ time (here $d(u)d(v) > 0$ as $u \mathrel{\dot{\preceq}} v$). Consequently the total cost of all invocations of *find*(u,i,v) is again $O(|E_S||E_T|)$. $\square$

4. Unordered embedding. Having presented the algorithm for ordered embedding, we present in this section an algorithm for the unordered embedding problem. Analogous to the approach developed in the previous section, the unordered embedding problem is transformed into the problem of how to find the maximum solution of a system of boolean equations.

For every $(s,t) \in V_S \times V_T$, let

$$\Re_s^t = \left\{(l_1,\ldots,l_{d(s)}) \in [1..d(t)]^{d(s)} \;\middle|\; \forall i,j \in [1..d(s)] :\ i = j \vee l_i \neq l_j\right\}.$$

The following lemma is analogous to Lemma 3.1 and can be easily derived from Definition 2.1.

LEMMA 4.1. *Let $(s,t) \in V_S \times V_T$. Then tree$(s) \sqsubseteq_{\mathrm{u}}$ tree(t) iff $s \preceq t$ and at least one of the following conditions holds:*

(a) $d(s) = 0$.

(b) *There exists a sequence $(l_1,\ldots,l_{d(s)}) \in \Re_s^t$ such that tree$(s[i]) \sqsubseteq_{\mathrm{u}}$ tree$(t[l_i])$ for $i \in [1..d(s)]$.*

Now take into account the system of the following monotonic equations: $\forall (s,t) \in V_S \times V_T$,

$$x_s^t \;=\; s \preceq t \,\wedge\, \left((d(s)=0) \vee \bigvee_{(l_1,\ldots,l_{d(s)})\in\Re_s^t} \bigwedge_{i=1}^{d(s)} x_{s[i]}^{t[l_i]}\right). \tag{9}$$

Lemma 4.1 shows that

$$x_s^t = (tree(s) \sqsubseteq_{\mathrm{u}} tree(t)),\ \ (s,t) \in V_S \times V_T \tag{10}$$

is a solution of (9). Furthermore, the following lemma can be proved along the same lines as in Lemma 3.3.

LEMMA 4.2. *The values of x_s^t given by expression* (10) *are the maximum solution of* (9)*.*

Similar to the previous section, we present a refinement of algorithm **R** for monotonic (9). This refinement also benefits from the idea of unordered finite tree embedding algorithms [9, 11] that use an efficient algorithm to compute maximum matching in bipartite graphs to achieve a good efficiency. More concretely, a *matching* in a graph $G = (V,E)$ is a subset of E with no two edges incident upon the same vertex in V. A *maximum matching* is a matching with the maximum number of edges. A graph $G = (V,E)$ is *bipartite* if V can be partitioned into two disjoint sets X and Y such that $E \subseteq X \times Y$. Given a bipartite graph $G = (V,E)$, the Hopcroft–Karp algorithm [6] finds a maximum matching of G in $O(s^{1/2}|E|)$ time, where s is the size of maximum matching. Assume S and T both are unordered finite trees. The unordered finite tree embedding problem is to determine the relation $S \sqsubseteq_u T$ as defined in Definition 2.1. By the use of the above bipartite matching algorithm the unordered finite tree embedding problem can be solved in $O(|S|^{3/2}|T|)$ time [9, 11].

For the unordered regular tree embedding problem, however, the direct use of Hopcroft–Karp algorithm is not very efficient, as under our circumstance all bipartite graphs and their maximum matchings are incrementally maintained. For our problem it suffices to adopt a simple incremental algorithm. Detailed ideas will be presented later.

The following algorithm $\mathbf{R}_u$ is a refinement of $\mathbf{R}$ for (9) that uses the following variables:

- x_s^t of type $\mathcal{B}$ for each $(s,t) \in V_S \times V_T$;
- *pred*$[s]$ and *pred*$[t]$, resp., of type **set of** $(V_S \times \mathbb{N})$ and **set of** $(V_T \times \mathbb{N})$ for each $s \in V_S$ and $t \in V_T$.

The above variables are initialized by the following routine:

```
routine init()
{ foreach $s \in V_S$ do $pred[s] \leftarrow \{(u,i) \mid u[i] = s\}$
  foreach $t \in V_T$ do $pred[t] \leftarrow \{(v,j) \mid v[j] = t\}$
  foreach $(s,t) \in V_S \times V_T$ do $x_s^t \leftarrow \mathbf{1}$
  foreach $(s,t) \in V_S \times V_T$ such that $s \dot{\preceq} t$ do
    create a complete bipartite graph $G_s^t = (V_s^t, E_s^t)$, where
      $V_s^t = \{(s,1),\ldots,(s,d(s)),\ (t,1),\ldots,(t,d(t))\}$
      $E_s^t = \{(s,i) \rightarrow (t,j) \mid i \in [1..d(s)],\ j \in [1..d(t)]\}$
}
```

The main body of algorithm $\mathbf{R}_u$ is as follows:

```
algorithm $\mathbf{R}_u$
{ init()
  foreach $(s,t) \in V_S \times V_T$ such that $s \not\preceq t \wedge x_s^t = \mathbf{1}$ do set_zero(s,t)
}

routine set_zero(s,t)
{ $x_s^t \leftarrow \mathbf{0}$
  foreach $((u,i),(v,j)) \in pred[s] \times pred[t]$ do
  { remove edge $(u,i) \rightarrow (v,j)$ from $G_u^v$
    if $x_u^v = \mathbf{1}$ then
(s)    { compute $k$, the size of maximum matching on $G_u^v$
      if $k < d(u)$ then set_zero(u,v)
    }
  }
}
```

For the proof of the correctness of algorithm $\mathbf{R}_u$, assume the execution of $\mathbf{R}_u$ produces a series of timestamps $\omega_1,\ldots,\omega_n$, where ω_i corresponds to the start time for the ith call of routine *set_zero*. Also, let ω_{n+1} denote the timestamps produced at the end of algorithm $\mathbf{R}_u$.

LEMMA 4.3. *Let $(u,v) \in V_S \times V_T$, $u \dot{\preceq} v$, and $(i,j) \in [1..d(u)] \times [1..d(v)]$. At every time ω_k, $x_{u[i]}^{v[j]} = \mathbf{0}$ iff there exists no edge connecting (u,i) and (v,j) in G_u^v.*

Proof. It can be directly derived from routine *set_zero.* $\square$

THEOREM 4.4. *Algorithm $\mathbf{R}_{\mathrm{u}}$ is correct. That is, once $\mathbf{R}_{\mathrm{u}}$ terminates, $x_s^t = (tree(s) \sqsubseteq_{\mathrm{u}} tree(t))$ for each $(s,t) \in V_S \times V_T$.*

Proof. By Theorem 3.5 and Lemma 4.2, it suffices to prove the equivalence of algorithms $\mathbf{R}$ and $\mathbf{R}_{\mathrm{u}}$ for (9). Let $E(s,t)$ denote the expression at the right-hand side of (9). Now we must prove that

(a) the precondition to the call of *set_zero*(u,v) is $x_u^v > E(u,v)$;

(b) at time ω_{n+1} there exists no $(u,v) \in V_S \times V_T$ with $x_u^v > E(u,v)$.

Proof of (a). The routine *set_zero*(u,v) is invoked either by the main routine or recursively by itself. In the first case condition (a) is clearly satisfied. If *set_zero*(u,v) is called by another invocation of *set_zero*, then $x_u^v = \mathbf{1}$ and a maximum matching for bipartite graph G_u^v has less than $d(u)$ edges. Lemma 4.3 shows that $E(u,v) = \mathbf{0}$.

Proof of (b). Assume contrapositively that there exists (u,v) such that $x_u^v = \mathbf{1}$ and $E(u,v) = \mathbf{0}$ at time ω_{n+1}. Surely *set_zero*(u,v) is never called, so we have $u \dot{\preceq} v$ and $E(u,v) = \mathbf{1}$ at time ω_1. Consider the smallest $k > 1$ such that $E(u,v) = \mathbf{0}$ at ω_k. Thus $E(u,v) = \mathbf{1}$ at ω_{k-1}. In the time interval between ω_{k-1} and ω_k, *set_zero*$(u[i], v[j])$, for some i and j, must be called, and in this routine setting $x_{u[i]}^{v[j]}$ to $\mathbf{0}$ causes $E(u,v) = \mathbf{0}$ at time ω_k. By Lemma 4.3, the relation $E(u,v) = \mathbf{0}$ implies that the maximum matching of G_u^v has size less than $d(u)$. Consequently, *set_zero*(u,v) must be invoked in the call of *set_zero*$(u[i], v[j])$. This induces a contradiction. $\square$

The algorithm $\mathbf{R}_{\mathrm{u}}$ needs to incrementally compute the size of maximum matching on bipartite graphs G_u^v (line (s)). Kilpeläinen uses a similar incremental approach for the *unordered finite tree region inclusion* problem [7]. The basic idea in our context is as follows. After the routine *init* constructs the complete bipartite graphs G_u^v for $(u,v) \in V_S \times V_T$, some edges in these graphs are removed by routine *set_zero.* During the course of the algorithm, we keep a maximum matching M_u^v for each bipartite graph G_u^v. If an edge, say e, is removed from G_u^v, then the size of maximum matching of $G_u^v \backslash e$, the graph G_u^v after removing e, is reduced at most by one. With this property the maximum matching of $G_u^v \backslash e$ can be computed as follows. If $e \notin M_u^v$, then M_u^v is still the maximum matching of $G_u^v \backslash e$. If $e \in M_u^v$, then $M_u^v - \{e\}$ is a matching (but probably not the maximum one) of G_u^v, and it suffices to check the existence of an augmenting path[1] with respect to matching $M_u^v - \{e\}$ in $G_u^v \backslash e$, as each augmenting path extends the matching by one edge. It is known that an augmenting path can be computed in time linear to the size of the graph [10]. Thus, the maximum matching of graph $G_u^v \backslash e$ can be computed in $O(|E_u^v|)$ time, based on old matching M_u^v. Taking this strategy to compute maximum matching we obtain the following result.

THEOREM 4.5. *Algorithm $\mathbf{R}_{\mathrm{u}}$ runs in $O(|E_S||E_T|\, D_S\, D_T)$ time and requires $O(|E_S||E_T|)$ space, where D_S and D_T denote the maximum outdegrees in G_S and G_T.*

Proof. We represent the set-typed variables *pred*$[s]$, *pred*$[t]$ by lists, and represent the edges in each bipartite graph G_s^t by a $d(s) \times d(t)$ adjacency matrix.

The space requirement is dominated by the space needed to store the bipartite graphs. Each graph G_s^t requires $d(s)d(t)$ space. Hence the total space required is $O(|E_S||E_T|)$.

[1]An *augmenting path* with respect to a matching M in a bipartite graph $G = (V,E)$ is a simple path consisting of edges alternatively in M and $E - M$, while both endpoints of the path are not incident upon any edge in M.

To count the time-cost of $\mathbf{R}_{\mathrm{u}}$, we first measure the cost for all repeated computations of bipartite maximum matchings. Afterwards, it is easy to see that this cost dominates the cost for the other steps in $\mathbf{R}_{\mathrm{u}}$.

Assume after the routine *init* we construct a maximum matching M_s^t for each complete bipartite graph G_s^t in time $O(d(s))$. As there are at most $d(s)d(t)$ edges to be removed in G_s^t, the total time-cost of computing maximum matchings (and their sizes) in G_s^t is $O(d(s)^2\, d(t)^2)$. Thus, the time-cost for all repeated computations of bipartite maximum matching is

$$\sum_{(s,t)\in V_S\times V_T} O\bigl(d(s)^2 d(t)^2\bigr) \;=\; O\bigl(|E_S||E_T|D_S D_T\bigr). \tag{11}$$

It is easy to check that the time-cost for other steps in $\mathbf{R}_{\mathrm{u}}$ is dominated by (11). Especially, the routine *init* runs in $O(|E_S||E_T|)$ time. $\Box$

5. Comparisons with related work. Aside from previous work on the path-embedding for finite trees, there exists other related work on pattern matching for regular trees and graphs. Closely related work is due to Fu, who considers two kinds of problems for regular trees [5]. The first one, denoted $\mathbf{M}$, is a pattern matching problem. We say that an ordered regular tree S *matches* another one T if there is a injection $f : S \longrightarrow T$ that preserves roots, labels, degrees (except for the leaves of S), parents, and order of siblings. In other words, S matches T iff

$$S \sqsubseteq^f T \;\wedge\; (\forall s \in S : d(s) = 0 \vee d(s) = d(f(s))).$$

Given two ordered rational graphs $G_S = (E_S, V_S, r_S)$ and $G_T = (V_T, E_T, r_T)$, the ordered version of the problem $\mathbf{M}$, denoted $\mathbf{M}_{\mathrm{o}}$, is to determine if *tree*(r_S) matches *tree*(t) for each $t \in V_T$. The unordered version of $\mathbf{M}$ is denoted $\mathbf{M}_{\mathrm{u}}$; here the order among siblings is not necessarily preserved.

The difference between problem $\mathbf{M}$ and the path-embedding problem considered in this paper is the preservation of the degrees of the vertices. In problem $\mathbf{M}_{\mathrm{o}}$, the relation of whether s matches t can be checked recursively by the following condition:

$$\mathit{label}(s) = \mathit{label}(t) \;\wedge\; (d(s) = 0 \;\vee\; d(s) = d(t) \wedge \forall i \in [1..d(s)] : s[i] \text{ matches } t[i])\,.$$

Using this property Fu gives an $O(|E_S||V_T| + |E_T|)$ time and space algorithm for problem $\mathbf{M}_{\mathrm{o}}$. Note that the algorithm for $\mathbf{M}_{\mathrm{o}}$ does not need to search alternative positive lists as required in algorithm $\mathbf{R}_{\mathrm{o}}$ because of the condition to preserve degrees.

On the other hand, for problem $\mathbf{M}_{\mathrm{u}}$, the unordered version of $\mathbf{M}$, Fu asserts that it is NP-complete, but no proof is given. In fact, the problem $\mathbf{M}_{\mathrm{u}}$ can be solved by our algorithm $\mathbf{R}_{\mathrm{u}}$ with a slight modification of the relation $\preceq$ (originally defined in (1)) as follows: for $(s,t) \in V_S \times V_T$, $s \preceq t$ is defined as

$$\mathit{label}(s) = \mathit{label}(t) \;\wedge\; (d(s) = 0 \;\vee\; d(s) = d(t))\,. \tag{12}$$

Under this modification the proof of the correctness of algorithm $\mathbf{R}_{\mathrm{u}}$ is still valid for $\mathbf{M}_{\mathrm{u}}$. Also this modification does not alter the time and space complexities of the algorithm. As a result, the problem $\mathbf{M}_{\mathrm{u}}$ can be solved in polynomial time and space by the above variant of algorithm $\mathbf{R}_{\mathrm{u}}$ and Theorem 4.5.

Fu also considers another problem, denoted $\mathbf{E}$, that tests regular tree topological embedding. A regular tree S is *topological embeddable* in T if there is an injection

$f : S \longrightarrow T$ that preserves roots, labels, degrees (except for the leaves of S), and the following condition:

$$\forall s \in S \ \wedge \ i \in [1..d(s)] : f(s[i]) \in \mathit{tree}(f(s)[i]).$$

The ordered version of $\mathbf{E}$, denoted $\mathbf{E}_{\mathrm{o}}$, is to test whether $\mathit{tree}(r_S)$ is topological embeddable in $\mathit{tree}(t)$ for each $t \in V_T$. A similar argumentation can be applied to problem $\mathbf{E}_{\mathrm{u}}$, the unordered version of $\mathbf{E}$. Fu presents an $O(|V_S||E_T| + |E_S|)$ time and $O(|V_S||V_T| + |E_S| + |E_T|)$ space algorithm for problem $\mathbf{E}_{\mathrm{o}}$ and asserts the NP-completeness of problem $\mathbf{E}_{\mathrm{u}}$. Note that our algorithms $\mathbf{R}_{\mathrm{o}}$ and $\mathbf{R}_{\mathrm{u}}$ cannot be applied to problems $\mathbf{E}_{\mathrm{o}}$ and $\mathbf{E}_{\mathrm{u}}$ with trivial extensions.

Another related problem is the *directed subgraph isomorphism* problem, which tests whether a given unordered digraph is a subgraph (under isomorphism) of another one. This problem is NP-complete [3, 8]. The subgraph isomorphism problem takes a stronger condition than that in the unordered path-embedding problem. Concretely, if G_S is subgraph of G_T and s isomorphically corresponds to t, then $\mathit{tree}(s) \sqsubseteq_{\mathrm{u}} \mathit{tree}(t)$, but the converse is not true. For example, in Figure 3 all graphs represent the same regular tree but none of them is a subgraph of any of the others.

6. Conclusion. We presented two algorithms for the ordered and unordered regular tree path-embedding problems by treating the algorithms as optimized procedures to find the maximal solution of a system of boolean equations that describe the pairwise matches between the vertices in pattern and target trees. As a result, the ordered version of our algorithm runs in quadratic time (Theorem 3.11) and the unordered version in quartic time (Theorem 4.5).

Acknowledgment. Rimli Sengupta and the anonymous referees made helpful comments on a draft of this paper.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers—Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[3] S. A. Cook, *The complexity of theorem-proving procedures*, in Proc. of 3rd Annual Symposium on the Theory of Computing, 1971, pp. 151–158.

[4] B. Courcelle, *Fundamental properties of infinite trees*, Theoret. Comput. Sci., 25 (1983), pp. 95–169.

[5] J. J. Fu, *Directed graph pattern matching and topological embedding*, J. Algorithms, 22 (1997), pp. 372–391.

[6] J. E. Hopcroft and R. M. Karp, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[7] P. Kilpeläinen, *Tree Matching Problems with Applications to Structured Text Databases*, Ph.D. thesis, Dept. of Comp. Science, Univ. of Helsinki, Helsinki, Finland, Nov., 1992.

[8] J. van Leeuwen, *Graph algorithms*, J. van Leeuwen, ed., in Handbook of Theoretical Computer Science, Algorithms and Complexity, Elsevier Science Publishers B. V., Amsterdam, 1990, pp. 525–631.

[9] D. W. Matula, *An algorithm for the subtree identification*, SIAM Rev., 10 (1968), pp. 273–274.

[10] P. S. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ, 1982.

[11] S. W. Reyner, *An analysis of a good algorithm for the subtree problem*, SIAM J. Comput., 6 (1977), pp. 730–732.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 302–326

EQUIVALENCE OF MEASURES OF COMPLEXITY CLASSES[*]

JOSEF M. BREUTZMANN[†] AND JACK H. LUTZ[‡]

Abstract. The resource-bounded measures of complexity classes are shown to be robust with respect to certain changes in the underlying probability measure. Specifically, for any real number $\delta > 0$, any uniformly polynomial-time computable sequence $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ of real numbers (biases) $\beta_i \in [\delta, 1-\delta]$, and for any complexity class $\mathcal{C}$ (such as P, NP, BPP, P/Poly, PH, PSPACE, etc.) that is closed under positive, polynomial-time, truth-table reductions with queries of at most linear length, it is shown that the following two conditions are equivalent.

(1) $\mathcal{C}$ has p-measure 0 (respectively, measure 0 in E, measure 0 in E_2) relative to the coin-toss probability measure given by the sequence $\vec{\beta}$.

(2) $\mathcal{C}$ has p-measure 0 (respectively, measure 0 in E, measure 0 in E_2) relative to the uniform probability measure.

The proof introduces three techniques that may be useful in other contexts, namely, (i) the transformation of an efficient martingale for one probability measure into an efficient martingale for a "nearby" probability measure; (ii) the construction of a *positive bias reduction*, a truth-table reduction that encodes a positive, efficient, approximate simulation of one bias sequence by another; and (iii) the use of such a reduction to *dilate* an efficient martingale for the simulated probability measure into an efficient martingale for the simulating probability measure.

Key words. complexity classes, martingales, resource-bounded measure

AMS subject classification. 68Q15

PII. S0097539796302269

1. Introduction.

In the 1990s, the measure-theoretic study of complexity classes has yielded a growing body of new, quantitative insights into various much-studied aspects of computational complexity. Benefits of this study to date include improved bounds on the densities of hard languages [15]; newly discovered relationships among circuit-size complexity, pseudorandom generators, and natural proofs [21]; strong new hypotheses that may have sufficient explanatory power (in terms of provable, plausible consequences) to help unify our present plethora of unsolved fundamental problems [18, 15, 7, 17, 14]; and a new generalization of the completeness phenomenon that dramatically enlarges the set of computational problems that are provably strongly intractable [13, 6, 2, 7, 8, 1]. See [12] for a survey of these and related developments.

Intuitively, suppose that a language $A \subseteq \{0,1\}^*$ is chosen according to a random experiment in which an independent toss of a fair coin is used to decide whether each string is in A. Then *classical* Lebesgue measure theory (described in [5, 20], for example) identifies certain *measure* 0 sets X of languages, for which the probability that $A \in X$ in this experiment is 0. *Effective* measure theory, which says what it means for a set of decidable languages to have measure 0 as a subset of the set of all such languages, has been investigated by Freidzon [4], Mehlhorn [19], and others. The *resource-bounded* measure theory introduced by Lutz [11] is a powerful generalization of Lebesgue measure. Special cases of resource-bounded measure include classical

[*]Received by the editors April 16, 1996; accepted for publication (in revised form) November 17, 1997; published electronically September 22, 1999. This research was supported in part by National Science Foundation grants CCR-9157382 (with matching funds from Rockwell, Microware Systems Corporation, and the Amoco Foundation) and CCR-9610461.

http://www.siam.org/journals/sicomp/29-1/30226.html

[†]Department of Mathematics and Computer Science, Wartburg College, Waverly, IA 50677.

[‡]Department of Computer Science, Iowa State University, Ames, IA 50011 (lutz@cs.iastate.edu).

Lebesgue measure; a strengthened version of effective measure; and most importantly, measures in $\mathrm{E} = \mathrm{DTIME}(2^{\mathrm{linear}})$, $\mathrm{E}_2 = \mathrm{DTIME}(2^{\mathrm{polynomial}})$, and other complexity classes. The *small* subsets of such a complexity class are then the measure 0 sets; the *large* subsets are the measure 1 sets (complements of measure 0 sets). We say that *almost every* language in a complexity class $\mathcal{C}$ has a given property if the set of languages in $\mathcal{C}$ that exhibit the property has measure 1 in $\mathcal{C}$.

All work to date on the measure-theoretic structure of complexity classes has employed the resource-bounded measure that is described briefly and intuitively above. This resource-bounded measure is based on the *uniform* probability measure, corresponding to the fact that the coin tosses are fair and independent in the above-described random experiment. The uniform probability measure has been a natural and fruitful starting point for the investigation of resource-bounded measure (just as it was for the investigation of classical measure), but there are good reasons to also investigate resource-bounded measures that are based on other probability measures. For example, the study of such alternative resource-bounded measures may be expected to have the following benefits.

(i) The study will enable us to determine which results of resource-bounded measure are particular to the uniform probability measure and which are not. This, in turn, will provide some criteria for identifying contexts in which the uniform probability measure is, or is not, the natural choice.

(ii) The study is likely to help us understand how the complexity of the underlying probability measure interacts with other complexity parameters, especially in such areas as algorithmic information theory, average case complexity, cryptography, and computational learning, where the variety of probability measures already plays a major role.

(iii) The study will provide new tools for proving results concerning resource-bounded measure based on the uniform probability measure.

The present paper initiates the study of resource-bounded measures that are based on nonuniform probability measures.

Let $\mathbf{C}$ be the set of all languages $A \subseteq \{0,1\}^*$. (The set $\mathbf{C}$ is often called *Cantor space.*) Given a probability measure ν on $\mathbf{C}$ (a term defined precisely below), section 3 of this paper describes the basic ideas of resource-bounded ν-measure, generalizing definitions and results from [11, 13, 12] to ν in a natural way. In particular, section 3 specifies what it means for a set $X \subseteq \mathbf{C}$ to have p-ν-measure 0 (written $\nu_{\mathrm{p}}(X) = 0$), p-ν-measure 1, ν-measure 0 in E (written $\nu(X|\mathrm{E}) = 0$), ν-measure 1 in E, ν-measure 0 in E_2, or ν-measure 1 in E_2. Intuitively, for example, a set $X \subseteq \mathbf{C}$ has p-ν-measure 0 if, when a language $A \in \mathbf{C}$ is chosen probabilistically according to the probability measure ν, the event "$A \in X$" has probability 0 in a strong sense that can be exploited by a polynomial-time betting algorithm.

Most of the results in the present paper concern a restricted (but broad) class of probability measures on $\mathbf{C}$, namely, coin-toss probability measures that are given by P-computable, strongly positive sequences of biases. These probability measures are described intuitively in the following paragraphs (and precisely in section 3).

Given a sequence $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ of real numbers (biases) $\beta_i \in [0,1]$, the *coin-toss probability measure* (also call the *product probability measure*) *given by* $\vec{\beta}$ is the probability measure $\mu^{\vec{\beta}}$ on $\mathbf{C}$ that corresponds to the random experiment in which a language $A \in \mathbf{C}$ is chosen probabilistically as follows. For each string s_i in the standard enumeration $s_0, s_1, s_2, \dots$ of $\{0,1\}^*$, we toss a special coin, whose probability is β_i of coming up heads, in which case $s_i \in A$, and $1 - \beta_i$ of coming up tails, in which case $s_i \notin A$. The coin tosses are independent of one another.

In the special case where $\vec{\beta} = (\beta, \beta, \beta, \dots)$, i.e., the biases in the sequence $\vec{\beta}$ are all β, we write μ^{β} for $\mu^{\vec{\beta}}$. In particular, $\mu^{\frac{1}{2}}$ is the uniform probability measure, which, in the literature of resource-bounded measure, is denoted simply by μ.

A sequence $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ of biases is *strongly positive* if there is a real number $\delta > 0$ such that each $\beta_i \in [\delta, 1-\delta]$. The sequence $\vec{\beta}$ is P-*computable* (and we call it a P-*sequences of biases*) if there is a polynomial-time algorithm that, on input $(s_i, 0^r)$, computes a rational approximation of β_i to within 2^{-r}.

In section 4, we prove the summable equivalence theorem, which implies that, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive P-sequences of biases that are "close" to one another, in the sense that $\sum_{i=0}^{\infty} |\alpha_i - \beta_i| < \infty$, then for every set $X \subseteq \mathbf{C}$,

$$\mu_{\mathrm{p}}^{\vec{\alpha}}(X) = 0 \Longleftrightarrow \mu_{\mathrm{p}}^{\vec{\beta}}(X) = 0.$$

That is, the p-measure based on $\vec{\alpha}$ and the p-measure based on $\vec{\beta}$ are in absolute agreement as to which sets of languages are small.

In general, if $\vec{\alpha}$ and $\vec{\beta}$ are not in some sense close to one another, then the p-measures based on $\vec{\alpha}$ and $\vec{\beta}$ need not agree in the above manner. For example, if $\alpha, \beta \in [0,1]$, $\alpha \neq \beta$, and

$$X_{\alpha} = \left\{ A \in \mathbf{C} \middle| \lim_{n \to \infty} 2^{-n} |A \cap \{0,1\}^n| = \alpha \right\},$$

then a routine extension of the weak stochasticity theorem of [15] shows that $\mu_{\mathrm{p}}^{\alpha}(X_{\alpha}) = 1$, while $\mu_{\mathrm{p}}^{\beta}(X_{\alpha}) = 0$.

Notwithstanding this example, many applications of resource-bounded measure do not involve *arbitrary* sets $X \subseteq \mathbf{C}$, but rather are concerned with the measures of *complexity classes* and other closely related classes of languages. Many such classes of interest, including P, NP, co-NP, R, BPP, AM, P/Poly, PH, PSPACE, etc., are closed under positive, polynomial-time truth-table reductions ($\leq_{\mathrm{pos-tt}}^{\mathrm{P}}$-reductions), and their intersections with E are closed under $\leq_{\mathrm{pos-tt}}^{\mathrm{P}}$-reductions with linear bounds on the lengths of the queries ($\leq_{\mathrm{pos-tt}}^{\mathrm{P,lin}}$-reductions).

The main theorem of this paper is the bias equivalence theorem. This result, proven in section 8, says that, for every class $\mathcal{C}$ of languages that is closed under $\leq_{\mathrm{pos-tt}}^{\mathrm{P,lin}}$-reductions, the p-measure of $\mathcal{C}$ is somewhat robust with respect to changes in the underlying probability measure. Specifically, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive P-sequences of biases and $\mathcal{C}$ is a class of languages that is closed under $\leq_{\mathrm{pos-tt}}^{\mathrm{P,lin}}$-reductions, then the bias equivalence theorem says that

$$\mu_{\mathrm{p}}^{\vec{\alpha}}(\mathcal{C}) = 0 \Longleftrightarrow \mu_{\mathrm{p}}^{\vec{\beta}}(\mathcal{C}) = 0.$$

To put the matter differently, for every strongly positive P-sequence $\vec{\beta}$ of biases and for every class $\mathcal{C}$ that is closed under $\leq_{\mathrm{pos-tt}}^{\mathrm{P,lin}}$-reductions,

$$\mu_{\mathrm{p}}^{\vec{\beta}}(\mathcal{C}) = 0 \Longleftrightarrow \mu_{\mathrm{p}}(\mathcal{C}) = 0.$$

This result implies that most applications of resource-bounded measure to date can be immediately generalized from the uniform probability measure (in which they were developed) to arbitrary coin-toss probability measures given by strongly positive P-sequences of biases.

The bias equivalence theorem also offers the following new technique for proving resource-bounded measure results. If $\mathcal{C}$ is a class that is closed under $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reductions, then in order to prove that $\mu_{\mathrm{p}}(\mathcal{C}) = 0$, it suffices to prove that $\mu^{\vec{\beta}}_{\mathrm{p}}(\mathcal{C}) = 0$ for some conveniently chosen strongly positive P-sequence $\vec{\beta}$ of biases. (The bias equivalence theorem has already been put to this use in the forthcoming paper [16].)

The plausibility and consequences of the hypothesis $\mu_{\mathrm{p}}(\mathrm{NP}) \neq 0$ are subjects of recent and ongoing research [18, 15, 7, 17, 14, 3, 16]. The bias equivalence theorem immediately implies that the following three statements are equivalent.

(H1) $\mu_{\mathrm{p}}(\mathrm{NP}) \neq 0$.

(H2) For every strongly positive P-sequence $\vec{\beta}$ of biases, $\mu^{\vec{\beta}}_{\mathrm{p}}(\mathrm{NP}) \neq 0$.

(H3) There exists a strongly positive P-sequence $\vec{\beta}$ of biases such that $\mu^{\vec{\beta}}_{\mathrm{p}}(\mathrm{NP}) \neq 0$.

The statements (H2) and (H3) are thus new, equivalent formulations of the hypothesis (H1).

The proof of the bias equivalence theorem uses three main tools. The first is the summable equivalence theorem, which we have already discussed. The second is the martingale dilation theorem, which is proven in section 6. This result concerns martingales (defined in section 3), which are the betting algorithms on which resource-bounded measure is based. Roughly speaking, the martingale dilation theorem gives a method of transforming ("dilating") a martingale for one coin-toss probability measure into a martingale for another, perhaps very different, coin-toss probability measure, provided that the former measure is obtained from the latter via an "orderly" truth-table reduction.

The third tool used in the proof of our main theorem is the positive bias reduction theorem, which is presented in section 7. If $\vec{\alpha}$ and $\vec{\beta}$ are two strongly positive sequences of biases that are exactly P-computable (with no approximation), then the *positive bias reduction* of $\vec{\alpha}$ to $\vec{\beta}$ is a truth-table reduction (in fact, an orderly $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reduction) that uses the sequence $\vec{\beta}$ to "approximately simulate" the sequence $\vec{\alpha}$. It is especially crucial for our main result that this reduction is efficient and positive. (The circuits constructed by the truth-table reduction contain AND gates and OR gates, but no NOT gates.)

The summable equivalence theorem, the martingale dilation theorem, and the positive bias reduction theorem are only developed and used here as tools to prove our main result. Nevertheless, these three results are of independent interest, and are likely to be useful in future investigations.

2. Preliminaries.

In this paper, $\mathbb{N}$ denotes the set of all nonnegative integers, $\mathbb{Z}$ denotes the set of all integers, $\mathbb{Z}^+$ denotes the set of all positive integers, $\mathbb{Q}$ denotes the set of all rational numbers, and $\mathbb{R}$ denotes the set of all real numbers.

We write $\{0,1\}^*$ for the set of all (finite, binary) *strings*, and we write $|x|$ for the length of a string x. The empty string, λ, is the unique string of length 0. The *standard enumeration* of $\{0,1\}^*$ is the sequence $s_0 = \lambda, s_1 = 0, s_2 = 1, s_3 = 00, \ldots,$ ordered first by length and then lexicographically. For $x, y \in \{0,1\}^*$, we write $x < y$ if x precedes y in this standard enumeration. For $n \in \mathbb{N}$, $\{0,1\}^n$ denotes the set of all strings of length n, and $\{0,1\}^{\leq n}$ denotes the set of all strings of length at most n.

If x is a string or an (infinite, binary) *sequence*, and if $0 \leq i \leq j < |x|$, then $x[i..j]$ is the string consisting of the ith through jth bits of x. In particular, $x[0..i-1]$ is the *i-bit prefix* of x. We write $x[i]$ for $x[i..i]$, the ith bit of x. (Note that the leftmost bit of x is $x[0]$, the 0th bit of x.)

If w is a string and x is a string or sequence, then we write $w \sqsubseteq x$ if w is a prefix of x, i.e., if there is a string or sequence y such that $x = wy$.

The *Boolean value* of a condition ϕ is $[\![\phi]\!] = \textbf{if } \phi \textbf{ then } 1 \textbf{ else } 0$.

In this paper we use both the binary logarithm $\log \alpha = \log_2 \alpha$ and the natural logarithm $\ln \alpha = \log_e \alpha$.

Many of the functions in this paper are real-valued functions on discrete domains. These typically have the form

$$f : \mathbb{N}^m \times \{0,1\}^* \longrightarrow \mathbb{R}, \tag{2.1}$$

where $m \in \mathbb{N}$. (If $m = 0$, we interpret this to mean that $f : \{0,1\}^* \longrightarrow \mathbb{R}$.) Such a function f is defined to be p-*computable* if there is a function

$$\hat{f} : \mathbb{N} \times \mathbb{N}^m \times \{0,1\}^* \longrightarrow \mathbb{Q} \tag{2.2}$$

with the following two properties.

(i) For all $r, k_1, \ldots, k_m \in \mathbb{N}$ and $w \in \{0,1\}^*$,

$$|\hat{f}(r, k_1, \ldots, k_m, w) - f(k_1, \ldots, k_m, w)| \leq 2^{-r}.$$

(ii) There is an algorithm that, on input $(r, k_1, \ldots, k_m, w)$, computes the value $\hat{f}(r, k_1, \ldots, k_m, w)$ in $(r + k_1 + \cdots + k_m + |w|)^{O(1)}$ time.

Similarly, f is defined to be p_2-*computable* if there is a function $\hat{f}$ as in (2.2) that satisfies condition (i) above and the following condition.

(ii′) There is an algorithm that, on input $(r, k_1, \ldots, k_m, w)$, computes the value $\hat{f}(r, k_1, \ldots, k_m, w)$ in $2^{\log(r+k_1+\cdots+k_m+|w|)^{O(1)}}$ time.

In this paper, functions of the form (2.1) always have the form

$$f : \mathbb{N}^m \times \{0,1\}^* \longrightarrow [0, \infty)$$

or the form

$$f : \mathbb{N}^m \times \{0,1\}^* \longrightarrow [0, 1].$$

If such a function is p-computable or p_2-computable, then we assume without loss of generality that the approximating function $\hat{f}$ of (2.2) actually has the form

$$\hat{f} : \mathbb{N} \times \mathbb{N}^m \times \{0,1\}^* \longrightarrow \mathbb{Q} \cap [0, \infty)$$

or the form

$$\hat{f} : \mathbb{N} \times \mathbb{N}^m \times \{0,1\}^* \longrightarrow \mathbb{Q} \cap [0, 1],$$

respectively.

3. Resource-bounded ν-measure. In this section, we develop basic elements of resource-bounded measure based on an arbitrary (Borel) probability measure ν. The ideas here generalize the corresponding ideas of "ordinary" resource-bounded measure (based on the uniform probability measure μ) in a straightforward and natural way, so our presentation is relatively brief. The reader is referred to [11, 12] for additional discussion.

We work in the *Cantor space* **C**, consisting of all languages $A \subseteq \{0,1\}^*$. We identify each language A with its *characteristic sequence*, which is the infinite binary sequence χ_A defined by

$$\chi_A[n] = [\![s_n \in A]\!]$$

for each $n \in \mathbb{N}$. Relying on this identification, we also consider **C** to be the set of all infinite binary sequences.

For each string $w \in \{0,1\}^*$, the *cylinder generated by* w is the set

$$\mathbf{C}_w = \{A \in \mathbf{C} \mid w \sqsubseteq \chi_A\}.$$

Note that $\mathbf{C}_\lambda = \mathbf{C}$.

We first review the well-known notion of a (Borel) probability measure on **C**.

Definition 3.1. *A* probability measure *on* **C** *is a function*

$$\nu : \{0,1\}^* \longrightarrow [0,1]$$

such that $\nu(\lambda) = 1$, and for all $w \in \{0,1\}^*$,

$$\nu(w) = \nu(w0) + \nu(w1).$$

Intuitively, $\nu(w)$ is the probability that $A \in \mathbf{C}_w$ when we "choose a language $A \in \mathbf{C}$ according to the probability measure ν." We sometimes write $\nu(\mathbf{C}_w)$ for $\nu(w)$.

Examples 3.2.

1. *The* uniform probability measure μ *is defined by*

$$\mu(w) = 2^{-|w|}$$

for all $w \in \{0,1\}^*$.

2. *A* sequence of biases *is a sequence* $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$, *where each* $\beta_i \in [0,1]$. *Given a sequence of biases* $\vec{\beta}$, *the* $\vec{\beta}$*-coin-toss probability measure (also called the* $\vec{\beta}$-product probability measure*) is the probability measure* $\mu^{\vec{\beta}}$ *defined by*

$$\mu^{\vec{\beta}}(w) = \prod_{i=0}^{|w|-1} ((1-\beta_i)\cdot(1-w[i]) + \beta_i \cdot w[i])$$

for all $w \in \{0,1\}^*$.

3. *If* $\beta = \beta_0 = \beta_1 = \beta_2 = \cdots$, *then we write* μ^β *for* $\mu^{\vec{\beta}}$. *In this case, we have the simpler formula*

$$\mu^\beta(w) = (1-\beta)^{\#(0,w)} \cdot \beta^{\#(1,w)},$$

where $\#(b,w)$ *denotes the number of* b*'s in* w. *Note that* $\mu^{\frac{1}{2}} = \mu$.

Intuitively, $\mu^{\vec{\beta}}(w)$ is the probability that $w \sqsubseteq A$ when the language $A \subseteq \{0,1\}^*$ is chosen probabilistically according to the following random experiment. For each string s_i in the standard enumeration $s_0, s_1, s_2, \dots$ of $\{0,1\}^*$, we (independently of all other strings) toss a special coin, whose probability is β_i of coming up heads, in which case $s_i \in A$, and $1-\beta_i$ of coming up tails, in which case $s_i \notin A$.

Definition 3.3. *A probability measure* ν *on* **C** *is* positive *if, for all* $w \in \{0,1\}^*$, $\nu(w) > 0$.

Definition 3.4. *If ν is a positive probability measure and $u, v \in \{0,1\}^*$, then the* conditional ν-measure of u given v *is*

$$\nu(u|v) = \begin{cases} 1 & \text{if } u \sqsubseteq v, \\ \frac{\nu(u)}{\nu(v)} & \text{if } v \sqsubseteq u, \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\nu(u|v)$ is the conditional probability that $A \in \mathbf{C}_u$, given that $A \in \mathbf{C}_v$, when $A \in \mathbf{C}$ is chosen according to the probability measure ν.

Most of this paper concerns the following special type of probability measure.

Definition 3.5. *A probability measure ν on $\mathbf{C}$ is* strongly positive *if* (ν *is positive and*) *there is a constant $\delta > 0$ such that, for all $w \in \{0,1\}^*$ and $b \in \{0,1\}$, $\nu(wb|w) \geq \delta$.*

Definition 3.6. *A sequence of biases $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ is* strongly positive *if there is a constant $\delta > 0$ such that, for all $i \in \mathbb{N}$, $\beta_i \in [\delta, 1-\delta]$.*

If $\vec{\beta}$ is a sequence of biases, then the following two observations are clear.

1. $\mu^{\vec{\beta}}$ is positive if and only if $\beta_i \in (0,1)$ for all $i \in \mathbb{N}$.
2. If $\mu^{\vec{\beta}}$ is positive, then for each $w \in \{0,1\}^*$,

$$\mu^{\vec{\beta}}(w0|w) = 1 - \beta_{|w|}$$

and

$$\mu^{\vec{\beta}}(w1|w) = \beta_{|w|}.$$

It follows immediately from these two things that the probability measure $\mu^{\vec{\beta}}$ is strongly positive if and only if the sequence of biases $\vec{\beta}$ is strongly positive.

In this paper, we are primarily interested in strongly positive probability measures ν that are p-computable in the sense defined in section 2.

We next review the well-known notion of a martingale over a probability measure ν. Computable martingales were used by Schnorr [23, 24, 25, 26] in his investigations of randomness, and have more recently been used by Lutz [11] in the development of resource-bounded measure.

Definition 3.7. *Let ν be a probability measure on $\mathbf{C}$. Then a ν-*martingale *is a function $d : \{0,1\}^* \longrightarrow [0, \infty)$ such that, for all $w \in \{0,1\}^*$,*

$$d(w)\nu(w) = d(w0)\nu(w0) + d(w1)\nu(w1). \tag{3.1}$$

*If $\vec{\beta}$ is a sequence of biases, then a $\mu^{\vec{\beta}}$-martingale is simply called a $\vec{\beta}$-*martingale*. A μ-martingale is even more simply called a* martingale. (*That is, when the probability measure is not specified, it is assumed to be the uniform probability measure μ.*)

Intuitively, a ν-martingale d is a "strategy for betting" on the successive bits of (the characteristic sequence of) a language $A \in \mathbf{C}$. The real number $d(\lambda)$ is regarded as the amount of money that the strategy starts with. The real number $d(w)$ is the amount of money that the strategy has after betting on a prefix w of χ_A. The identity (3.1) ensures that the betting is "fair" in the sense that, if A is chosen according to the probability measure ν, then the expected amount of money is constant as the betting proceeds. (See [23, 24, 25, 26, 27, 11, 13, 12] for further discussion.) Of course, the "objective" of a strategy is to win a lot of money.

DEFINITION 3.8. *A ν-martingale d* succeeds *on a language $A \in \mathbf{C}$ if*

$$\limsup_{n \longrightarrow \infty} d(\chi_A[0..n-1]) = \infty.$$

The success set *of a ν-martingale d is the set*

$$S^\infty[d] = \{A \in \mathbf{C} \mid d \textit{ succeeds on } A\}.$$

We are especially interested in martingales that are computable within some resource bound. (Recall that the p-computability and p_2-computability of real valued functions were defined in section 2.)

DEFINITION 3.9. *Let ν be a probability measure on $\mathbf{C}$.*

1. *A p-ν-*martingale *is a ν-martingale that is p-computable.*
2. *A p_2-ν-*martingale *is a ν-martingale that is p_2-computable.*

A p-$\mu^{\vec{\beta}}$-martingale is called a p-$\vec{\beta}$-martingale, a p-μ-martingale is called a p-martingale, and similarly for p_2.

We now come to the fundamental ideas of resource-bounded ν-measure.

DEFINITION 3.10. *Let ν be a probability measure on $\mathbf{C}$, and let $X \subseteq \mathbf{C}$.*

1. *X has p-ν-*measure 0*, and we write $\nu_\mathrm{p}(X) = 0$, if there is a p-ν-martingale d such that $X \subseteq S^\infty[d]$.*
2. *X has p-ν-*measure 1*, and we write $\nu_\mathrm{p}(X) = 1$, if $\nu_\mathrm{p}(X^c) = 0$, where $X^c = \mathbf{C} - X$.*

The conditions $\nu_{\mathrm{p}_2}(X) = 0$ and $\nu_{\mathrm{p}_2}(X) = 1$ are defined analogously.

DEFINITION 3.11. *Let ν be a probability measure on $\mathbf{C}$, and let $X \subseteq \mathbf{C}$.*

1. *X has ν-*measure 0 in *E, and we write $\nu(X|\mathrm{E}) = 0$, if $\nu_\mathrm{p}(X \cap E) = 0$.*
2. *X has ν-*measure 1 in *E, and we write $\nu(X|\mathrm{E}) = 1$, if $\nu(X^c|\mathrm{E}) = 0$.*
3. *X has ν-*measure 0 in E_2*, and we write $\nu(X|\mathrm{E}_2) = 0$, if $\nu_{\mathrm{p}_2}(X \cap \mathrm{E}_2) = 0$.*
4. *X has ν-*measure 1 in E_2*, and we write $\nu(X|\mathrm{E}_2) = 1$, if $\nu(X^c|\mathrm{E}_2) = 0$.*

Just as in the uniform case [11], the resource bounds p and p_2 of the above definitions are only two possible values of a very general parameter. Other choices of this parameter yield classical ν-measure [5], constructive ν-measure (as used in algorithmic information theory [29, 27]), ν-measure in the set REC, consisting of all decidable languages, ν-measure in ESPACE, etc.

The rest of this section is devoted to a very brief presentation of some of the fundamental theorems of resource-bounded ν-measure. One of the main objectives of these results is to justify the intuition that *a set with ν-measure* 0 *in* E *contains only a "negligibly small" part of* E (with respect to ν). For the purpose of this paper, it suffices to present these results for p-ν-measure and ν-measure in E. We note, however, that all these results hold *a fortiori* for p_2-ν-measure, rec-ν-measure, classical ν-measure, ν-measure in E_2, ν-measure in ESPACE, etc.

We first note that ν-measure 0 sets exhibit the set-theoretic behavior of small sets.

DEFINITION 3.12. *Let $X, X_0, X_1, X_2, \ldots \subseteq \mathbf{C}$.*

1. *X is a p-*union of the p-ν-measure 0 sets $X_0, X_1, X_2, \ldots$ *if $X = \cup_{k=0}^\infty X_k$ and there is a sequence $d_0, d_1, d_2, \ldots$ of ν-martingales with the following two properties.*
   (i) *For each $k \in \mathbb{N}$, $X_k \subseteq S^\infty[d_k]$.*
   (ii) *The function $(k, w) \mapsto d_k(w)$ is p-computable.*

2. *X is a* p-union of the sets $X_0, X_1, X_2, \ldots$ of ν-measure 0 in E *if* $X = \cup_{k=0}^{\infty} X_k$ *and there is a sequence* $d_0, d_1, d_2, \ldots$ *of* ν*-martingales with the following two properties.*
(i) *For each* $k \in \mathbb{N}$, $X_k \cap \mathrm{E} \subseteq S^\infty[d_k]$.
(ii) *The function* $(k, w) \mapsto d_k(w)$ *is p-computable.*

LEMMA 3.1. *Let* ν *be a probability measure on* $\mathbf{C}$, *and let* $\mathcal{I}$ *be either the collection of all p-*ν*-measure* 0 *subsets of* $\mathbf{C}$, *or the collection of all subsets of* $\mathbf{C}$ *that have* ν*-measure* 0 *in E. Then* $\mathcal{I}$ *has the following three closure properties.*
1. *If* $X \subseteq Y \in \mathcal{I}$, *then* $X \in \mathcal{I}$.
2. *If* X *is a finite union of elements of* $\mathcal{I}$, *then* $X \in \mathcal{I}$
3. *If* X *is a p-union of elements of* $\mathcal{I}$, *then* $X \in \mathcal{I}$.

Proof (sketch). Assume that X is a p-union of the p-ν-measure 0 sets $X_0, X_1, X_2, \ldots$, and let $d_0, d_1, d_2, \ldots$ be as in the definition of this condition. It suffices to show that $\nu_{\mathrm{p}}(X) = 0$. (The remaining parts of the lemma are obvious or follow directly from this.) The p-computability of the function $(k, w) \mapsto d_k(w)$ implies that there is a nondecreasing polynomial q such that, for all $k \in \mathbb{N}$ and $w \in \{0,1\}^*$, $d_k(w) \leq 2^{q(k+|w|)}$. Define

$$d : \{0,1\}^* \longrightarrow [0, \infty)$$
$$d(w) = \sum_{k=0}^{\infty} 2^{-q(2k)-k} d_k(w).$$

It is easily checked that d is a p-ν-martingale and that $X \subseteq S^\infty[d]$, so $\nu_{\mathrm{p}}(X) = 0$. □

We next note that, if ν is strongly positive and p-computable, then every singleton subset of E has p-ν-measure 0.

LEMMA 3.2. *If* ν *is a strongly positive*, *p-computable probability measure on* $\mathbf{C}$, *then for every* $A \in \mathrm{E}$,

$$\nu_{\mathrm{p}}(\{A\}) = \nu(\{A\}|\mathrm{E}) = 0.$$

Proof (sketch). Assume the hypothesis, and fix $\delta > 0$ such that, for all $w \in \{0,1\}^*$ and $b \in \{0,1\}$, $\nu(wb|w) \geq \delta$. Define

$$d : \{0,1\}^* \longrightarrow [0, \infty)$$
$$d(\lambda) = 1,$$
$$d(wb) = \frac{d(w)}{\nu(wb|w)} \cdot [\![b = [\![s_{|w|} \in A]\!]]\!].$$

It is easily checked that d is a p-ν-martingale and that, for all $n \in \mathbb{N}$, $d(\chi_A[0..n-1]) \geq (1-\delta)^{-n}$, whence $A \in S^\infty[d]$. □

Note that, for $A \in \mathrm{E}$, the "point-mass" probability measure

$$\pi_A : \{0,1\}^* \longrightarrow [0,1]$$
$$\pi_A(w) = \begin{cases} 1 & \text{if } w \sqsubseteq \chi_A, \\ 0 & \text{if } w \not\sqsubseteq \chi_A \end{cases}$$

is p-computable, and $\{A\}$ does *not* have p-π_A-measure 0. Thus, the strong positivity hypothesis cannot be removed from Lemma 3.2.

We now come to the most crucial issue in the development of resource-bounded measure. If a set X has ν-measure 0 in E, then we want to say that X contains only a "negligible small" part of E. In particular, then, it is critical that E itself not have ν-measure 0 in E. The following theorem establishes this and more.

THEOREM 3.3. *Let* ν *be a probability measure on* $\mathbf{C}$, *and let* $w \in \{0,1\}^*$. *If* $\nu(w) > 0$, *then* $\mathbf{C}_w$ *does not have* ν*-measure* 0 *in E.*

Proof (sketch). Assume the hypothesis, and let d be a p-ν-martingale. It suffices to show that $\mathbf{C}_w \cap \mathrm{E} \not\subseteq S^\infty[d]$.

Since d is p-computable, there is a function $\hat{d} : \mathbb{N} \times \{0,1\}^* \longrightarrow \mathbb{Q} \cap [0,\infty)$ with the following two properties.

(i) For all $r \in \mathbb{N}$ and $w \in \{0,1\}^*$, $|\hat{d}(r,w) - d(w)| \leq 2^{-r}$.

(ii) There is an algorithm that computes $\hat{d}(r,w)$ in time polynomial in $r + |w|$.

Define a language A recursively as follows. First, for $0 \leq i < |w|$, $[\![s_i \in A]\!] = w[i]$. Next assume that the string $x_i = \chi_A[0..i-1]$ has been defined, where $i \geq |w|$. Then

$$[\![s_i \in A]\!] = [\![\hat{d}(i+1, x_i1) \leq \hat{d}(i+1, x_i0)]\!].$$

With the language A so defined, it is easy to check that $A \in \mathbf{C}_w \cap \mathrm{E}$. It is also routine to check that, for all $i \geq |w|$,

$$\begin{aligned} d(x_{i+1}) &\leq \hat{d}(i+1, x_{i+1}) + 2^{-(i+1)} \\ &= \min\left\{\hat{d}(i+1, x_i0), \hat{d}(i+1, x_i1)\right\} + 2^{-(i+1)} \\ &\leq \min\{d(x_i0), d(x_i1)\} + 2^{-i} \\ &\leq d(x_i) + 2^{-i}. \end{aligned}$$

It follows inductively that, for all $n \geq |w|$,

$$\begin{aligned} d(x_n) &\leq d(w) + \sum_{i=|w|}^{n-1} 2^{-i} \\ &< d(w) + \sum_{i=|w|}^{\infty} 2^{-i} = d(w) + 2^{1-|w|}. \end{aligned}$$

This implies that

$$\limsup_{n \longrightarrow \infty} d(\chi_A[0..n-1]) \leq d(w) + 2^{1-|w|} < \infty,$$

whence $A \notin S^\infty[d]$. $\square$

As in the case of the uniform probability measure [11], more quantitative results on resource-bounded ν-measure can be obtained by considering the *unitary success set*

$$S^1[d] = \bigcup_{\substack{w \\ d(w) \geq 1}} \mathbf{C}_w$$

and the *initial value* $d(\lambda)$ of a p-ν-martingale d. For example, generalizing the arguments in [11] in a straightforward manner, this approach yields a measure conservation theorem for ν-measure (a quantitative extension of Theorem 3.3) and a uniform, resource-bounded extension of the classical first Borel–Cantelli lemma. As these results are not used in the present paper, we refrain from elaborating here.

4. Summable equivalence. If two probability measures on $\mathbf{C}$ are sufficiently "close" to one another, then the summable equivalence theorem says that the two probability measures are in absolute agreement as to which sets of languages have p-measure 0 and which do not. In this section, we define this notion of "close" and prove this result.

DEFINITION 4.1. *Let ν be a positive probability measure on $\mathbf{C}$, let $A \subseteq \{0,1\}^*$, and let $i \in \mathbb{N}$. Then the i*th conditional ν-probability along A *is*

$$\nu_A(i+1|i) = \nu(\chi_A[0..i] \mid \chi_A[0..i-1]).$$

DEFINITION 4.2. *Two positive probability measures ν and ν' on $\mathbf{C}$ are* summably equivalent, *and we write $\nu \approx \nu'$, if for every $A \subseteq \{0,1\}^*$,*

$$\sum_{i=0}^{\infty} |\nu_A(i+1|i) - \nu'_A(i+1|i)| < \infty.$$

It is clear that summable equivalence is an equivalence relation on the collection of all positive probability measures on $\mathbf{C}$. The following fact is also easily verified.

LEMMA 4.1. *Let ν and ν' be positive probability measures on $\mathbf{C}$. If $\nu \approx \nu'$, then ν is strongly positive if and only if ν' is strongly positive.*

The following definition gives the most obvious way to transform a martingale for one probability measure into a martingale for another.

DEFINITION 4.3. *Let ν and ν' be probability measures on $\mathbf{C}$ with ν' positive, and let d be a ν-martingale. Then the* canonical adjustment *of d to ν' is the ν'-martingale d' defined by*

$$d'(w) = \frac{\nu(w)}{\nu'(w)} d(w)$$

for all $w \in \{0,1\}^*$.

It is trivial to check that the above function d' is indeed a ν'-martingale. The following lemma shows that, for strongly positive probability measures, summable equivalence is a sufficient condition for d' to succeed whenever d succeeds.

LEMMA 4.2. *Let ν and ν' be strongly positive probability measures on $\mathbf{C}$, let d be a ν-martingale, and let d' be the canonical adjustment of d to ν'. If $\nu \approx \nu'$, then $S^\infty[d] \subseteq S^\infty[d']$.*

Proof. Assume the hypothesis, and let $A \in S^\infty[d]$. For each $i \in \mathbb{N}$, let

$$\nu_i = \nu_A(i+1|i), \quad \nu'_i = \nu'_A(i+1|i), \quad \tau_i = \nu_i - \nu'_i.$$

The hypothesis $\nu \approx \nu'$ says that $\sum_{i=0}^{\infty} |\tau_i| < \infty$. In particular, this implies that $\tau_i \longrightarrow 0$ as $i \longrightarrow \infty$, so we have the Taylor approximation

$$\ln \frac{\nu_i}{\nu'_i} = \ln\left(1 + \frac{\tau_i}{\nu'_i}\right) = \frac{\tau_i}{\nu'_i} + \mathrm{o}\left(\frac{\tau_i}{\nu'_i}\right)$$

as $i \longrightarrow \infty$. Thus $|\ln \frac{\nu_i}{\nu'_i}|$ is asymptotically equivalent to $\frac{|\tau_i|}{\nu'_i}$ as $i \longrightarrow \infty$. Since ν' is strongly positive, it follows that $\sum_{i=0}^{\infty} |\ln \frac{\nu_i}{\nu'_i}| < \infty$. Thus, if we let $w_k = \chi_A[0..k-1]$, then there is a positive constant c such that, for all $k \in \mathbb{N}$,

$$c \geq \sum_{i=0}^{k-1} \left(-\ln \frac{\nu_i}{\nu'_i}\right) = -\ln \prod_{i=0}^{k-1} \frac{\nu_i}{\nu'_i} = -\ln \frac{\nu(w_k)}{\nu'(w_k)},$$

whence

$$d'(w_k) = \frac{\nu(w_k)}{\nu'(w_k)} d(w_k) \geq e^{-c} d(w_k).$$

Since $A \in S^\infty[d]$, we thus have

$$\limsup_{k \longrightarrow \infty} d'(w_k) \geq \limsup_{k \longrightarrow \infty} e^{-c} d(w_k) = \infty,$$

so $A \in S^\infty[d']$. □

The following useful result is now easily established.

THEOREM 4.3 (summable equivalence theorem). *If ν and ν' are strongly positive, p-computable probability measures on* $\mathbf{C}$ *such that $\nu \approx \nu'$, then for every set $X \subseteq \mathbf{C}$,*

$$\nu_{\mathrm{p}}(X) = 0 \Longleftrightarrow \nu'_{\mathrm{p}}(X) = 0.$$

Proof. Assume the hypothesis, and assume that $\nu_{\mathrm{p}}(X) = 0$. By symmetry, it suffices to show that $\nu'_{\mathrm{p}}(X) = 0$. Since $\nu_{\mathrm{p}}(X) = 0$, there is a p-computable ν-martingale d such that $X \subseteq S^\infty[d]$. Let d' be the canonical adjustment of d to ν'. Since $d, \nu,$ and ν' are all p-computable, it is easy to see that d' is p-computable. Since $\nu \approx \nu'$, Lemma 4.2 tells us that

$$X \subseteq S^\infty[d] \subseteq S^\infty[d'].$$

Thus $\nu'_{\mathrm{p}}(X) = 0$. □

5. Exact computation. It is sometimes useful or convenient to work with probability measures that are rational-valued and efficiently computable in an exact sense, with no approximation. This section presents two very easy results identifying situations in which such probability measures are available.

DEFINITION 5.1. *A probability measure ν on* $\mathbf{C}$ *is* exactly p-computable *if $\nu : \{0,1\}^* \longrightarrow \mathbb{Q} \cap [0,1]$ and there is an algorithm that computes $\nu(w)$ in time polynomial in $|w|$.*

LEMMA 5.1. *For every strongly positive, p-computable probability measure ν on* $\mathbf{C}$, *there is an exactly p-computable probability measure ν' on* $\mathbf{C}$ *such that $\nu \approx \nu'$.*

Proof. Let ν be a p-computable probability measure on $\mathbf{C}$, and fix a function $\hat{\nu} : \mathbb{N} \times \{0,1\}^* \longrightarrow \mathbb{Q} \cap [0,1]$ that testifies to the p-computability of ν. Since ν is strongly positive, there is a constant $c \in \mathbb{N}$ such that, for all $w \in \{0,1\}^*$, $2^{-c|w|} \leq \nu(w) \leq 1 - 2^{-c|w|}$. Fix such a c and, for all $w \in \{0,1\}^*$, define

$$\nu'(w0|w) = \min\left\{1, \frac{\hat{\nu}((2c+1)|w|+3, w0)}{\hat{\nu}((2c+1)|w|+3, w)}\right\},$$

$$\nu'(w1|w) = 1 - \nu'(w0|w),$$

$$\nu'(w) = \prod_{i=0}^{|w|-1} \nu'(w[0..i] \big| w[0..i-1]).$$

It is clear that ν' is an exactly p-computable probability measure on $\mathbf{C}$.

Now let $w \in \{0,1\}^*$ and $b \in \{0,1\}$. For convenience, let

$$\begin{aligned} \delta &= 2^{-(1+c|w|)}, \\ \epsilon &= 2^{-(2c+1)|w|-3}, \\ a_1 &= \nu(wb), \\ a_2 &= \nu(w). \end{aligned}$$

Note that

$$\hat{\nu}((2c+1)|w|+3, w) \geq \nu(w) - \epsilon > \nu(w) - \delta \geq \delta.$$

It is clear by inspection that $\nu'(wb|w)$ can be written in the form

$$\nu'(wb|w) = \frac{a_1'}{a_2'},$$

where

$$|a_1' - a_1| \leq \epsilon \text{ and } |a_2' - a_2| \leq \epsilon.$$

We thus have

$$\begin{aligned} |a_1'a_2 - a_1a_2'| &\leq |a_1'a_2 - a_1a_2| + |a_1a_2 - a_1a_2'| \\ &\leq |a_1' - a_1| + |a_2' - a_2| \\ &\leq 2\epsilon, \end{aligned}$$

whence

$$\begin{aligned} |\nu'(wb|w) - \nu(wb|w)| &= \left| \frac{a_1'}{a_2'} - \frac{a_1}{a_2} \right| \\ &= \frac{|a_1'a_2 - a_1a_2'|}{a_2a_2'} \\ &\leq 2\epsilon\delta^{-2} \\ &= 2^{-|w|}. \end{aligned}$$

For all $A \subseteq \{0,1\}^*$, then, we have

$$\sum_{i=0}^{\infty} \left|\nu_A(i+1\middle|i) - \nu_A'(i+1\middle|i)\right| \;\leq\; \sum_{i=0}^{\infty} 2^{-i} \;=\; 2,$$

so $\nu \approx \nu'$. □

For some purposes (including those of this paper), the p-computability requirement is too weak, because it allows $\nu(w)$ to be computed (or approximated) in time polynomial in $|w|$, which is exponential in the length of the last string decided by w when we regard w as a prefix of a language A. In such situations, the following sort of requirement is often more useful. (We give only the definitions for sequences of biases, i.e., coin-toss probability measures, because this suffices for our purposes in this paper. It is clearly a routine matter to generalize further.)

DEFINITION 5.2.

1. *A P*-sequence of biases *is a sequence $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ of biases $\beta_i \in [0,1]$ for which there is a function*

$$\hat{\beta} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{Q} \cap [0,1]$$

   with the following two properties.
   (i) *For all $i, r \in \mathbb{N}$, $|\hat{\beta}(i,r) - \beta_i| \leq 2^{-r}$.*
   (ii) *There is an algorithm that, for all $i, r \in \mathbb{N}$, computes $\hat{\beta}(i,r)$ in time polynomial in $|s_i| + r$ (i.e., in time polynomial in $\log(i+1) + r$).*
2. *A P*-exact sequence of biases *is a sequence $\vec{\beta} = (\beta_0, \beta_1, \beta_2, \dots)$ of (rational) biases $\beta_i \in \mathbb{Q} \cap [0,1]$ such that the function $i \longmapsto \beta_i$ is computable in time polynomial in $|s_i|$.*

DEFINITION 5.3. *If $\vec{\alpha}$ and $\vec{\beta}$ are sequences of biases, then $\vec{\alpha}$ and $\vec{\beta}$ are* summably equivalent, *and we write $\vec{\alpha} \approx \vec{\beta}$, if $\sum_{i=0}^{\infty} |\alpha_i - \beta_i| < \infty$.*

It is clear that $\vec{\alpha} \approx \vec{\beta}$ if and only if $\mu^{\vec{\alpha}} \approx \mu^{\vec{\beta}}$.

LEMMA 5.2. *For every P-sequence of biases $\vec{\beta}$, there is a P-exact sequence of biases $\vec{\beta}'$ such that $\vec{\beta} \approx \vec{\beta}'$.*

Proof. Let $\vec{\beta}$ be a strongly positive P-sequence of biases, and let $\hat{\beta} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{Q} \cap [0,1]$ be a function that testifies to this fact. For each $i \in \mathbb{N}$, let

$$\beta'_i = \hat{\beta}(i, 2|s_i|),$$

and let $\vec{\beta}' = (\beta'_0, \beta'_1, \beta'_2, \dots)$. Then $\vec{\beta}'$ is a P-exact sequence of biases, and

$$\begin{aligned}\sum_{i=0}^{\infty} |\beta_i - \beta'_i| &\leq \sum_{i=0}^{\infty} 2^{-2|s_i|} \\ &\leq \sum_{i=0}^{\infty} 2^{-2\log(i+1)} \\ &= \sum_{i=0}^{\infty} \frac{1}{(i+1)^2} < \infty,\end{aligned}$$

so $\vec{\beta} \approx \vec{\beta}'$. □

6. Martingale dilation. In this section we show that certain truth-table reductions can be used to *dilate* martingales for one probability measure into martingales for another, perhaps dissimilar, probability measure on **C**. We first present some terminology and notation on truth-table reductions. (Most of this notation is standard [22], but some is specialized to our purposes.)

A truth-table reduction is a pair (f, g) of algorithms that, given a language A, is used to decide a language $B = F_{(f,g)}(A)$. Given a particular string x, the algorithm f specifies a list of membership queries regarding A, and the algorithm g specifies how the answers to these queries are to be used in deciding whether $x \in B$. The list of queries given by f is nonadaptive, i.e., the entire list is specified prior to the knowing of any of the answers to the queries. More formally, a *truth-table reduction* (briefly, a $\leq_{\text{tt}}$-reduction) is an ordered pair (f, g) of total recursive functions such that for each $x \in \{0,1\}^*$, there exists $n(x) \in \mathbb{Z}^+$ such that the following two conditions hold.

(i) $f(x)$ is (the standard encoding of) an $n(x)$-tuple $(f_1(x), \dots, f_{n(x)}(x))$ of strings $f_i(x) \in \{0,1\}^*$, which are called the *queries* of the reduction (f, g)

on input x. We use the notation $Q_{(f,g)}(x) = \{f_1(x), \ldots, f_{n(x)}(x)\}$ for the set of such queries.

(ii) $g(x)$ is (the standard encoding of) an $n(x)$-input, 1-output Boolean circuit, called the *truth table* of the reduction (f, g) on input x. We identify $g(x)$ with the Boolean function computed by this circuit, i.e.,

$$g(x) : \{0,1\}^{n(x)} \longrightarrow \{0,1\}.$$

A truth-table reduction (f, g) *induces* the function

$$F_{(f,g)} : \mathbf{C} \longrightarrow \mathbf{C}$$
$$F_{(f,g)}(A) = \left\{x \in \{0,1\}^* \ \mid g(x)\left([\![f_1(x) \in A]\!] \cdots [\![f_{n(x)}(x) \in A]\!]\right) = 1\right\}.$$

If A and B are languages and (f, g) is a $\leq_{\mathrm{tt}}$-reduction, then (f, g) *reduces B to A*, and we write

$$B \leq_{\mathrm{tt}} A \text{ via } (f, g),$$

if $B = F_{(f,g)}(A)$. More generally, if A and B are languages, then B is *truth-table reducible* (briefly, $\leq_{\mathrm{tt}}$-reducible) to A, and we write $B \leq_{\mathrm{tt}} A$, if there exists a $\leq_{\mathrm{tt}}$-reduction (f, g) such that $B \leq_{\mathrm{tt}} A$ via (f, g).

If (f, g) is a $\leq_{\mathrm{tt}}$-reduction, then the function $F_{(f,g)} : \mathbf{C} \longrightarrow \mathbf{C}$ defined above induces a corresponding function

$$F_{(f,g)} : \{0,1\}^* \longrightarrow \{0,1\}^* \cup \mathbf{C}$$

defined as follows. (It is standard practice to use the same notation for these two functions, and no confusion will result from this practice here.) Intuitively, if $A \in \mathbf{C}$ and $w \sqsubseteq A$, then $F_{(f,g)}(w)$ is the largest prefix of $F_{(f,g)}(A)$ such that w answers all queries in this prefix. Formally, let $w \in \{0,1\}^*$, and let

$$A_w = \left\{s_i \,\middle|\, 0 \leq i < |w| \text{ and } w[i] = 1\right\}.$$

If $Q_{(f,g)}(x) \subseteq \{s_0, \ldots s_{|w|-1}\}$ for all $x \in \{0,1\}^*$, then

$$F_{(f,g)}(w) = F_{(f,g)}(A_w).$$

Otherwise,

$$F_{(f,g)}(w) = \chi_{F_{(f,g)}(A_w)}[0..m-1],$$

where m is the greatest nonnegative integer such that

$$\bigcup_{i=0}^{m-1} Q_{(f,g)}(s_i) \subseteq \left\{s_0, \ldots, s_{|w|-1}\right\}.$$

Now let (f, g) be a $\leq_{\mathrm{tt}}$-reduction, and let $z \in \{0,1\}^*$. Then the *inverse image* of the cylinder $\mathbf{C}_z$ under the reduction (f, g) is

$$\begin{aligned} F^{-1}_{(f,g)}(\mathbf{C}_z) &= \left\{A \in \mathbf{C} \mid F_{(f,g)}(A) \in \mathbf{C}_z\right\} \\ &= \left\{A \in \mathbf{C} \mid z \sqsubseteq F_{(f,g)}(A)\right\}. \end{aligned}$$

We can write this set in the form

$$F^{-1}_{(f,g)}(\mathbf{C}_z) = \bigcup_{w \in I} \mathbf{C}_w,$$

where I is the set of all strings $w \in \{0,1\}^*$ with the following properties.

(i) $z \sqsubseteq F_{(f,g)}(w)$.

(ii) If w' is a proper prefix of w, then $z \not\sqsubseteq F_{(f,g)}(w')$.

Moreover, the cylinders $\mathbf{C}_w$ in this union are disjoint, so if ν is a probability measure on $\mathbf{C}$, then

$$\nu(F^{-1}_{(f,g)}(\mathbf{C}_z)) = \sum_{w \in I} \nu(w).$$

The following well-known fact is easily verified.

LEMMA 6.1. *If ν is a probability measure on $\mathbf{C}$ and (f,g) is a $\leq_{\mathrm{tt}}$-reduction, then the function*

$$\nu^{(f,g)} : \{0,1\}^* \longrightarrow [0,1],$$
$$\nu^{(f,g)}(z) = \nu(F^{-1}_{(f,g)}(\mathbf{C}_z))$$

is also a probability measure on $\mathbf{C}$.

The probability measure $\nu^{(f,g)}$ of Lemma 6.1 is called the *probability measure induced by ν and (f,g)*.

In this paper, we only use the following special type of $\leq_{\mathrm{tt}}$-reduction.

DEFINITION 6.1. *A $\leq_{\mathrm{tt}}$-reduction (f,g) is* orderly *if, for all $x,y,u,v \in \{0,1\}^*$, if $x < y$, $u \in Q_{(f,g)}(x)$, and $v \in Q_{(f,g)}(y)$, then $u < v$. That is, if x precedes y (in the standard ordering of $\{0,1\}^*$), then every query of (f,g) on input x precedes every query of (f,g) on input y.*

The following is an obvious property of orderly $\leq_{\mathrm{tt}}$-reductions.

LEMMA 6.2. *If ν is a coin-toss probability measure on $\mathbf{C}$ and (f,g) is an orderly $\leq_{\mathrm{tt}}$-reduction, then $\nu^{(f,g)}$ is also a coin-toss probability measure on $\mathbf{C}$.*

Note that, if (f,g) is an orderly $\leq_{\mathrm{tt}}$-reduction, then $F_{(f,g)}(w) \in \{0,1\}^*$ for all $w \in \{0,1\}^*$. Note also that the length of $F_{(f,g)}(w)$ depends only upon the length of w (i.e., $|w| = |w'|$ implies that $|F_{(f,g)}(w)| = |F_{(f,g)}(w')|$). Finally, note that for each $m \in \mathbb{N}$ there exists $l \in \mathbb{N}$ such that $|F_{(f,g)}(0^l)| = m$.

DEFINITION 6.2. *Let (f,g) be an orderly $\leq_{\mathrm{tt}}$-reduction.*

1. *An (f,g)-*step *is a positive integer l such that $F_{(f,g)}(0^{l-1}) \neq F_{(f,g)}(0^l)$.*
2. *For $k \in \mathbb{N}$, we let step(k) be the least (f,g)-step l such that $l \geq k$.*

The following construction is crucial to the proof of our main theorem.

DEFINITION 6.3. *Let ν be a positive probability measure on $\mathbf{C}$, let (f,g) be an orderly $\leq_{\mathrm{tt}}$-reduction, and let d be a $\nu^{(f,g)}$-martingale. Then the (f,g)-*dilation *of d is the function*

$$(f,g)\widehat{\ }d : \{0,1\}^* \longrightarrow [0,\infty)$$
$$(f,g)\widehat{\ }d(w) = \sum_{u \in \{0,1\}^{l-k}} d(F_{(f,g)}(wu))\nu(wu|w),$$

where $k = |w|$ and $l = $ step(k).

In other words, $(f,g)\widehat{\ }d(w)$ is the conditional ν-expected value of $d(F_{(f,g)}(w'))$, given that $w \sqsubseteq w'$ and $|w'| = $ *step*$(|w|)$. We do not include the probability measure ν in the notation $(f,g)\widehat{\ }d$ because ν (being positive) is implicit in d.

Intuitively, the function $(f,g)\widehat{\ }d$ is a strategy for betting on a language A, assuming that d itself is a strategy for betting on the language $F_{(f,g)}(A)$. The following theorem makes this intuition precise.

THEOREM 6.3 (martingale dilation theorem). *Assume that ν is a positive coin-toss probability measure on* $\mathbf{C}$, *(f,g) is an orderly $\leq_{\mathrm{tt}}$-reduction, and d is a $\nu^{(f,g)}$-martingale. Then $(f,g)\hat{}d$ is a ν-martingale. Moreover, for every language $A \subseteq \{0,1\}^*$, if d succeeds on $F_{(f,g)}(A)$, then $(f,g)\hat{}d$ succeeds on A.*

A very special case of the above result (for strictly increasing $\leq_{\mathrm{m}}^{\mathrm{P}}$-reductions under the uniform probability measure) was developed by Ambos-Spies, Terwijn, and Zheng [2] and made explicit by Juedes and Lutz [8]. Our use of martingale dilation in the present paper is very different from the simple padding arguments of [2, 8].

The following two technical lemmas are used in the proof of Theorem 6.3.

LEMMA 6.4. *Assume that ν is a positive coin-toss probability measure on* $\mathbf{C}$ *and (f,g) is an orderly $\leq_{\mathrm{tt}}$-reduction. Let $F = F_{(f,g)}$, let $w \in \{0,1\}^*$, and assume that $k = |w|$ is an (f,g)-step. Let $l = step(k+1)$. Then, for $b \in \{0,1\}$,*

$$\nu^{(f,g)}(F(w)b|F(w)) = \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(wu) = F(w)b}} \nu(wu|w).$$

Proof. Assume the hypothesis. Then

$$\begin{aligned}\nu^{(f,g)}(F(w)b) &= \sum_{\substack{w' \in \{0,1\}^k \\ F(w') = F(w)}} \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(w'u) = F(w')b}} \nu(w'u) \\ &= \sum_{\substack{w' \in \{0,1\}^k \\ F(w') = F(w)}} \nu(w') \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(w'u) = F(w')b}} \nu(w'u|w').\end{aligned}$$

Now, since ν is a coin-toss probability measure, we have $\nu(w'u|w') = \nu(wu|w)$ for each $w' \in \{0,1\}^k$ such that $F(w') = F(w)$. Also, since (f,g) is orderly, the conditions $F(w'u) = F(w')b$ and $F(wu) = F(w)b$ are equivalent for each $u \in \{0,1\}^{l-k}$. Hence,

$$\begin{aligned}\nu^{(f,g)}(F(w)b) &= \sum_{\substack{w' \in \{0,1\}^k \\ F(w') = F(w)}} \nu(w') \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(wu) = F(w)b}} \nu(wu|w) \\ &= \nu^{(f,g)}(F(w)) \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(wu) = F(w)b}} \nu(wu|w). \qquad \square\end{aligned}$$

LEMMA 6.5. *Assume that ν is a positive coin-toss probability measure on* $\mathbf{C}$ *and (f,g) is an orderly $\leq_{\mathrm{tt}}$-reduction. Let $F = F_{(f,g)}$, and assume that d is a $\nu^{(f,g)}$-martingale. Let $w \in \{0,1\}^*$, assume that $k = |w|$ is an (f,g)-step, and let $l = step(k+1)$. Then*

$$d(F(w)) = \sum_{u \in \{0,1\}^{l-k}} d(F(wu))\nu(wu|w).$$

Proof. Assume the hypothesis. Since d is a $\nu^{(f,g)}$-martingale and $\nu^{(f,g)}(F(w))$ is positive, we have

$$d(F(w)) = \sum_{b=0}^{1} d(F(w)b)\nu^{(f,g)}(F(w)b|F(w)).$$

It follows by Lemma 6.4 that

$$\begin{aligned} d(F(w)) &= \sum_{b=0}^{1} d(F(w)b) \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(wu) = F(w)b}} \nu(wu|w) \\ &= \sum_{b=0}^{1} \sum_{\substack{u \in \{0,1\}^{l-k} \\ F(wu) = F(w)b}} d(F(wu))\nu(wu|w) \\ &= \sum_{u \in \{0,1\}^{l-k}} d(F(wu))\nu(wu|w). \qquad \square \end{aligned}$$

Proof (Theorem 6.3). Assume the hypothesis, and let $F = F_{(f,g)}$.

To see that $(f,g)\widehat{\ }d$ is a ν-martingale, let $w \in \{0,1\}^*$, let $k = |w|$, and let $l = \mathit{step}(k+1)$. We have two cases.

Case 1. $\mathit{step}(k) = l$. Then

$$\begin{aligned} \sum_{b=0}^{1} (f,g)\widehat{\ }d(wb)\nu(wb) &= \sum_{b=0}^{1} \sum_{u \in \{0,1\}^{l-k-1}} d(F(wbu))\nu(wbu|wb)\nu(wb) \\ &= \sum_{b=0}^{1} \sum_{u \in \{0,1\}^{l-k-1}} d(F(wbu))\nu(wbu) \\ &= \sum_{u \in \{0,1\}^{l-k}} d(F(wu))\nu(wu) \\ &= (f,g)\widehat{\ }d(w)\nu(w). \end{aligned}$$

Case 2. $\mathit{step}(k) < l$. Then k is an (f,g)-step, so $(f,g)\widehat{\ }d(w) = d(F(w))$, whence by Lemma 6.5

$$(f,g)\widehat{\ }d(w)\nu(w) = \sum_{u \in \{0,1\}^{l-k}} d(F(wu))\nu(wu).$$

Calculating as in Case 1, it follows that

$$(f,g)\widehat{\ }d(w)\nu(w) = \sum_{b=0}^{1} (f,g)\widehat{\ }d(wb)\nu(wb).$$

This completes the proof that $(f,g)\widehat{\ }d$ is a ν-martingale.

To complete the proof, let $A \subseteq \{0,1\}^*$, and assume that d succeeds on $F(A)$. For each $n \in \mathbb{N}$, let $w_n = \chi_A[0..l_n - 1]$, where l_n is the unique (f,g)-step such that $|F(0^{l_n})| = n$. Then, for all $n \in \mathbb{N}$,

$$(f,g)\widehat{\ }d(w_n) = d(F(w_n)) = d(\chi_{F(A)}[0..n-1]),$$

so

$$\begin{aligned} \limsup_{k \longrightarrow \infty} (f,g)\widehat{\ }d(\chi_A[0..k-1]) &\geq \limsup_{n \longrightarrow \infty} (f,g)\widehat{\ }d(w_n) \\ &= \limsup_{n \longrightarrow \infty} d(\chi_{F(A)}[0..n-1]) \\ &= \infty. \end{aligned}$$

Thus $(f,g)\widehat{\ }d$ succeeds on A. $\square$

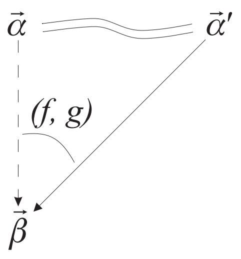


FIG. 1. *Schematic depiction of positive bias reduction.*

7. Positive bias reduction. In this section, we define and analyze a positive truth-table reduction that encodes an efficient, approximate simulation of one sequence of biases by another.

Intuitively, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive sequences of biases, then the positive bias reduction of $\vec{\alpha}$ to $\vec{\beta}$ is a $\leq_{\mathrm{tt}}$-reduction (f, g) that "tries to simulate" the sequence $\vec{\alpha}$ with the sequence $\vec{\beta}$ by causing $\mu^{\vec{\alpha}}$ to be the probability distribution induced by $\mu^{\vec{\beta}}$ and (f, g). In general, this objective will only be approximately achieved, in the sense that the probability distribution induced by $\mu^{\vec{\beta}}$ and (f, g) will actually be a probability distribution $\mu^{\vec{\alpha}'}$, where $\vec{\alpha}'$ is a sequence of biases such that $\vec{\alpha}' \approx \vec{\alpha}$. This situation is depicted schematically in Figure 1, where the broken arrow indicates that (f, g) "tries" to reduce $\vec{\alpha}$ to $\vec{\beta}$, while the solid arrow indicates that (f, g) actually reduces $\vec{\alpha}'$ to $\vec{\beta}$.

The reduction (f, g) is constructed precisely as follows.

CONSTRUCTION 7.1 (positive bias reduction). *Let $\vec{\alpha}$ and $\vec{\beta}$ be strongly positive sequences of biases. Let*

$$\delta = \inf \{\alpha_i, 1 - \alpha_i, \beta_i, 1 - \beta_i \mid i \in \mathbb{N}\},$$

$$c = \left\lceil \frac{-4 \log e}{\log(1 - \delta^2)} \right\rceil.$$

For each $x \in \{0,1\}^*$ and $0 \leq n < 2^{c|x|}$, let $q(x, n) = xy$, where y is the nth element of $\{0,1\}^{c|x|}$, and let $j(x, n)$ be the index of the string $q(x, n)$, i.e., $s_{j(x,n)} = q(x, n)$. Then the positive bias reduction of $\vec{\alpha}$ to $\vec{\beta}$ is the ordered pair (f, g) of functions defined by the procedure in Figure 2. (For convenience, the procedure defines additional parameters that are useful in the subsequent analysis.)

The following general remarks will be helpful in understanding Construction 7.1.

(a) The boldface variables $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots$ denote Boolean inputs to the Boolean function $g(x)$ being constructed. The Boolean function $g(x)$ is an OR of $k(x)$ Boolean functions $h(x, k)$, i.e.,

$$g(x) = \bigvee_{k=0}^{k(x)-1} h(x, k).$$

The Boolean functions $g(x, 0), g(x, 1), \ldots$ are preliminary approximations of the Boolean function $g(x)$. In particular,

$$g(x, k) = \bigvee_{j=0}^{k-1} h(x, j)$$

```
begin
input $x = s_i$;
$n := 0$;
$g(x,0) := 0$; $\alpha'_i(0) = 0$;
$k := 0$;
while $\alpha'_i(k) < \alpha_i - (i+1)^{-2}$ do
        begin
        $h(x,k,0) := 1$; $\gamma_{i,k}(0) := 1$;
        $l := 0$;
        while $\alpha'_i(k) + \gamma_{i,k}(l) - \alpha'_i(k) \cdot \gamma_{i,k}(l) > \alpha_i$ do
                begin
                $h(x,k,l+1) := h(x,k,l)$ AND $v_n$;
                $\gamma_{i,k}(l+1) := \gamma_{i,k}(l) \cdot \beta_{j(x,n)}$;
                $l := l+1$;
                $n := n+1$;
                end ;
        $l(x,k) := l$;
        $h(x,k) := h(x,k,l(x,k))$;
        $\gamma_{i,k} := \gamma_{i,k}(l(x,k))$;
        $g(x,k+1) := g(x,k)$ OR $h(x,k)$;
        $\alpha'_i(k+1) := \alpha'_i(k) + \gamma_{i,k} - \alpha'_i(k) \cdot \gamma_{i,k}$;
        $k := k+1$
        end ;
$k(x) := k$;
$n(x) := n$;
$f(x) := (q(x,0), \ldots, q(x,n(x)-1))$;
$g(x) := g(x,k(x))$;
$\alpha'_i := \alpha'_i(k(x))$
end .
```

FIG. 2. *Construction of positive bias reduction.*

for all $0 \le k \le k(x)$. Thus $g(x,0)$ is the constant-0 Boolean function.

(b) The Boolean function $h(x,k)$ is an AND of $l(x,k)$ consecutive input variables. The subscript n is incremented globally so that no input variable appears more than once in $g(x)$. Just as $g(x,k)$ is the kth "partial OR" of $g(x)$, $h(x,k,l)$ is the lth "partial AND" of $h(x,k)$. Thus $h(x,k,0)$ is the constant-1 Boolean function.

(c) The input variables $\boldsymbol{v}_0$, $\boldsymbol{v}_1, \ldots$ of g correspond to the respective queries $q(x,0), q(x,1), \ldots$ of f. If $A = F_{(f,g)}(B)$, then we have $[\![x \in A]\!] = g(x)(\boldsymbol{v}_0 \cdots \boldsymbol{v}_{n(x)-1})$, where each $\boldsymbol{v}_n = [\![q(x,n) \in B]\!]$. If B is chosen according to the sequence of biases $\vec{\beta}$, then $\beta_{j(x,n)}$ is the probability that $\boldsymbol{v}_n = 1$, $\gamma_{i,k}$ is the probability that $h(x,k) = 1$, and α'_i is the probability that $g(x) = 1$. The while-loops ensure that $\alpha_i - (i+1)^{-2} \le \alpha'_i \le \alpha_i$.

The following lemmas provide some quantitative analysis of the behavior of Construction 7.1.

LEMMA 7.2. *In Construction* 7.1, *for all* $x \in \{0,1\}^*$ *and* $0 \le k \le k(x)$,

$$l(x,k) \le 1 + \frac{c|x|}{2\log e}.$$

Proof. Fix such x and k, and let $l^* = l(x,k)$. If $l^* = 0$, the result is trivial, so assume that $l^* > 0$. Then, by the minimality of l^*,

$$\alpha_i'(k) + \gamma_{i,k}(l^* - 1) > \alpha_i,$$

so

$$\gamma_{i,k}(l^* - 1) > \alpha_i - \alpha_i'(k) > (i+1)^{-2},$$

so

$$(i+1)^{-2} < \gamma_{i,k}(l^* - 1) \leq (1-\delta)^{l^*-1}.$$

It follows that

$$-2\log(i+1) \leq (l^* - 1)\log(1-\delta),$$

whence

$$\begin{aligned} l^* &\leq 1 - \frac{2\log(i+1)}{\log(1-\delta)} \\ &\leq 1 - \frac{2|x|}{\log(1-\delta^2)} \\ &\leq 1 + \frac{c|x|}{2\log e}. \qquad \square \end{aligned}$$

LEMMA 7.3. *In Construction* 7.1, *for all* $x \in \{0,1\}^*$, *and* $0 \leq k \leq k(x) - 1$,

$$\alpha_i - \alpha_i'(k) \leq (1-\delta^2)^k.$$

Proof. Fix such x and k with $k < k(x) - 1$, and let $l^* = l(x,k)$. Then $\gamma_{i,k}(l^* - 1) > \alpha_i - \alpha_i'(k)$, so $\gamma_{i,k} \geq \delta \cdot \gamma_{i,k}(l^* - 1) > \delta \cdot (\alpha_i - \alpha_i'(k))$, whence

$$\begin{aligned} \frac{\alpha_i - \alpha_i'(k+1)}{\alpha_i - \alpha_i'(k)} &= \frac{\alpha_i - (\alpha_i'(k) + \gamma_{i,k} - \alpha_i'(k)\cdot\gamma_{i,k})}{\alpha_i - \alpha_i'(k)} \\ &= \frac{\alpha_i - \alpha_i'(k) - \gamma_{i,k}(1 - \alpha_i'(k))}{\alpha_i - \alpha_i'(k)} \\ &< 1 - \delta\cdot(1-\alpha_i'(k)) \\ &\leq 1 - \delta\cdot(1-\alpha_i) \\ &\leq 1 - \delta^2. \end{aligned}$$

The lemma now follows immediately by induction. $\square$

LEMMA 7.4. *In Construction* 7.1, *for all* $x \in \{0,1\}^*$,

$$k(x) \leq 1 + \frac{c|x|}{2\log e}.$$

Proof. Fix $x \in \{0,1\}^*$. By Lemma 7.3 and the minimality of $k(x)$,

$$\alpha_i - (1-\delta^2)^{k(x)-1} \leq \alpha_i'(k(x)-1) < \alpha_i - (i+1)^{-2},$$

so

$$(1-\delta^2)^{k(x)-1} > (i+1)^{-2},$$

so

$$k(x) < 1 - \frac{2\log(i+1)}{\log(1-\delta^2)} \leq 1 + \frac{c|x|}{2\log e}. \qquad \square$$

LEMMA 7.5. *In Construction* 7.1, *for all* $x \in \{0,1\}^*$,

$$n(x) \leq \left(1 + \frac{c|x|}{2\log e}\right)^2 \leq 2^{c|x|}.$$

Proof. Let $x \in \{0,1\}^*$. Then

$$n(x) = \sum_{k=0}^{k(x)-1} l(x,k),$$

so by Lemmas 7.2 and 7.4, and the bound $1 + t \leq e^t$,

$$n(x) \leq \left(1 + \frac{c|x|}{2\log e}\right)^2 \leq e^{\frac{c|x|}{\log e}} = 2^{c|x|}. \qquad \square$$

DEFINITION 7.1. *Let* (f,g) *be a* $\leq_{\rm tt}$*-reduction.*

1. (f,g) *is* positive *(briefly, a* $\leq_{\rm pos-tt}$*-reduction) if, for all* $A, B \subseteq \{0,1\}^*$, $A \subseteq B$ *implies*$F_{(f,g)}(A) \subseteq F_{(f,g)}(B)$.
2. (f,g) *is* polynomial-time computable *(briefly, a* $\leq^{\rm P}_{\rm tt}$*-reduction) if the functions* f *and* g *are computable in polynomial time.*
3. (f,g) *is* polynomial-time computable with linear-bounded queries *(briefly, a* $\leq^{\rm P,lin}_{\rm tt}$-reduction*) if* (f,g) *is a* $\leq^{\rm P}_{\rm tt}$*-reduction and there is a constant* $c \in \mathbb{N}$ *such that, for all* $x \in \{0,1\}^*$, $Q_{(f,g)}(x) \subseteq \{0,1\}^{\leq c(1+|x|)}$.

Of course, a $\leq^{\rm P,lin}_{\rm pos-tt}$-reduction is a $\leq_{\rm tt}$-reduction with all the above properties.

The following result presents the properties of the positive bias reduction that are used in the proof of our main theorem.

THEOREM 7.6 (positive bias reduction theorem). *Let* $\vec{\alpha}$ *and* $\vec{\beta}$ *be strongly positive,* P*-exact sequences of biases, and let* (f,g) *be the positive bias reduction of* $\vec{\alpha}$ *to* $\vec{\beta}$*. Then* (f,g) *is an orderly* $\leq^{\rm P,lin}_{\rm pos-tt}$*-reduction, and the probability measure induced by* $\mu^{\vec{\beta}}$ *and* (f,g) *is a coin-toss probability measure* $\mu^{\vec{\alpha}'}$*, where* $\vec{\alpha} \approx \vec{\alpha}'$.

Proof. Assume the hypothesis. By inspection and Lemma 7.5, the pair (f,g) is an orderly $\leq^{\rm P,lin}_{\rm pos-tt}$-reduction. (Lemma 7.5 also ensures that $f(x)$ is well-defined.) The reduction is also positive, since only AND's and OR's are used in the construction of $g(x)$. Thus (f,g) is an orderly $\leq^{\rm P,lin}_{\rm pos-tt}$-reduction.

By remark (c) following Construction 7.1, the probability measure induced by $\mu^{\vec{\beta}}$ and (f,g) is the coin-toss probability measure $\mu^{\vec{\alpha}'}$, where $\vec{\alpha}' = (\alpha'_0, \alpha'_1, \dots)$ is defined in the construction. Moreover,

$$\sum_{i=0}^{\infty} |\alpha_i - \alpha'_i| \leq \sum_{i=0}^{\infty} (i+1)^{-2} < \infty,$$

so $\vec{\alpha} \approx \vec{\alpha}'$. $\square$

$$\begin{tikzcd}
\vec{\alpha} \arrow[r, no head, squiggly] & \vec{\alpha}' \arrow[r, no head, squiggly] \arrow[d, dashed] & \vec{\alpha}'' \arrow[r, no head, dashed] \arrow[dl, "(f,g)"'] & d \arrow[d, dashed] \\
\vec{\beta} \arrow[r, no head, squiggly] & \vec{\beta}' \arrow[rr, no head, dashed] & & (f,g)\hat{\ }d
\end{tikzcd}$$

FIG. 3. *Scheme of proof of the bias equivalence theorem.*

8. Equivalence for complexity classes. Many important complexity classes, including P, NP, co-NP, R, BPP, AM, P/Poly, PH, PSPACE, etc., are known to be closed under $\leq^{\mathrm{P}}_{\mathrm{pos-tt}}$-reductions, hence certainly under $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reductions. The following theorem, which is the main result of this paper, says that the p-measure of such a class is somewhat insensitive to certain changes in the underlying probability measure. The proof is now easy, given the machinery of the preceding sections.

THEOREM 8.1 (bias equivalence theorem). *Assume that $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive P-sequences of biases, and let $\mathcal{C}$ be a class of languages that is closed under $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reductions. Then*

$$\mu^{\vec{\alpha}}_{\mathrm{p}}(\mathcal{C}) = 0 \Longleftrightarrow \mu^{\vec{\beta}}_{\mathrm{p}}(\mathcal{C}) = 0.$$

Proof. Assume the hypothesis, and assume that $\mu^{\vec{\alpha}}_{\mathrm{p}}(\mathcal{C}) = 0$. By symmetry, it suffices to show that $\mu^{\vec{\beta}}_{\mathrm{p}}(\mathcal{C}) = 0$.

The proof follows the scheme depicted in Figure 3. By Lemma 5.2, there exist P-exact sequences $\vec{\alpha}'$ and $\vec{\beta}'$ such that $\vec{\alpha} \approx \vec{\alpha}'$ and $\vec{\beta} \approx \vec{\beta}'$. Let (f,g) be the positive bias reduction of $\vec{\alpha}'$ to $\vec{\beta}'$. Then, by the positive bias reduction theorem (Theorem 7.6), (f,g) is an orderly $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reduction, and the probability measure induced by $\mu^{\vec{\beta}}$ and (f,g) is $\mu^{\vec{\alpha}''}$, where $\vec{\alpha}' \approx \vec{\alpha}''$.

Since $\vec{\alpha} \approx \vec{\alpha}' \approx \vec{\alpha}''$ and $\mu^{\vec{\alpha}}_{\mathrm{p}}(\mathcal{C}) = 0$, the summable equivalence theorem (Theorem 4.3) tells us that there is a p-$\vec{\alpha}''$-martingale d such that $\mathcal{C} \subseteq S^{\infty}[d]$. By the martingale dilation theorem (Theorem 6.3), the function $(f,g)\hat{\ }d$ is then a $\vec{\beta}'$-martingale. In fact, it is easily checked that $(f,g)\hat{\ }d$ is a p-$\vec{\beta}'$-martingale.

Now let $A \in \mathcal{C}$. Then, since $\mathcal{C}$ is closed under $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reductions, $F_{(f,g)}(A) \in \mathcal{C} \subseteq S^{\infty}[d]$. It follows by the martingale dilation theorem that $A \in S^{\infty}[(f,g)\hat{\ }d]$. Thus $\mathcal{C} \subseteq S^{\infty}[(f,g)\hat{\ }d]$. Since $(f,g)\hat{\ }d$ is a p-$\vec{\beta}'$-martingale, this shows that $\mu^{\vec{\beta}'}_{\mathrm{p}}(\mathcal{C}) = 0$. Finally, since $\vec{\beta} \approx \vec{\beta}'$, it follows by the summable equivalence theorem that $\mu^{\vec{\beta}}_{\mathrm{p}}(X) = 0$. ☐

It is clear that the bias equivalence theorem remains true if the resource bound on the measure is relaxed. That is, the analogs of Theorem 8.1 for p_2-measure, pspace-measure, rec-measure, constructive measure, and classical measure all immediately follow. We conclude by noting that the analogs of Theorem 8.1 for measure in E and measure in E_2 also immediately follow.

COROLLARY 8.2. *Under the hypothesis of Theorem* 8.1,

$$\mu^{\vec{\alpha}}(\mathcal{C}|\mathrm{E}) = 0 \Longleftrightarrow \mu^{\vec{\beta}}(\mathcal{C}|\mathrm{E}) = 0$$

and

$$\mu^{\vec{\alpha}}(\mathcal{C}|\mathrm{E}_2) = 0 \Longleftrightarrow \mu^{\vec{\beta}}(\mathcal{C}|\mathrm{E}_2) = 0.$$

Proof. If $\mathcal{C}$ is closed under $\leq^{\mathrm{P,lin}}_{\mathrm{pos-tt}}$-reductions, then so are the classes $\mathcal{C} \cap \mathrm{E}$ and $\mathcal{C} \cap \mathrm{E}_2$. □

9. Conclusion.

Our main result, the bias equivalence theorem, says that every strongly positive, P-computable, coin-toss probability measure ν is *equivalent* to the uniform probability measure μ, in the sense that

$$\nu_{\mathrm{p}}(\mathcal{C}) = 0 \Longleftrightarrow \mu_{\mathrm{p}}(\mathcal{C}) = 0$$

for all classes $\mathcal{C} \in \Gamma$, where Γ is a family that contains P, NP, co-NP, R, BPP, P/Poly, PH and many other classes of interest. It would be illuminating to learn more about which probability measures are, and which probability measures are not, equivalent to μ in this sense.

It would also be of interest to know whether the summable equivalence theorem can be strengthened. Specifically, say that two sequences of biases $\vec{\alpha}$ and $\vec{\beta}$ are *square-summably equivalent*, and write $\vec{\alpha} \approx^2 \vec{\beta}$, if $\sum_{i=0}^{\infty}(\alpha_i - \beta_i)^2 < \infty$. A classical theorem of Kakutani [9] says that, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive sequences of biases such that $\vec{\alpha} \approx^2 \vec{\beta}$, then for every set $C \subseteq \mathbf{C}$, X has (classical) $\vec{\alpha}$-measure 0 if and only if X has $\vec{\beta}$-measure 0. A constructive improvement of this theorem by Vovk [28] says that, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive, computable sequences of biases such that $\vec{\alpha} \approx^2 \vec{\beta}$, then for every set $X \subseteq \mathbf{C}$, X has constructive $\vec{\alpha}$-measure 0 if and only if X has constructive $\vec{\beta}$-measure 0. (The Kakutani and Vovk theorems are more general than this, but for the sake of brevity, we restrict the present discussion to coin-toss probability measures.) The summable equivalence theorem is stronger than these results in one sense, but weaker in another. It is stronger in that it holds for p-measure, but it is weaker in that it requires the stronger hypothesis that $\vec{\alpha} \approx \vec{\beta}$. We thus ask whether there is a "square-summable equivalence theorem" for p-measure. That is, if $\vec{\alpha}$ and $\vec{\beta}$ are strongly positive, p-computable sequences of biases such that $\vec{\alpha} \approx^2 \vec{\beta}$, is it necessarily the case that, for every set $X \subseteq \mathbf{C}$, X has p-$\vec{\alpha}$-measure 0 if and only if X has p-$\vec{\beta}$-measure 0? (Note: Kautz [10] has very recently answered this question affirmatively.)

Acknowledgments. We thank Giora Slutzki, Martin Strauss, and other participants in the ISU Information and Complexity Seminar for useful remarks and suggestions. We especially thank Giora Slutzki for suggesting a simplified presentation of Lemma 4.2. We also thank the referee for a very careful reading of our manuscript.

REFERENCES

[1] K. Ambos-Spies, E. Mayordomo, and X. Zheng, *A comparison of weak completeness notions*, in Proceedings of the Eleventh IEEE Conference on Computational Complexity, IEEE Computer Society Press, 1996, pp. 171–178.

[2] K. Ambos-Spies, S. A. Terwijn, and X. Zheng, *Resource bounded randomness and weakly complete problems*, Theoret. Comput. Sci., 172 (1977), pp. 195–207.

[3] J. Cai and A. L. Selman, *Fine separation of average time complexity classes*, in Proceedings of the Thirteenth Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, 1996, pp. 331–343.

[4] R. I. Freidzon, *Families of recursive predicates of measure zero*, J. Soviet Math., 6 (1976), pp. 449–455, 1972.

[5] P. R. Halmos, *Measure Theory*, Springer-Verlag, Berlin, New York, 1950.

[6] D. W. Juedes, *Weakly complete problems are not rare*, Computational Complexity, 5 (1995), pp. 267–283.
[7] D. W. Juedes and J. H. Lutz, *The complexity and distribution of hard problems*, SIAM J. Comput., 24 (1995), pp. 279–295.
[8] D. W. Juedes and J. H. Lutz, *Weak completeness in E and E_2*, Theoret. Comput. Sci., 143 (1995), pp. 149–158.
[9] S. Kakutani, *On the equivalence of infinite product measures*, Annals of Mathematics, 49 (1948), pp. 214–224.
[10] S. M. Kautz, *Resource-bounded randomness and compressibility with respect to nonuniform measures*, in Proceedings of the International Workshop on Randomization and Approximation Techniques in Computer Science, Springer-Verlag, Berlin, New York, 1997, pp. 197–211.
[11] J. H. Lutz, *Almost everywhere high nonuniform complexity*, J. Comput. System Sci., 44 (1992), pp. 220–258.
[12] J. H. Lutz, *The quantitative structure of exponential time*, in Complexity Theory Retrospective II, L. A. Hemaspaandra and A. L. Selman, eds., Springer-Verlag, Berlin, New York, 1997, pp. 225–254.
[13] J. H. Lutz, *Weakly hard problems*, SIAM J. Comput., 24 (1995), pp. 1170–1189.
[14] J. H. Lutz, *Observations on measure and lowness for Δ_2^{P}*, Theory Comput. Syst., 30 (1997), pp. 429–442.
[15] J. H. Lutz and E. Mayordomo, *Measure, stochasticity, and the density of hard languages*, SIAM J. Comput., 23 (1994), pp. 762–779.
[16] J. H. Lutz and E. Mayordomo, *Genericity, measure, and inseparable pairs*, in preparation.
[17] J. H. Lutz and E. Mayordomo, *Cook versus Karp-Levin: Separating completeness notions if NP is not small*, Theoret. Comput. Sci., 164 (1996), pp. 141–163.
[18] E. Mayordomo, *Almost every set in exponential time is P-bi-immune*, Theoret. Comput. Sci., 136 (1994), pp. 487–506.
[19] K. Mehlhorn, *The "almost all" theory of subrecursive degrees is decidable*, in Proceedings of the Second Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci., 14, Springer-Verlag, New York, 1974, pp. 317–325.
[20] J. C. Oxtoby, *Measure and Category*, 2nd ed., Springer-Verlag, Berlin, New York, 1980.
[21] K. W. Regan, D. Sivakumar, and J. Cai, *Pseudorandom generators, measure theory, and natural proofs*, in 36th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1995, pp. 26–35.
[22] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.
[23] C. P. Schnorr, *Klassifikation der Zufallsgesetze nach Komplexität und Ordnung*, Z. Wahrscheinlichkeitstheorie verw. Geb., 16 (1970), pp. 1–21.
[24] C. P. Schnorr, *A unified approach to the definition of random sequences*, Math. Systems Theory, 5 (1971), pp. 246–258.
[25] C. P. Schnorr, *Zufälligkeit und Wahrscheinlichkeit*, Lecture Notes in Mathematics, 218 (1971).
[26] C. P. Schnorr, *Process complexity and effective random tests*, J. Comput. System Sci., 7 (1973), pp. 376–388.
[27] M. van Lambalgen, *Random Sequences*, Ph.D. thesis, Department of Mathematics, University of Amsterdam, Amsterdam, the Netherlands, 1987.
[28] V. G. Vovk, *On a randomness criterion*, Soviet Mathematics Doklady, 35 (1987), pp. 656–660.
[29] A. K. Zvonkin and L. A. Levin, *The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms*, Russian Math. Surveys, 25 (1970), pp. 83–124.

SIAM J. COMPUT.
Vol. 29, No. 1, pp. 327–335

THE SWAPPING PROBLEM ON A LINE[*]

SHOSHANA ANILY[†], MICHEL GENDREAU[‡], AND GILBERT LAPORTE[‡]

Abstract. We consider the problem of optimally swapping objects between N workstations, which we refer to as nodes, located on a line. There are m types of objects, and the set of object-types is denoted by $S = \{1, \dots, m\}$. Object-type 0 is a dummy type, the *null object*. Each node v contains one unit of a certain object-type $a_v \in S \cup \{0\}$ and requires one unit of object-type $b_v \in S \cup \{0\}$. We assume that the total supply equals the total demand for each of the object-types separately. A vehicle of unit capacity ships the objects so that the requirements of all nodes are satisfied. The set of object-types is partitioned into two sets: objects that may be temporarily dropped at intermediate nodes before reaching their destination and objects that have to be shipped directly from their origin to their destination. The objective is to design a route that starts and ends at the depot and a feasible assignment of object-types to the route's arcs so that the total distance is minimized. We propose an $O(N^2)$ algorithm to compute the optimal solution for this problem.

Key words. the swapping problem, transshipment problem

AMS subject classifications. 05C12, 05C45, 05C40, 05C85

PII. S0097539797323108

1. Introduction.

A set V of N workstations, which we refer to as nodes, is located on a line. We are also given a set of m object-types $S = \{1, \dots, m\}$. Object-type 0 is the null object. Each workstation v contains one unit of a certain object $a_v \in S \cup \{0\}$ and requires one unit of object $b_v \in S \cup \{0\}$. $a_v = 0 (b_v = 0)$ means that no object is currently (required) at node v. For each object-type separately, the total supply (i.e., the number of nodes currently containing that object-type) equals the total demand (the number of nodes requiring that object-type). A vehicle of unit capacity that starts and ends at node $v_D \in V$ ships the objects from their initial locations so that the requirements of the workstations are satisfied. The objective is to design a feasible route of minimum length.

The set of object-types is partitioned into two sets $S = S_d \cup S_n$. Objects in S_d may be temporarily dropped at intermediate nodes before reaching their destination, while objects in S_n have to be shipped *directly* from their origin to their destination. One of these two sets may be empty. Clearly, the optimal solution when some of the objects may be dropped cannot be worse than in the case that $S = S_n$.

The general swapping problem where the workstations are the nodes of an undirected complete graph was studied by Anily and Hassin (1992). The authors show that the problem is NP-hard and prove the existence of an optimal solution that satisfies certain structural properties. They design two heuristics whose worst-case bounds are 2.5. The proposed heuristics are based on the composition of the optimal solutions to $m + 1$ matching problems, one for each object-type $i \in S \cup \{0\}$, where nodes v with $a_v = i$ are matched with nodes w with $b_w = i$. This step results in a set of disjoint cycles that are then patched into a single Eulerian tour. Except for Anily and Hassin (1992) and the current paper, the literature confines itself to systems containing one unit of each object-type, and moreover, all objects are in either S_n or

[*]Received by the editors June 11, 1997; accepted for publication (in revised form) May 29, 1998; published electronically September 22, 1999.
http://www.siam.org/journals/sicomp/29-1/32310.html

[†]Faculty of Management, Tel-Aviv University, Tel-Aviv 69978, Israel (anily@post.tau.ac.il).

[‡]École des Hautes Études Commerciales, Montreal, PQ, Canada (michelg@crt.umontreal.ca, gilbert@crt.umontreal.ca).

S_d. In the stacker-crane problem analyzed by Frederickson, Hecht, and Kim (1978) (see also Johnson and Papadimitriou (1985)), some directed arcs are given and the objective is to find a directed closed tour of minimum length containing these arcs. The stacker-crane problem is a special case of the swapping problem, where $S = S_n$, and there is one unit of each object-type. The authors propose a polynomial approximation for this problem whose worst-case bound is 9/5. Atallah and Kosaraju (1988) analyze the swapping problem on a line and on a circular track when there exists exactly one unit of each object-type in S. They consider the cases of $S = S_n$ (no drops) and $S = S_d$ (with drops). Their motivation to study the problem arises from the movement of a robot arm that is supposed to rearrange m objects among N stations. The robot arm consists of a single link that rotates around a fixed pivot. The link's length is variable since it can be extended in and out. A gripper that can grasp any of the objects is positioned at the end of the link. Minimizing the total distance traveled by the gripper is NP-hard. Instead, the authors focus separately on minimizing the total (a) telescoping motion, which corresponds to moving along a linear track, and (b) angular motion, which corresponds to moving along a circular track. The paper provides low polynomial algorithms for computing the optimal route: for the no-drop case the algorithm runs in $O(m + N\alpha(N))$ ($\alpha()$ is the inverse of Ackerman's function) for the linear track and in $O(m + N \log N)$ for the circular track (this last bound is further tightened in Frederickson (1993)). For the with-drop case the algorithm runs in $O(m + N)$ for both the linear and circular tracks. Frederickson and Guan (1992, 1993) studied the same problem as that of Atallah and Kosaraju (1988) when the objects are located on the vertices of a tree and the vehicle travels along its edges. For the with-drop case they present two algorithms that run in $O(m + Nq)$, where $q \leq \min\{m, N\}$. The no-drop case is shown to be NP-hard, and the authors provide two heuristics that run in low polynomial time with worst-case ratios of 1.5 and 1.25, respectively.

In this paper, we develop an $O(N^2)$-step algorithm for the linear track allowing no-drop and with-drop objects. Section 2 introduces the terminology, and section 3 establishes the necessary structural properties of any optimal solution. Finally, the main algorithm and its analysis is presented in section 4.

2. Notations and preliminaries.

$V = \{v_1, v_2, \ldots, v_N\}$ is a set of N workstations. Vertex $v_D, 1 \leq D \leq N$, is the depot where the vehicle starts and terminates the tour. The vertices are indexed according to their location on the line from left to right. $S = \{1, \ldots, m\}$ is a set of m object-types. Object-type 0 denotes the null object. Each vertex v is associated with a pair $(a_v, b_v) \in [S \cup \{0\}] \times [S \cup \{0\}]$, where a_v is the object-type currently at v and b_v is the object-type desired at v. Without loss of generality (w.l.o.g.) we assume that $a_v \neq b_v$. (If $a_D = b_D = 0$, then an equivalent problem can be defined such that $a_D \neq b_D$ by introducing a new node at the depot's location and a new object-type $m + 1$. Associate the depot with $(0, m + 1)$ and the new vertex with $(m + 1, 0)$.) The set S is partitioned into S_n, the set of *no-drop* objects-types, and S_d, the set of object-types that can be *dropped* at intermediate vertices. $d(u, v)$ is the distance between vertices u and v. $P^i \subseteq V$ are the supply vertices of object-type $i (i = 0, 1, \ldots, m)$, i.e., $P^i = \{v : a_v = i\}$. Let $P^i = \{p_{i1}, p_{i2}, \ldots, p_{ik(i)}\}$. $R^i \subseteq V$ are the vertices that require object-type $i (i = 0, 1, \ldots, m)$, i.e., $R^i = \{v : b_v = i\}$. Let $R^i = \{r_{i1}, r_{i2}, \ldots, r_{ik(i)}\}$. Let $V^i = P^i \cup R^i$, and $k(i) = |P^i| = |R^i|$ is the number of objects of type i in the system. Thus, $\sum_{i=0}^{m} k(i) = N$. The supply (demand) vertices of object-type i are indexed according to their location from left to right.

If $S = S_n$, then once we know which supply vertex in P^i serves each of the demand vertices in R^i for $i = 1, \ldots, m$, our problem reduces to the respective problem solved by Atallah and Kosaraju (1988).

DEFINITION 2.1. *A* path *is a sequence of directed arcs where the tail of one arc is the head of the preceding arc in the sequence, and all arcs in the sequence are assigned the same object-type* $i, 0 \leq i \leq m$.

DEFINITION 2.2. *If object-type* $i, 1 \leq i \leq m$, *that is initially located at vertex* p_{ih} *is used to supply the requirement of vertex* $r_{ih'}$, *then the path from* p_{ih} *to* $r_{ih'}$ *along which the object is shipped is called a* service path. *(See below for an extension of the definition for object-type* 0.*)*

Assume w.l.o.g. that objects are never dropped in order to pick up objects of the same type. Thus, in any feasible solution and any $v \in V$ with $a_v = i$, there exists a service path of object i initiating at v and ending at vertex u, $u \in V$ with $b_u = i$, $1 \leq i \leq m$. If $i \in S_n$, then the arcs of this path appear consecutively in the solution; i.e., the vehicle traverses the service path with no intermediate stops. If $i \in S_d$, then the arcs of this path appear in the solution in the same order as in the path, but not necessarily consecutively, meaning that the vehicle may drop the object and pick it up later.

DEFINITION 2.3. *A segment of the solution that the vehicle traverses empty is called a* deadheading.

According to Anily and Hassin (1992), if $a_v = 0$ for $v \in V$, then any feasible solution includes a path of deadheadings originating at v and ending at some vertex u with $b_u = 0$. Such a path is called a *service path* of the null object. Note that a feasible solution consists of $k(i)$ service paths for each object-type i, $i \in S \cup \{0\}$ and possibly some additional deadheadings.

DEFINITION 2.4. *The* end points *of a path starting at* u *and ending at* v, $u, v \in V$ *are the vertices* u *and* v.

DEFINITION 2.5. *A given arc is said to* cover *all points (not necessarily vertices) on the track in between its tail and its head including its end points. A path covers all points covered by its arcs.*

DEFINITION 2.6. *A path is* left-to-right (right-to-left) *if each of its arcs is directed to the right (left).*

DEFINITION 2.7. *Two paths are* intersecting in opposite directions *if* (a) *one path is left-to-right and other is right-to-left and* (b) *at least one end point of one path is covered by the other path.*

3. Structural properties of an optimal solution.

In this section we prove some theorems that ensure the existence of an optimal solution that satisfies some specified properties. The theorems hold for all the problem's variants discussed here.

THEOREM 3.1. *There exists an optimal solution that does not contain any pair of service paths for the same object-type* $i, 0 \leq i \leq m$, *that are intersecting in opposite directions.*

Proof. The proof is by contradiction. Suppose the theorem is false, i.e., any optimal routing contains service paths of the same object-type that are intersecting in opposite directions. Consider an optimal solution and let SP_1 and SP_2 be two such service paths for object-type i, $0 \leq i \leq m$. Suppose SP_1 connects $p_{ij(1)}$ to $r_{ih(1)}$ and SP_2 connects $p_{ij(2)}$ to $r_{ih(2)}$. W.l.o.g. let SP_1 be a left-to-right path. We show that there exists an alternative feasible solution that follows the same route and the only difference is with respect to the assignment of object i to the arcs of SP_1 and SP_2. Some arcs on these paths are assigned object i and the others are turned into deadheadings. As a consequence, the two new service paths are no longer intersecting in opposite directions. For the null object, the new solution is identical to the given one, but we distinguish differently between the service paths of object 0 and the deadheadings.

According to our assumptions, there exists a segment (u, v) that is covered by SP_1 and SP_2, where SP_1 is directed from u to v and SP_2 from v to u. The alternative solution is defined such that $r_{ih(1)}(r_{ih(2)})$ is served from $p_{ij(2)}(p_{ij(1)})$. It is easy to verify that the two new service paths consist of exactly those parts in SP_1 and SP_2 that do not cover (u, v) and their directions are consistent with the respective directions of SP_1 and SP_2. Also, the segment (u, v) on SP_1 and SP_2 is turned into a deadheading.

In the following, we derive additional properties satisfied by optimal paths.

DEFINITION 3.2. *$V^{\ell(i)} \subseteq V^i$ is* consecutive *if it consists of V^i nodes that appear consecutively on the line.*

DEFINITION 3.3. *A* consecutive partition *of V^i is a partition of V^i into consecutive disjoint subsets.*

DEFINITION 3.4. *A subset of vertices $V^{\ell(i)} \subseteq V^i$ is called* balanced *if it is consecutive and is of even cardinality, where half of its vertices are in P^i and the remaining are in R^i.*

DEFINITION 3.5. *A subset is* minimally balanced *if it is balanced and there does not exist any consecutive partition of the subset into two balanced subsets.*

DEFINITION 3.6. *The* consecutive minimally balanced partition (CMBP) *of V^i is the consecutive partition of V^i, where each of the subsets in the partition is minimally balanced.*

Note that any feasible solution is associated with $m + 1$ sets of service paths, one set for each object-type. The service paths of object i induce a consecutive balanced partition of V^i defined recursively as follows: (1) the two end points of a service path belong to the same subset in the partition; (2) all vertices of V^i covered by a certain service path belong to the same subset as the end points of the path; and (3) the subsets are minimal. It is easy to see that for a given solution the consecutive balanced partition of V^i induced by the service paths of object i is well defined. Moreover, if the solution does not contain any intersecting in opposite directions service paths for the same object-type, then all service paths within a set of this partition are in the same direction.

THEOREM 3.7. *In any optimal solution that does not contain intersecting in opposite directions service paths of object i, the service paths of this object-type* (1) *induce the CMBP of V^i and* (2) *have constant total length.*

Proof. Suppose Z is an optimal solution that satisfies the property of the theorem. The service paths of object-type i that are associated with solution Z induce a consecutive balanced partition of V^i. We want to show first that the subsets of this partition are *minimally balanced*. Assume w.l.o.g. that the leftmost vertex in V^i is a supply vertex $s_1 \in P^i$. Let $L_{i1} \subseteq V^i$ be the set induced by solution Z that contains s_1.

The proof is by induction on the cardinality of V^i. The minimum cardinality of V^i is two; i.e., V^i consists of a supply vertex and a demand vertex and the theorem is trivial. According to the inductive hypothesis, the theorem holds for any set of cardinality less than $|V^i|$. Suppose $|V^i| > 2$. We distinguish between two cases: (a) No consecutive partition of V^i into two balanced subsets exists. Thus, $L_{i1} = V^i$; i.e., the partition of V^i induced by the service paths of object-type i in Z is into a single set, which is the CMBP of V^i. (b) There exists a consecutive balanced partition of V^i into two subsets: let $V^i = V_{i1} \cup V_{i2}$ be a consecutive partition of V^i, where V_{i1} is a minimally balanced subset that contains s_1. We will show that $L_{i1} = V_{i1}$ and the rest will follow by the inductive hypothesis. Suppose that $V_{i1} \subset L_{i1}$ but $V_{i1} \neq L_{i1}$. Since s_1 is the leftmost supply vertex of object i on the line, it must serve a demand vertex to its right. As a consequence, all service paths within L_{i1} should be from left to right, otherwise there would be intersecting paths in opposite directions within L_{i1}, in contradiction to our assumption about Z. Since $V_{i1} \subset L_{i1}$ but $V_{i1} \neq L_{i1}$, there should be in Z a supply vertex $p \in V_{i1}$ that serves a demand vertex $r \in L_{i1} - V_{i1}$. In view of the fact that V_{i1} is balanced, at least one of the demand vertices r' in V_{i1} should be served according to Z by a supply vertex $s' \in L_{i1} - V_{i1}$. This contradicts the assumption that Z does not contain any service paths for object-type i that are intersecting in opposite directions.

In order to prove the second part of the theorem, we will show that the total length of the service paths of object i within any subset of the CMBP of V^i is constant. Suppose w.l.o.g. that we are given a minimally balanced subset of the CMBP of V^i for which all service paths are from left to right. Also suppose that x and y are consecutive vertices in V^i, where x is strictly to the left of y. Let $lp^i(z)(lr^i(z))$ denote the number of supply (demand) vertices from V^i within the subset that are located to the left or at the location of z. Then, the number of service paths of object i that cover the segment connecting x to y is $lp^i(x) - lr^i(x)$, which is strictly positive according to our assumptions. Accordingly, a simple calculation that depends only on the subset gives the total length of service paths within that subset.

4. The algorithm for the linear track case.

In this section we present an algorithm for finding an optimal policy for linear graphs. As a consequence of the previous section, an optimal solution exists in which the service paths of each object-type i, $0 \leq i \leq m$, induce the CMBP of V^i. Let $\{V_{i\ell}\}_{\ell=1,\dots,L(i)}$ denote the CMBP of V^i for $i = 0, \dots, m$. We now propose an algorithm for computing an optimal routing policy. The first step in the algorithm is to define a directed graph on V that consists of those arcs that must appear in such an optimal solution. Each arc in the graph connects two vertices in $V_{i\ell}$ for some $i = 0, \dots, m$ and $\ell = 1, \dots, L(i)$.

DEFINITION 4.1. *A set $V_{i\ell}$ of the CMBP of V^i is called a* left-set (right-set) *if the number of supply vertices is no smaller (no larger) than the number of demand vertices of object-type i in any consecutive subset of $V_{i\ell}$ that contains the leftmost vertex of $V_{i\ell}$.*

The directed graph G on V is defined by the following algorithm, called Basic Graph.

ALGORITHM BASIC GRAPH.

Step 0. For $i = 0, \dots, m$ and $\ell = 1, \dots, L(i)$ do steps 1 and 2:

Step 1. Let k be the number of demand vertices in $V_{i\ell}$. If $V_{i\ell}$ is a left-set (right-set), then index the demand vertices in $V_{i\ell}$ from left to right (right to left) as $\{r_1, r_2, \dots, r_k\}$. Let $G_{i\ell}$ initially be a graph with no arcs. Add to $G_{i\ell}$ a directed arc from each supply vertex in $P^i \cap V_{i\ell}$ toward the first demand vertex in $R^i \cap V_{i\ell}$ to its right (left), i.e., the first demand vertex it may serve.

Step 2. If $|V_{i\ell}| = 2$, then stop (the graph $G_{i\ell}$ contains a single arc). Otherwise, set $j = 1$; while $j < k$ do begin: Count the number of incoming arcs in $G_{i\ell}$ to r_j. Let this number be $in(j)$; add $in(j) - 1$ arcs to $G_{i\ell}$ all from r_j towards r_{j+1}. endwhile;

Step 3. Let graph G be the composition of all subgraphs $G_{i\ell}$ for $i = 0, \dots, m$ and $\ell = 1, \dots, L(i)$.

Remark. Note that in Step 2, $V_{i\ell}$ is a subset of the CMBP of V^i; thus $in(j) > 1$ for $j \leq k - 1$ and $in(k) = 1$.

LEMMA 4.1. *The in-degree equals the out-degree for any of the vertices of graph G.*

Proof. We have to show that the number of incoming arcs equals the number of outgoing arcs for any vertex $v \in V$. Let $v \in V$, and denote $(a_v, b_v) = (k, j)$ k, $j \in S \cup \{0\} k \neq j$. Also suppose that $v \in V_{k\ell} \cap V_{jh}$. The only arcs incident to v are arcs in the subgraphs $G_{k\ell}$ and G_{jh}. In $V_{k\ell}$, v is a supply vertex; thus exactly one arc exits from v in $G_{k\ell}$. In V_{jh}, v is a demand vertex; thus in G_{jh}, the number of outgoing arcs from v is one less than the number of incoming arcs to v. Thus overall, G satisfies the lemma.

A directed graph may be partitioned into a collection of equivalence classes such that vertices w and v are in the same class if and only if the graph contains directed paths from v to w and from w to v. (See section 5.5 in Aho, Hopcroft, and Ullman (1974).) The subgraph induced by a certain class consists of the vertices in the class as well as all edges in the graph that connect a pair of vertices in the class. These subgraphs are called the *strongly connected components* of the given graph. In light of Lemma 4.1, the union of the strongly connected components of G results in G itself.

We first demonstrate the algorithm when G is strongly connected: Since G is Eulerian, there exists an optimal solution that consists of G's arcs and does not use the drop option. This is observed by noting that if the number of incoming arcs to vertex v is k, $k > 1$, then *all* incoming arcs to v carry item b_v, where $k-1$ outgoing arcs carry item b_v. Thus, in $k-1$ of the k entrances to v the vehicle continues with the same item; at one entrance item b_v is unloaded and at one exit item a_v is loaded. Any Euler tour in G produces an optimal solution by starting at the depot. The complexity of finding such a tour is linear in the number of arcs of G. A simple calculation demonstrates that the number of arcs in G carrying item i is at most $k(i)(k(i)+1)/2$. Thus, G contains at most $\sum_{i=0}^{m} k(i)(k(i)+1)/2$ arcs. Since $\sum_{i=0}^{m} k(i) = N$, the maximum number of arcs in G is of order $O(N^2)$.

We continue with the general case when G is not necessarily strongly connected, i.e., G is the union of a number of strongly connected components where each is an Eulerian subgraph. Up to now, we have not used the fact that the vehicle is empty along arcs associated with the null object and we have not used the drop option. For that sake, we extend the term "connectivity:" It is not necessarily the case that for two different components of G none is reachable from the other, moreover it may well be that each is reachable from the other. An arc in G may cover intermediate vertices, thus it consists of a sequence of *basic arcs*, where a basic arc is defined as a directed arc connecting two consecutive vertices of V. We distinguish between three types of arcs in G: (1) arcs of object-type i, $i \in S_n$; (2) arcs of object-type i, $i \in S_d$; and (3) arcs of the null object. Arcs of no-drop objects should be followed continuously from their initial vertex to their terminal vertex; i.e., the respective sequence of basic arcs should occur in the solution consecutively. Arcs of drop-objects should be followed from their initial vertex to their terminal vertex with possible stops at intermediate vertices for drops; i.e., the respective basic arcs should be followed at the order they occur in the sequence, but not necessarily consecutively. The basic arcs of the null object occur in the solution with no restriction on their order.

DEFINITION 4.2. *Let C^s_ℓ and C^s_k be different strongly connected components of G. C^s_k is* directly reachable *from C^s_ℓ if at least one of the following conditions holds:* (1) *There exists an arc of the null object in G that covers vertices $v \in C^s_\ell$ and $w \in C^s_k$ and is directed from v to w; or* (2) *there exists an arc in G of object i, $i \in S_d$, whose tail is in $C^s_\ell \cap V^i$, and at least one vertex in C^s_k is covered by this arc. G's arcs that allow direct reachability of strongly connected components are called* reachability arcs.

DEFINITION 4.3. *Let $C_1^s, C_2^s, \dots, C_n^s$ be different strongly connected components of G. We say that C_n^s is* reachable *from C_1^s if C_k^s is directly reachable from C_{k-1}^s for $k = 2, \dots, n$.*

DEFINITION 4.4. *Two different strongly connected components of G, C_1^s and C_2^s, such that each is reachable from the other are said to be* weakly connected.

For practical purposes, a maximal weakly connected component is a connected component since *starting at any vertex* in the set there exists a closed tour that serves all vertices in the set, possibly by using the drop option. Therefore, we repartition V into rougher equivalence classes by grouping the strongly connected components of G according to the weak connectivity relation. Let $C_1^w, C_2^w, \dots, C_k^w$ be the weakly connected components of G, where the depot is assumed to be in C_1^w. In each of these components identify a closed feasible tour that serves all its vertices: This tour is a patching of the tours found in the strongly connected components, where the patching is done along reachability arcs. For that sake, we extend the reachability definition to weakly connected components.

DEFINITION 4.5. *C_h^w is said to be reachable from C_ℓ^w if C_h^w contains a strongly connected component that is reachable from a strongly connected component contained in C_ℓ^w, $\ell \neq h$.*

DEFINITION 4.6. *A weakly connected component of G that is not reachable from any other weakly connected component is said to be an* unreachable component. *Let C_1^w be an unreachable component independently of its reachability from other components as the tour should start at the depot.*

DEFINITION 4.7. *The* reachable set *of the unreachable component C_ℓ^w is the set of weakly connected components of G that are reachable from C_ℓ^w. (C_ℓ^w is said to be reachable from itself.) We note that a weakly connected component of G either may be an unreachable component or belongs to at least one reachable set of some unreachable component.*

If C_k^w is reachable from C_ℓ^w, then there exists a feasible closed tour that *starts* at C_ℓ^w and serves all vertices in $C_\ell^w \cup C_k^w$: The vehicle, while either carrying a droppable item loaded at C_ℓ^w or while being empty, traverses a reachability arc of G that connects two vertices of C_ℓ^w and covers a vertex of C_k^w; in the first case, the item may be dropped at such a vertex of C_k^w. The vehicle may then serve all vertices of C_k^w; in the first case the vehicle then reloads the droppable item. The vehicle then continues toward its destination in C_ℓ^w. Thus, a closed feasible tour that starts at the unreachable component (the tour starts in C_1^w at the depot) may be identified within each of the reachable sets such that all vertices within the set are served. In order to execute this step, no augmentation of the graph is needed, as it is a patching of the tours that have been found separately in each of the weakly connected components contained within the reachable set, where the patching is done along reachability arcs. Thus, if C_1^w is the only unreachable component, no augmentation of G is required, i.e., there exists an optimal solution with total length identical to the total length of G's arcs. Otherwise, G must be augmented in order to reach each of the unreachable components from C_1^w. Each augmentation arc added to G must be traversed twice, once in each direction. We next propose a minimum cost augmentation technique for G whose complexity time is $O(N^2 \log N)$; then we show how the complexity may be reduced to $O(N \log N)$ by using the fact that the vertices are located on a line.

Define a directed graph T on the set of vertices that correspond one-to-one to the unreachable components of G. Let node ℓ of T represent C_ℓ^w (node 1 of T corresponds to C_1^w). There exists a directed arc in T from node ℓ to node k, $k \neq 1$, if and only if there exists a pair of vertices v_j and v_h in V such that v_j is in the reachable set of C_ℓ^w and v_h is in the unreachable component C_k^w. The length of this arc, denoted by $\delta(\ell, k)$, is defined as the length of the shortest arc connecting the reachable set of C_ℓ^w and unreachable component C_k^w. The minimum cost augmentation of G may be found by solving a *minimum directed spanning tree* (MDST) on T rooted at node 1. The MDST was solved independently by Chu and Liu (1965), Edmonds (1967), and Bock (1971). The complexity of their algorithm on a general directed graph with N nodes and $|E|$ edges is $O(\min\{|E| \log N, N^2\})$. The number of nodes of T is the number of unreachable components which is at most $O(N)$. Thus, T contains at most $O(N^2)$ edges. Therefore, the complexity of the MDST algorithm on T is $O(N^2 \log N)$. Below we show that due to the linearity of G, we may apply the MDST algorithm on a subgraph of T that contains at most $O(N)$ edges. As a result the complexity of this step will be reduced to $O(N \log N)$.

DEFINITION 4.8. *A vertex $v_j \in V \cap S$, $j > 1$, is called* extreme to the left in S *if and only if $v_{j-1} \notin S$; a vertex $v_j \in V \cap S$, $j < N$, is called* extreme to the right in S *if and only if $v_{j+1} \notin S$. A vertex is called* extreme *if it is either extreme to the left or extreme to the right.*

Let v_j be an extreme to the left (right) vertex in the reachable set of the unreachable component C_ℓ^w. Let $v_h \in V$, $h < j (h > j)$, be the rightmost (leftmost) vertex that belongs to some unreachable component C_k^w, $k \notin \{1, \ell\}$. If such a vertex h exists, then we say that the extreme vertex v_j and the reachable set of C_ℓ^w have a neighboring unreachable component C_k^w. *Each extreme vertex may have at most two neighboring unreachable components (one to its left and one to its right), but a reachable set may have several neighboring unreachable components. Due to the linearity of the track, there exists a minimum cost augmentation of G using only those edges of T that connect reachable sets to* their neighboring *unreachable components. Indeed, it is sufficient to include in T only edges connecting a vertex ℓ to a vertex k if and only if C_k^w, $k \notin \{1, \ell\}$, is a neighboring unreachable component of the reachable set of C_ℓ^w. The length of such an edge is given by $\delta(\ell, k) = \min\{d(v, w)$: v* is an extreme vertex in the reachable set of *C_ℓ^w and $w \in C_k^w\}$. According to the new procedure, T now contains a subset of the arcs that were previously contained in T. An arc of T that is eliminated is not needed in the augmentation of G as it is too expensive and can be replaced by a cheaper sequence of those arcs that are left in T. It is easily verified that any unreachable component C_k^w can be reached from any reachable set C_ℓ^w, $k \notin \{1, \ell\}$, via those arcs that are left in T. This property ensures that T contains a directed spanning tree. The number of directed arcs in T is at most of size $O(N)$, as each extreme vertex may have at most two neighboring unreachable components.*

Let MDST(T) be the MDST length of T, and let $A(T)$ be its corresponding set of arcs on the line. Along the arcs of $A(T)$ the vehicle travels empty twice, once in each direction. There exists an optimal solution for the problem whose length is $L(G) + 2\text{MDST}(T)$, where $L(G)$ is the total length of all arcs in G. The complexity of the whole procedure is $O(N^2)$. Below we summarize the complete algorithm.

THE SWAPPING ALGORITHM FOR THE LINEAR TRACK CASE.

Step 1. Apply the Basic Graph algorithm on V. Let G be the resulting graph, and let $L(G)$ be its total length.

Step 2. Identify the strongly connected components of G.

Step 3. If the system contains the null object or $S_d \neq \emptyset$, then partition the strongly connected components into equivalence classes according to the weak connectivity relation. Let $C_1^w, C_2^w, \ldots, C_k^w$ be the weakly connected components, with the depot at C_1^w. Within each of them find a closed feasible tour that serves all its vertices, using only G's arcs where some of them may be broken into basic arcs. Identify the unreachable components, the reachable sets, and a corresponding set of reachability arcs.

Step 4. Define a directed graph T on the set of nodes that corresponds one-to-one to the unreachable components. A directed arc from node ℓ to node k in T, $k \neq 1$, exists if and only if the unreachable component C_k^w is a neighbor of some extreme vertex in the reachable set of the unreachable component C_ℓ^w. The distance between these nodes is determined by the minimum distance between an extreme vertex in the reachable set of C_ℓ^w and an extreme vertex in its neighboring unreachable component C_k^w. Let MDST(T) be the MDST length on T, and let $A(T)$ be the corresponding set of arcs in the linear track.

Step 5. Use two copies of $A(T)$'s arcs and reachability arcs that connect weakly connected components within the same reachable set to patch the closed tours associated with the weakly connected components. Along $A(T)$'s arcs the vehicle travels empty. Starting at the depot, follow a feasible closed tour that serves all vertices by traversing G's arcs and twice the arcs of $A(T)$ once in each direction. We conclude the paper with a proof that the above algorithm solves the swapping problem optimally.

THEOREM 4.9. *The swapping algorithm for the linear track case solves the swapping problem on a line optimally.*

Proof. Any feasible solution is the union of service paths for each object-type $i \in \{0, \dots, m\}$ and possibly some deadheadings; see Anily and Hassin (1992). As shown in Theorems 3.1 and 3.7, any feasible solution can be transformed into a feasible solution of the same cost, where the length of service paths of each object-type $i \in \{0, \dots, m\}$ is separately minimized and possibly some deadheadings. The minimum cost service paths are obtained by algorithm BASIC GRAPH G. A minimum cost set of deadheadings is found by applying the MDST procedure on T. Two copies of each such deadheading is added to the BASIC GRAPH G in order to preserve the final graph as Eulerian while making it connected.

REFERENCES

S. ANILY AND R. HASSIN (1992), *The swapping problem*, Networks, 22, pp. 419–433.

A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithm*, Addison–Wesley, Reading, MA.

M. J. ATALLAH AND S. R. KOSARAJU (1988), *Efficient solutions to some transportation problems with applications to minimizing robot arm travel*, SIAM J. Comput., 17, pp. 849–869.

F. BOCK (1971), *An algorithm to construct a minimum directed spanning tree in a directed network*, in Developments in Operations Research, Gordon and Breach, New York, pp. 29–44.

Y. J. CHU AND T. H. LIU (1965), *On the shortest arborescence of a directed graph*, Sci. Sinica, 14, pp. 1396–1400.

J. EDMONDS (1967), *Optimum branchings*, J. Res. Nat. Bur. Standards Sect. B, 71, pp. 233–240.

G. N. FREDERICKSON (1993), *A note on the complexity of a simple transportation problem*, SIAM J. Comput, 22, pp. 57–61.

G. N. FREDERICKSON AND D. J. GUAN (1992), *Preemptive ensemble motion planning on a tree*, SIAM J. Comput., 21, pp. 1130–1152.

G. N. FREDERICKSON AND D. J. GUAN (1993), *Nonpreemptive ensemble motion planning on a tree*, J. Algorithms, 15, pp. 29–60.

G. N. FREDERICKSON, M. S. HECHT, AND C. E. KIM (1978), *Approximation algorithms for some routing problems*, SIAM J. Comput., 7, pp. 178–193.

D. S. JOHNSON AND C. H. PAPADIMITRIOU (1985), *Performance guarantees for heuristics*, in The Traveling Salesman Problem, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds., John Wiley, New York, Chapter 5, pp. 145–180.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 1, pp. 336–350

HOW GOOD IS THE GOEMANS–WILLIAMSON MAX CUT ALGORITHM?[*]

HOWARD KARLOFF[†]

Abstract. The celebrated semidefinite programming algorithm for MAX CUT introduced by Goemans and Williamson was known to have a performance ratio of at least $\alpha = \frac{2}{\pi} \min_{0<\theta\leq\pi} \frac{\theta}{1-\cos\theta}$ ($0.87856 < \alpha < 0.87857$); the exact performance ratio was unknown. We prove that the performance ratio of their algorithm is exactly α. Furthermore, we show that it is impossible to add valid linear constraints to improve the performance ratio.

Key words. MAX CUT, semidefinite programming, approximation algorithm, optimization

AMS subject classifications. 05C85, 49M37, 68Q25, 68R05, 68R10

PII. S0097539797321481

1. Introduction.

The *performance ratio* of a randomized algorithm for a maximization problem is defined as follows. Let $W(I)$ be the (random) weight of the feasible solution it produces on instance I and let $OPT(I)$ be the weight of the optimal solution for instance I. The performance ratio is then

$$\inf \frac{E[W(I)]}{OPT(I)},$$

where the infimum is over instances with $OPT(I)$ positive.

In 1994, M. X. Goemans and D. P. Williamson published a new approximation algorithm for MAX CUT, the NP-complete problem of finding a maximum cut in a nonnegatively weighted graph [GW]. (Readers unfamiliar with [GW] should read the description of the algorithm in section 1.1 before continuing.) Much of the mathematical programming community turned its attention to this new algorithm, not only because it improved the performance ratio for MAX CUT from 0.5 to at least 0.87856, but also because Goemans and Williamson's paper was the first use of semidefinite programming in the area of approximation algorithms. Naturally researchers asked how far semidefinite programming could go.

In this paper, we analyze MAX CUT and the semidefinite programming algorithm proposed by Goemans and Williamson, hereafter called *Algorithm GW.* Goemans and Williamson proved a lower bound of $\alpha = \frac{2}{\pi} \min_{0<\theta\leq\pi} \frac{\theta}{1-\cos\theta}$ ($0.87856 < \alpha < 0.87857$) on its performance ratio, a huge improvement over the 0.5 known before. But exactly how good is Algorithm GW? At no point did they prove an upper bound on the performance ratio of Algorithm GW. Conceivably the performance ratio of Algorithm GW substantially exceeded α. We will prove, in fact, that its performance ratio is exactly α.

In addition, we study the effect of the addition of linear constraints to the semidefinite program of [GW]. For example, several authors have proposed adding to the

[*]Received by the editors May 14, 1997; accepted for publication (in revised form) October 23, 1998; published electronically September 22, 1999. This research was supported in part by NSF grant CCR-9319106. This paper appeared in the *Proceedings of the 28th ACM Symposium on the Theory of Computing*, Philadelphia, 1996.
http://www.siam.org/journals/sicomp/29-1/32148.html

[†]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280 (howard@cc.gatech.edu).

semidefinite program a family of $4\binom{n}{3}$ linear constraints, in the hope of narrowing the gap between the optimal value of the semidefinite program and the weight of a maximum cut. These constraints, given in section 2 and called the *triangle constraints*, are valid for all cuts (that is, when the vector variables are actually the *integers* ± 1). It is known [BM] that with the triangle constraints, the optimal value of the semidefinite program is exactly the weight of a maximum cut if the input graph does not contain K_5 as a minor (in particular, if the input graph is planar).

We will prove that the addition of any family of constraints, which, like the triangle constraints, are linear in the dot products and valid for all cuts, cannot improve the performance ratio; it will be exactly α, even with those constraints.

A caveat to the reader: We analyze the performance ratio of Algorithm GW, which randomly produces one cut. Should we instead run Algorithm GW many times and output the *best* of the random cuts produced, the current analysis says nothing about the performance ratio of the new algorithm.

In addition to the performance ratio, another interesting quantity to study is the "integrality ratio": the minimum, over all weighted graphs, of the ratio between the weight of a maximum cut and the optimal value of the semidefinite relaxation for that graph. Unlike the performance ratio, which is affected both by the relaxation and the rounding procedures, the integrality ratio depends, of course, only on the relaxation. Goemans and Williamson did study the integrality ratio for their semidefinite relaxation, noting that it is particularly bad for the cycle C_5. We caution the reader that the results in the present paper say nothing about the integrality ratio. In fact, for the graphs we will construct, the weight of a maximum cut exactly equals the optimal value of the semidefinite program.

1.1. A description of Algorithm GW.

Here we describe the algorithm of Goemans and Williamson. For more details, see [GW].

The Goemans and Williamson algorithm applies to MAX CUT, the problem of partitioning the vertices of a weighted graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, into two sets, so as to maximize the total weight of the edges across the cut. Let us use $w_{ij} \geq 0$ to denote the nonnegative weight of the edge between vertices i and j, with $w_{ij} = 0$ if $\{i, j\} \notin E$.

This problem can be rephrased as follows. Assign an integer $x_i \in \{-1, +1\}$ to each vertex i, -1 representing the left side of the cut and $+1$ the right side, so as to maximize the sum, over all $i < j$, of $w_{ij}(1 - x_i x_j)/2$. The reason this works is that $(1 - x_i x_j)/2$ is 1 if i and j are on opposite sides of the cut and 0 if they're on the same side. Since MAX CUT is NP-complete, clearly one cannot solve this problem exactly in polynomial time (unless P = NP). However, one can relax this program to a polynomial time-solvable one, as follows. Let v_i be an n-dimensional vector whose first component is x_i and whose others are 0. Clearly $||v_i|| = 1$ ($||v||$ being the Euclidean norm of v) and the objective function

$$\sum_{i<j} w_{ij} \frac{1 - x_i x_j}{2}$$

can be rewritten as

$$\sum_{i<j} w_{ij} \frac{1 - v_i \cdot v_j}{2}$$

for these vectors, where the usual multiplication of integers x_i and x_j has been replaced by the dot product of vectors v_i and $v_j \in \mathbb{R}^n$. Now consider the problem (SD): Find vectors $v_1, v_2, \ldots, v_n \in \mathbb{R}^n$ to

$$\max \sum_{i<j} w_{ij} \frac{1 - v_i \cdot v_j}{2}$$

and such that $||v_i|| = 1$ for all i.

Since the vectors v_i given above satisfy $||v_i|| = 1$, they are feasible for this program. This makes (SD) a relaxation of MAX CUT and thus the optimal value of this program is at least as large as the weight of a maximum cut in G.

Interestingly, (SD) can be (almost exactly) solved in polynomial time. Before we sketch how, let us describe what we can do with the solution $v_1, v_2, \ldots, v_n$, which we assume, for this summary, is the sequence of exactly optimal vectors. Somehow, we want to convert the sequence of n vectors into a cut (or into a sequence of n -1's and $+1$'s); that is, we want to *round* the vectors to an integral solution, so that the total weight of the edges across the cut is close to the value of the objective function in (SD).

The rounding procedure proposed by Goemans and Williamson is remarkably simple: just choose a random hyperplane through the origin, putting all the vertices on one side of the hyperplane on the "left" side L of the cut and the remainder on the "right" side R. More formally, choose a random unit vector r (the normal to the random hyperplane) and define $L = \{i | r \cdot v_i \leq 0\}$ and $R = \{i | r \cdot v_i > 0\}$. What is the expected weight of this cut? By linearity of expectation, the expected weight is the sum, over all $i < j$, of w_{ij} times the probability that i and j are on opposite sides of the cut. It is not hard to prove that the probability that i and j are on opposite sides of the cut is proportional to the angle $\arccos(v_i \cdot v_j)$ between v_i and v_j; in fact, the probability is exactly $\arccos(v_i \cdot v_j)/\pi$. Thus the expected value $E[W]$ of the weight W of the random cut is exactly

$$\sum_{i<j} w_{ij} \frac{\arccos(v_i \cdot v_j)}{\pi}.$$

However, the optimal value z_P^* of program (SD) (it does have an optimum) is exactly

$$\sum_{i<j} w_{ij} \frac{1 - v_i \cdot v_j}{2}.$$

Thus, by a term-by-term analysis, we have

$$\begin{aligned} \frac{E[W]}{z_P^*} &= \frac{\sum_{i<j} w_{ij} \arccos(v_i \cdot v_j)/\pi}{\sum_{i<j} w_{ij}(1 - v_i \cdot v_j)/2} \\ &\geq \min_{i<j} \frac{\arccos(v_i \cdot v_j)/\pi}{(1 - v_i \cdot v_j)/2}. \end{aligned}$$

Letting

$$\alpha = \frac{2}{\pi} \min_{0<\theta\leq\pi} \frac{\theta}{1 - \cos\theta}$$

(θ representing $\arccos(v_i \cdot v_j)$), we have

$$E[W] \geq \alpha \cdot z_P^*,$$

and, since z_P^* is at least as large as the weight OPT of a maximum cut, we have

$$E[W] \geq \alpha \cdot OPT.$$

Fortunately, $\alpha \in (0.87856, 0.87857)$. It follows that Algorithm GW is an α-approximation algorithm for MAX CUT.

Now we return to semidefinite programming. To understand semidefinite programming, one must understand what it means for an $n \times n$ symmetric matrix Y to be positive semidefinite. There are three equivalent definitions:

1. All the eigenvalues of Y are nonnegative. (Recall that the eigenvalues of any symmetric matrix are real.)
2. For any $x \in \mathbb{R}^n$, $x^T Y x \geq 0$.
3. There is an $n \times n$ matrix B such that $Y = B^T B$.

A *semidefinite program* is an optimization problem of the following sort. Find reals $y_{11}, y_{12}, \ldots, y_{nn}$ so as to maximize $c^T y$ (where $y = (y_{11}, y_{12}, y_{13}, \ldots, y_{nn})^T$) subject to a finite number of linear constraints $a_i^T y \geq b_i$ and such that the $n \times n$ matrix $Y = (y_{ij})$ is symmetric and positive semidefinite. The only differences between semidefinite programming and linear programming are the immaterial difference that the number of variables is a perfect square and the crucial difference that the matrix defined by the variables must be symmetric and positive semidefinite.

Semidefinite programs can be (almost exactly) solved in polynomial time; for more details, see [GW]. How does this help us? By the third definition of "positive semidefinite" we see that if Y is symmetric and positive semidefinite, then there is an $n \times n$ B such that $Y = B^T B$, and if B is an $n \times n$ matrix, then $Y = B^T B$ is positive semidefinite. To solve program (SD) on variables $v_1, v_2, \ldots, v_n \in \mathbb{R}^n$, we instead solve the following problem: Find a symmetric positive semidefinite matrix $Y = (y_{ij})$ to

$$\max \sum_{i<j} w_{ij} \frac{1 - y_{ij}}{2}$$

such that $y_{ii} = 1$ for all i.

Given Y, we can find a B such that $Y = B^T B$ in polynomial time. Letting v_i be the ith column of B, we have $v_i \cdot v_j = y_{ij}$ and $||v_i|| = 1$, exactly what we wanted. Conversely, given vectors $v_1, v_2, \ldots, v_n \in \mathbb{R}^n$, each of Euclidean length 1, we can build a matrix B whose ith column is v_i and then let $Y = B^T B$; we will have Y symmetric and positive semidefinite, $v_i \cdot v_j = y_{ij}$ and $y_{ii} = 1$ again. Thus, program (SD) can be phrased as a semidefinite program and thus (almost exactly) solved in polynomial time.

2. Bounds on the performance ratio. The troublesome graphs for the Goemans–Williamson algorithm are the graphs $J(m, t, b)$ defined as follows. The vertex set of $J(m, t, b)$ is the set of all $\binom{m}{t}$ t-element subsets of $\{1, 2, \ldots, m\}$, two t-subsets S, T being adjacent if and only if $|S \cap T| = b$ and $S \neq T$. (These graphs are similar to the Kneser graphs $K(m, t, b)$ used in [KMS], which have the same vertex set but with S and T adjacent if and only if $|S \cap T| \leq b$.) It will become clear that the smallest eigenvalue of the adjacency matrix of $J(m, m/2, b)$ plays a crucial role, so we now study the eigenvalues of $J(m, t, b)$. For $s = 0, 1, 2, \ldots, t$, let

$$\alpha_s = \sum_{r \geq 0} (-1)^{s-r} \binom{s}{r} \binom{t-r}{b-r} \binom{m-t-s+r}{t-b-s+r}$$

with the usual convention that $\binom{u}{v} = 0$ if $v < 0$.

THEOREM 2.1. *Let $0 \le t \le m/2$ and $0 \le b \le t$. Then for $s = 0, 1, 2, \dots, t$, α_s occurs as an eigenvalue of $J(m, t, b)$ with multiplicity $\binom{m}{s} - \binom{m}{s-1}$.*

See [Knuth] for a beautiful proof of this theorem, or see [Delsarte]. Notice that since $\sum_{s=0}^{t}[\binom{m}{s} - \binom{m}{s-1}] = \binom{m}{t}$, these are all the eigenvalues of $J(m, t, b)$.

In fact, we will need only $J(m, m/2, b)$ for even positive m's.

COROLLARY 2.2. *If m is an even positive integer, $0 \le b \le m/2$, then the eigenvalues of $J(m, m/2, b)$ are $\beta_0, \beta_1, \dots, \beta_{m/2}$ (each with some positive multiplicity), where*

$$\beta_s = \sum_{r \ge 0} (-1)^{s-r} \binom{s}{r} \binom{m/2 - r}{b - r} \binom{m/2 - s + r}{b}.$$

The next theorem gives the exact value of the smallest eigenvalue of $J(m, m/2, b)$.

THEOREM 2.3. *Let m be an even positive integer and let $0 \le b \le m/12$. The smallest eigenvalue of $J(m, m/2, b)$ is*

$$\binom{m/2}{b}^2 \left[\frac{4b}{m} - 1\right].$$

We defer the proof of Theorem 2.3. For now, let us say only that a simple calculation gives $\beta_1 = \binom{m/2}{b}^2 \left[\frac{4b}{m} - 1\right]$; thus proving Theorem 2.3 is the same as proving that β_1 is the smallest of all the β_s's.

The Goemans–Williamson semidefinite program to approximately solve the MAX CUT instance with edge weight w_{ij} between edges i and j ($w_{ij} = w_{ji}$ for all i, j and $w_{ii} = 0$ for all i) is, as mentioned above, program (SD): Find reals y_{ij} to

$$\max z_P = \sum_{i<j} w_{ij} \frac{1 - y_{ij}}{2},$$

subject to $y_{ii} = 1$ for all i and $Y = (y_{ij})$ is $n \times n$ and symmetric positive semidefinite.

In our case, we use $w_{ij} = 1$ if the ith set S is adjacent in $J(m, m/2, b)$ to the jth set T (and $i \ne j$), and 0 otherwise. Alternatively, $w_{S,T} = 1$ if $|S \cap T| = b$ (and $S \ne T$), and 0 otherwise. Thus the matrix of weights is exactly the adjacency matrix of $J(m, m/2, b)$.

Each vertex in $J(m, m/2, b)$, being a subset S of $\{1, 2, \dots, m\}$ of weight $m/2$, can be represented by an m-vector of $+1$'s and -1's, with the ith component $+1$ if $i \in S$ and the ith component -1 if $i \notin S$, then scaled by $1/\sqrt{m}$, so that the resulting m-vector v_S has unit length. Now, from the v_S's, build an $m \times n$ matrix B with the column indexed by S being the vector v_S; then set $Y = B^T B$. Y is symmetric positive semidefinite, is $n \times n$, and has $y_{ii} = 1$ for all i. Therefore Y is a feasible solution to (SD).

In fact, Y is an *optimal* solution to (SD), if $b \le m/12$.

THEOREM 2.4. *If $b \le m/12$ and m is even, then $Y = B^T B$ is an optimal solution to (SD).*

Proof. To show that a feasible solution to a semidefinite program is optimal, we use the dual of the semidefinite program [GW]. Let $A = (w_{ij})$ be the matrix of weights on the edges of the n-vertex graph. Let $W_{tot} = \sum_{1 \le i < j \le n} w_{ij}$. The dual to the semidefinite program (SD) is the semidefinite program (D), as follows. Find reals

$\gamma_1, \gamma_2, \ldots, \gamma_n$ to

$$\min z_D = \frac{1}{2} W_{tot} + \frac{1}{4} \sum_{i=1}^{n} \gamma_i,$$

such that

$$A + \operatorname{diag}(\gamma_1, \gamma_2, \ldots, \gamma_n)$$

is positive semidefinite, where $\operatorname{diag}(\gamma_1, \gamma_2, \ldots, \gamma_n)$ is the diagonal matrix whose ith diagonal entry is γ_i. It is known [GW] that $z_P \leq z_D$ for any pair of respectively (primal, dual) feasible solutions. In fact, (SD) and (D) attain optimal values z_P^* and z_D^*, respectively, and $z_P^* = z_D^*$.

Therefore, to prove that a particular feasible solution Y for (SD) is optimal, it suffices to exhibit a feasible solution $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ to (D) such that $z_P = z_D$. Let γ be the negation of the smallest eigenvalue of $J(m, m/2, b)$. By Theorem 2.3, $\gamma = \binom{m/2}{b}^2 \left[1 - \frac{4b}{m}\right]$. A symmetric matrix being positive semidefinite if and only if its eigenvalues are nonnegative, the choice of γ guarantees that $\gamma_i = \gamma$ for all i is a feasible solution to (D).

Let z_P be the (primal) cost of the primal feasible Y given above and let z_D be the (dual) cost of dual feasible $(\gamma, \gamma, \gamma, \ldots, \gamma)$. We prove that $z_P = z_D$, so that both are optimal.

In this formulation,

$$\begin{aligned} z_P &= \frac{1}{4} \sum_{|S|=|T|=m/2} w_{S,T}(1 - v_S \cdot v_T) \\ &= \frac{1}{4} \sum_{|S|=|T|=m/2, |S \cap T|=b} (1 - v_S \cdot v_T). \end{aligned}$$

By symmetry, we can take $S = \{1, 2, \ldots, m/2\}$ and infer that

$$\begin{aligned} z_P &= \frac{1}{4} n \sum \left[1 - \frac{1}{\sqrt{m}} (\underbrace{1, 1, \ldots, 1}_{m/2}, \underbrace{-1, -1, \ldots, -1}_{m/2}) \cdot v_T \right] \\ &= \frac{1}{4} n \left[\binom{m/2}{b}^2 - \frac{1}{\sqrt{m}} \sum (\underbrace{1, \ldots, 1}_{m/2}, \underbrace{-1, \ldots, -1}_{m/2}) \cdot v_T \right] \end{aligned}$$

(where both summations are over all T such that $|T| = m/2$ and $|T \cap \{1, 2, \ldots, m/2\}| = b$)

$$\begin{aligned} &= \frac{1}{4} n \left[\binom{m/2}{b}^2 - \frac{1}{\sqrt{m}} \binom{m/2}{b}^2 \frac{1}{\sqrt{m}} (4b - m) \right] \\ &= \frac{n}{2} \binom{m/2}{b}^2 \left[1 - \frac{2b}{m} \right]. \end{aligned}$$

Also,

$$z_D = \frac{1}{2} W_{tot} + \frac{1}{4} \sum_{i=1}^{n} \gamma_i$$

$$= \frac{1}{4}n\binom{m/2}{b}^2 + \frac{1}{4}n\gamma$$

$$= \frac{n}{4}\binom{m/2}{b}^2 + \frac{n}{4}\binom{m/2}{b}^2\left[1 - \frac{4b}{m}\right]$$

$$= \frac{n}{2}\binom{m/2}{b}^2\left[1 - \frac{2b}{m}\right].$$

It follows that $z_P = z_D$ and that, by the discussion earlier in this proof, both are optimal. $\square$

Now let us study in more detail the optimal system of vectors v_S given above. Define $y_S = v_S\sqrt{m}$. Since the lth component of v_S is $\pm 1/\sqrt{m}$, each component of y_S is ± 1. Let us define a sequence of cuts $\mathcal{C}_l = (\mathcal{S}_l, \bar{\mathcal{S}}_l)$, $1 \le l \le m$, with $\mathcal{S}_l = \{S | S \ni l, |S| = m/2\}$. Letting $(y_S)_l$ denote the lth component of vector y_S, notice that for all S

$$S \in \mathcal{S}_l \iff l \in S \iff (y_S)_l = +1.$$

When we solve the semidefinite program, we hope that the optimal vectors have the following property: There is an l such that each vector is ± 1 in component l, and 0 elsewhere; such vectors correspond naturally to cuts, and any solution of that form would be a maximum cut, since its weight would equal that of the optimum of the semidefinite program. Of course, the vectors y_S given above have ± 1 in *every* coordinate. This means that each vector v_S is the sum, over $l = 1, 2, 3, \ldots, m$, of a vector which is ± 1 in component l and 0 elsewhere (and then scaled down to unit length). The optimal family (v_S) can be viewed as an "average" of families of vectors representing the cuts $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$. This motivates the following.

LEMMA 2.5. *If $b \le m/12$, then the optimal value z_P^* of the semidefinite program is the average of the weights of cuts $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$.*

Proof. Where $(v_S)_l$ and $(y_S)_l$ denote the lth coordinates of v_S and y_S, respectively, we have

$$z_P^* = \frac{1}{2}\sum_{S,T} w_{S,T}\frac{1 - v_S \cdot v_T}{2}$$

$$= \frac{1}{2}\sum_{S,T} w_{S,T}\frac{1}{m}\sum_{l=1}^{m}\left[\frac{1 - m(v_S)_l(v_T)_l}{2}\right]$$

$$= \frac{1}{m}\frac{1}{2}\sum_{S,T} w_{S,T}\sum_{l=1}^{m}\left[\frac{1 - (y_S)_l(y_T)_l}{2}\right]$$

$$= \frac{1}{m}\sum_{l=1}^{m}\left[\frac{1}{2}\sum_{S,T} w_{S,T}\left[\frac{1 - (y_S)_l(y_T)_l}{2}\right]\right].$$

Since the weight of cut $\mathcal{C}_l$ is exactly

$$\frac{1}{2}\sum_{S,T} w_{S,T}\left[\frac{1 - (y_S)_l(y_T)_l}{2}\right],$$

we are done. $\square$

LEMMA 2.6. *Each cut $\mathcal{C}_l$ is a maximum cut (if $b \le m/12$).*

Proof. Because (SD) is a relaxation of MAX CUT, z_P^* is an upper bound on the weight of any cut. If any cut $\mathcal{C}_l$ were not a maximum cut, its weight would be strictly less than z_P^* and so would the average weight of these cuts. $\square$

We have proven the following theorem.

THEOREM 2.7. *The weight of a maximum cut in $J(m, m/2, b)$ is exactly*

$$z_P^* = \frac{n}{2}\binom{m/2}{b}^2\left[1 - \frac{2b}{m}\right]$$

if m is even and $b \le m/12$.

Now we study what happens when additional constraints are added to the semidefinite program. The results on this subject were obtained in collaboration with Uri Feige, who showed how the author's original argument, which applied only to the triangle constraints, could be generalized.

Where a_{ij}, $1 \le i < j \le n$, and b are reals, let us call a constraint

$$\sum_{i<j} a_{ij}(z_i \cdot z_j) \ge b$$

valid if it is satisfied whenever each z_i is the *integer* ± 1. (Whether a linear constraint is valid doesn't depend, of course, on the w_{ij}'s.)

If

$$b' = \frac{-b + \sum_{i<j} a_{ij}}{2},$$

then by a simple linear transformation,

$$\sum_{i<j} a_{ij}(z_i \cdot z_j) \ge b \iff \sum_{i<j} a_{ij}\frac{1 - z_i \cdot z_j}{2} \le b'.$$

Now build a complete graph H_a on $\{1, 2, \ldots, n\}$ with the (possibly negative) weight of edge $\{i, j\}$ being a_{ij}. Validity of the constraint is equivalent to saying that the weight of a maximum cut in H_a is at most b' (where we are including the trivial cut with one empty side as a cut).

Now suppose that $n = \binom{m}{m/2}$ for some positive even m. Label the vectors v_S above as $v_1, v_2, \ldots, v_n$ arbitrarily. Define $y_i = v_i\sqrt{m}$.

THEOREM 2.8. *Let a_{ij}, for all vertices $1 \le i < j \le n$, and b be reals such that the constraint*

$$\sum_{i<j} a_{ij}(z_i \cdot z_j) \ge b$$

is valid. Then the constraint is necessarily satisfied when $z_i = v_i$ for all i.

Proof.

$$\begin{aligned}\sum_{i<j} a_{ij}(v_i \cdot v_j) &= \sum_{i<j} a_{ij}\sum_{l=1}^{m}(v_i)_l(v_j)_l \\ &= \frac{1}{m}\sum_{l=1}^{m}\left(\sum_{i<j} a_{ij}(y_i)_l(y_j)_l\right).\end{aligned}$$

Clearly $(y_i)_l \in \{-1, +1\}$ for all vertices i. Therefore

$$\sum_{i<j} a_{ij}(y_i)_l(y_j)_l \geq b.$$

Hence

$$\frac{1}{m}\sum_{l=1}^{m}\sum_{i<j} a_{ij}(y_i)_l(y_j)_l \geq \frac{1}{m}\sum_{l=1}^{m} b = b. \qquad \square$$

Theorem 2.8 has the following consequence. Since the vectors $v_1, \ldots, v_n$ satisfy all valid constraints and happen to be optimal for the semidefinite program (SD) associated with $J(m, m/2, b)$ (at least if $0 \leq b \leq m/12$), they are obviously optimal in the augmented program for $J(m, m/2, b)$ (if $b \leq m/12$). Of course, the procedure that rounds the v_i's will proceed as before. However badly the algorithm rounded the vectors (v_i) beforehand, it will round them just as badly afterward.

For example, consider the triangle constraints (see [FG, GW] as well as [DL, Chapter 27] and the references therein). For each triple (i, j, k) of vertices, $1 \leq i < j < k \leq n$,

$$\begin{aligned}
+z_i \cdot z_j + z_i \cdot z_k + z_j \cdot z_k &\geq -1,\\
-z_i \cdot z_j - z_i \cdot z_k + z_j \cdot z_k &\geq -1,\\
-z_i \cdot z_j + z_i \cdot z_k - z_j \cdot z_k &\geq -1,\\
+z_i \cdot z_j - z_i \cdot z_k - z_j \cdot z_k &\geq -1.
\end{aligned}$$

The second, third, and fourth constraints in each block of four are obtained from the first one by negating z_i, z_j, or z_k, respectively. The reader can verify that these four constraints are valid. (The key is that given any three integers in $\{-1, +1\}$, not all three pairwise products can be -1.) It follows from Theorem 2.8, if $n = \binom{m}{m/2}$, that the vectors $z_i = v_i$ satisfy these vector constraints.

For another example, suppose we augment (SD) by adding all infinitely many possible valid constraints. That is, for each $a = (a_{ij})$, $1 \leq i < j \leq n$, we build a weighted complete graph H_a on $\{1, 2, \ldots, n\}$, with weight a_{ij} on edge $\{i, j\}$. We then calculate the maximum weight b' of a (possibly trivial) cut in H_a and add a constraint

$$\sum_{i<j} a_{ij}\frac{1 - v_i \cdot v_j}{2} \leq b'.$$

This constraint is evidently valid, and furthermore, by looking at all a's, we are including all possible valid constraints (except that we are using the best possible right-hand side for each a). By Theorem 2.8, if $n = \binom{m}{m/2}$, then the vectors $v_1, v_2, \ldots, v_n$ will satisfy all the infinitely many additional constraints (of which the triangle constraints are but a subset) and, being optimal for (SD) for $J(m, m/2, b)$, will still be optimal afterward (if $b \leq m/12$).

Now let us calculate just how badly Algorithm GW performs on $J(m, m/2, b)$, keeping in mind that the addition of more valid inequalities would not improve the outcome.

LEMMA 2.9. *The expected value of the weight W of the random cut produced from Y and the v_S's is exactly*

$$\frac{n}{2}\binom{m/2}{b}^2 \frac{1}{\pi}\arccos\left[\frac{4b}{m} - 1\right].$$

Proof. The expected weight of the cut is

$$E[W] = \frac{1}{2} \sum_{S,T:|S|=|T|=m/2} w_{S,T} \cdot \arccos(v_S \cdot v_T)/\pi$$
$$= \frac{1}{2} \sum_{S,T:|S|=|T|=m/2,|S\cap T|=b} \arccos(v_S \cdot v_T)/\pi.$$

By symmetry, we can take $S = \{1, 2, \ldots, m/2\}$ and get

$$E[W] = \frac{1}{2} n \sum \frac{1}{\pi} \arccos\left(v_S \cdot v_T\right),$$

where the summation is over all T such that $|T| = m/2$ and $|T \cap \{1, 2, \ldots, m/2\}| = b$. Thus

$$E[W] = \frac{n}{2} \binom{m/2}{b}^2 \frac{1}{\pi} \arccos\left[\frac{4b-m}{m}\right]. \qquad \square$$

Since we know that a maximum cut has weight exactly

$$\frac{n}{2} \binom{m/2}{b}^2 \left[1 - \frac{2b}{m}\right]$$

if $b \le m/12$, by Lemma 2.9 the ratio of $E[W]$ to the weight of a maximum cut is exactly

$$\frac{\frac{1}{\pi} \arccos\left[\frac{4b}{m} - 1\right]}{1 - \frac{2b}{m}}.$$

Where $\theta = \arccos[\frac{4b}{m} - 1]$, the ratio is $\frac{2}{\pi} \frac{\theta}{1-\cos\theta}$. But $\alpha = \min_{0<\theta\le\pi} \frac{2}{\pi} \frac{\theta}{1-\cos\theta}$. By choosing b, m, and hence θ auspiciously, we will approach the α of [GW].

THEOREM 2.10. *For each $\epsilon > 0$, there are b and m, m even and positive and $0 \le b \le m/12$, such that the expected weight of the cut produced by Algorithm GW applied to $J(m, m/2, b)$ is at most $\alpha + \epsilon$ times the weight of a maximum cut.*

Proof. A simple calculation shows that the unique θ^* that achieves the minimum value in the right-hand side of $\alpha = \min_{0<\theta\le\pi} \frac{2}{\pi} \frac{\theta}{1-\cos\theta}$ corresponds to a value of b which is less than $m/12.5$ and hence is safely away from $m/12$. Because $\frac{2}{\pi} \frac{\theta}{1-\cos\theta}$ is continuous, we can find a $\delta > 0$ so that if $\theta \in [\theta^* - \delta, \theta^* + \delta]$, then $\frac{2}{\pi} \frac{\theta}{1-\cos\theta} \le \alpha + \epsilon$. We can certainly choose a rational $\gamma < \frac{1}{12.5}$ such that $\theta = \arccos(4\gamma - 1)$ lies in $[\theta^* - \delta, \theta^* + \delta]$. Then we choose m even such that $b = \gamma m$ is an integer. But $\frac{2}{\pi} \frac{\theta}{1-\cos\theta}$ is precisely the ratio between $E[W]$ and the weight of a maximum cut. $\square$

Last, we prove Theorem 2.3.

Proof. We must prove that the smallest eigenvalue of $J(m, m/2, b)$ is

$$\binom{m/2}{b}^2 \left[\frac{4b}{m} - 1\right],$$

provided that m is even and that $0 \le b \le m/12$. Recall from Corollary 2.2 that the eigenvalues of $J(m, m/2, b)$ are $\beta_0, \beta_1, \ldots, \beta_{m/2}$, where

$$\beta_s = \sum_r (-1)^{s-r} \binom{s}{r} \binom{m/2-r}{b-r} \binom{m/2-s+r}{b}.$$

Since

$$\beta_1 = \binom{m/2}{b}^2 \left[\frac{4b}{m} - 1\right],$$

we must show that $\beta_1 \le \beta_s$ for $s = 0, 1, 2, \ldots, m/2$.

Fix s, $0 \le s \le m/2$. Given r, let

$$t_r = \binom{m/2 - r}{b - r}\binom{m/2 - s + r}{b} \ge 0$$

so that

$$\beta_s = (-1)^s \sum_r (-1)^r \binom{s}{r} t_r.$$

If s is even, then

$$\beta_s \ge -\sum_{r \text{ odd}} \binom{s}{r} t_r,$$

and if s is odd, then

$$\beta_s \ge -\sum_{r \text{ even}} \binom{s}{r} t_r.$$

Let us assume that $b \le m/2 - s + r$ and that $b \ge r$, for otherwise $t_r = 0$. Then

$$\frac{t_r}{\binom{m/2}{b}^2} = \frac{(m/2 - r)!}{(m/2)!}\frac{(m/2 - s + r)!}{(m/2)!}\frac{b!}{(b-r)!}\frac{(m/2 - b)!}{(m/2 - s + r - b)!}. \tag{2.1}$$

The product of the first and third fractions equals

$$\left[\frac{b(b-1)(b-2)\cdots(b-r+1)}{(\frac{m}{2})(\frac{m}{2}-1)(\frac{m}{2}-2)\cdots(\frac{m}{2}-r+1)}\right], \tag{2.2}$$

while the product of the second and fourth equals

$$\left[\frac{(\frac{m}{2}-b)(\frac{m}{2}-b-1)(\frac{m}{2}-b-2)\cdots(\frac{m}{2}-s+r-b+1)}{(\frac{m}{2})(\frac{m}{2}-1)(\frac{m}{2}-2)\cdots(\frac{m}{2}-s+r+1)}\right]. \tag{2.3}$$

It is easy to verify that for $0 \le i < m/2$,

$$\frac{b-i}{\frac{m}{2}-i} \le \frac{b}{\frac{m}{2}}$$

and

$$\frac{\frac{m}{2}-b-i}{\frac{m}{2}-i} \le \frac{\frac{m}{2}-b}{\frac{m}{2}}.$$

Therefore

$$\frac{t_r}{\binom{m/2}{b}^2} \le \left(\frac{2b}{m}\right)^r \left(1 - \frac{2b}{m}\right)^{s-r}.$$

This is true whether $b \le m/2 - s + r$ and $b \ge r$ or not. Let $\gamma = b/m \le 1/12$. Then

$$\frac{t_r}{\binom{m/2}{b}^2} \le (2\gamma)^r (1-2\gamma)^{s-r}.$$

For any p, let

$$x = \sum_{\text{odd } r} \binom{s}{r} p^r (1-p)^{s-r}$$

and

$$y = \sum_{\text{even } r} \binom{s}{r} p^r (1-p)^{s-r}.$$

Then $x + y = 1$ and $-x + y = (-p + (1-p))^s = (1-2p)^s$. It follows that

$$y = 1/2 + (1/2)(1-2p)^s$$

and

$$x = 1/2 - (1/2)(1-2p)^s.$$

Hence

$$\sum_{\text{odd } r} \binom{s}{r} (2\gamma)^r (1-2\gamma)^{s-r} = 1/2 - (1/2)(1-4\gamma)^s$$

and

$$\sum_{\text{even } r} \binom{s}{r} (2\gamma)^r (1-2\gamma)^{s-r} = 1/2 + (1/2)(1-4\gamma)^s.$$

Now

$$\frac{t_r}{\binom{m/2}{b}^2} \le (2\gamma)^r (1-2\gamma)^{s-r}.$$

So, if s is even,

$$\begin{aligned}
\beta_s &\ge -\left[\sum_{\text{odd } r} \binom{s}{r} (2\gamma)^r (1-2\gamma)^{s-r}\right] \binom{m/2}{b}^2 \\
&= -\left[\frac{1}{2} - \frac{1}{2}(1-4\gamma)^s\right] \binom{m/2}{b}^2 \\
&\ge -\left[\frac{1}{2} + \frac{1}{2}(1-4\gamma)^s\right] \binom{m/2}{b}^2;
\end{aligned}$$

if s is odd we have

$$\beta_s \ge -\left[\frac{1}{2} + \frac{1}{2}(1-4\gamma)^s\right] \binom{m/2}{b}^2.$$

So, in either case,

$$\beta_s \geq -\left[\frac{1}{2} + \frac{1}{2}(1-4\gamma)^s\right]\binom{m/2}{b}^2.$$

Now we need a technical fact.

FACT 2.11. *If $p \in [0, 1/3]$ and $s \geq 4$, then $(1-p)^s \leq 1-2p$.*

Proof (sketch). Let $f(p) = (1-p)^s/(1-2p)$ for $p \in [0, 1/3]$. Clearly $f(0) = 1$. It is not hard to see that $f'(p) \leq 0$ if and only if $p \leq \frac{s-2}{2s-2}$, and $s \geq 4$ implies that $(s-2)/(2s-2) \geq 1/3$. □

By the fact, because $\gamma \leq 1/12$, $(1-4\gamma)^s \leq 1-8\gamma$ if $s \geq 4$. Thus

$$\begin{aligned}\beta_s &\geq -\left[\frac{1}{2} + \frac{1}{2}(1-8\gamma)\right]\binom{m/2}{b}^2 \\ &= (4\gamma - 1)\binom{m/2}{b}^2 \\ &= \left[\frac{4b}{m} - 1\right]\binom{m/2}{b}^2,\end{aligned}$$

and we are finished, in the case $s \geq 4$.

Now we must dispense with $s = 0, 1, 2, 3$ by showing that $\beta_s \geq \beta_1 = \binom{m/2}{b}^2\left[\frac{4b}{m} - 1\right]$ for $s = 0, 1, 2, 3$. Notice that if $b = 0$, then for any even m, $\beta_s = (-1)^s$. This means that the smallest eigenvalue is -1, as it should be. In what follows, we assume that $m \geq 12$, for otherwise $b \leq m/12$ makes $b = 0$.

The $s = 0$ case is first. $\beta_0 = \binom{m/2}{b}^2$, the degree of a vertex in $J(m, m/2, b)$. Since this is positive and $\binom{m/2}{b}^2\left[\frac{4b}{m} - 1\right]$ is negative, this part is easy.

The $s = 1$ case is trivial.

The $s = 2$ case is still not hard.

$$\frac{\beta_2}{\binom{m/2}{b}^2} = \frac{(\frac{m}{2} - b)}{\frac{m}{2}}\frac{(\frac{m}{2} - b - 1)}{\frac{m}{2} - 1} - 2\frac{b}{\frac{m}{2}}\frac{(\frac{m}{2} - b)}{\frac{m}{2}} + \frac{b}{\frac{m}{2}}\frac{(b-1)}{(\frac{m}{2} - 1)}.$$

Since $\beta_1 = \binom{m/2}{b}^2\left[\frac{4b}{m} - 1\right] < 0$, it suffices to prove that

$$\frac{(\frac{m}{2} - b)}{\frac{m}{2}}\frac{(\frac{m}{2} - b - 1)}{\frac{m}{2} - 1} \geq 2\frac{b}{\frac{m}{2}}\frac{(\frac{m}{2} - b)}{\frac{m}{2}}.$$

That

$$\frac{\frac{m}{2} - b - 1}{\frac{m}{2} - 1} \geq \frac{2b}{\frac{m}{2}}$$

holds follows from $b \leq m/12$ and $m \geq 12$. This means that

$$\frac{(\frac{m}{2} - b)}{\frac{m}{2}}\frac{(\frac{m}{2} - b - 1)}{\frac{m}{2} - 1} \geq 2\frac{b}{\frac{m}{2}}\frac{(\frac{m}{2} - b)}{\frac{m}{2}},$$

and the proof of the case $s = 2$ is complete.

Now we attack $s = 3$. Since $t_r \geq 0$ for all r and

$$\beta_3 = -\sum_{r=0}^{3}(-1)^r\binom{3}{r}t_r,$$

the only possibly negative terms are the ones having $r = 0$ or $r = 2$. Using the formula for $t_r/\binom{m/2}{b}^2$ from (2.1) and using (2.2) and (2.3), it suffices to prove that

$$-\frac{(\frac{m}{2}-b)}{(\frac{m}{2})}\frac{(\frac{m}{2}-b-1)}{(\frac{m}{2}-1)}\frac{(\frac{m}{2}-b-2)}{(\frac{m}{2}-2)} - 3\frac{b}{\frac{m}{2}}\frac{(b-1)}{(\frac{m}{2}-1)}\frac{(\frac{m}{2}-b)}{\frac{m}{2}} \geq \frac{4b}{m} - 1.$$

Simple calculations yield that

$$-\left(\frac{\frac{m}{2}-b}{\frac{m}{2}}\right)^3 \leq -\frac{(\frac{m}{2}-b)}{\frac{m}{2}}\frac{(\frac{m}{2}-b-1)}{\frac{m}{2}-1}\frac{(\frac{m}{2}-b-2)}{\frac{m}{2}-2}$$

and that

$$-3\frac{b}{\frac{m}{2}}\frac{b}{\frac{m}{2}}\frac{(\frac{m}{2}-b)}{\frac{m}{2}} \leq -3\frac{b}{\frac{m}{2}}\frac{(b-1)}{(\frac{m}{2}-1)}\frac{(\frac{m}{2}-b)}{\frac{m}{2}}.$$

This means that it is sufficient to prove that

$$-\left(\frac{\frac{m}{2}-b}{\frac{m}{2}}\right)^3 - 3\left(\frac{b}{\frac{m}{2}}\right)^2\frac{\frac{m}{2}-b}{\frac{m}{2}} \geq \frac{4b}{m} - 1.$$

Let $\gamma = b/m$. The left-hand side equals

$$-(1-2\gamma)^3 - 3(2\gamma)^2(1-2\gamma) = -1 + 4\gamma - 16\gamma^2 + 2\gamma - 8\gamma^2 + 32\gamma^3.$$

Now

$$-1 + 6\gamma - 24\gamma^2 + 32\gamma^3 \geq 4\gamma - 1 \iff (2\gamma)(16\gamma^2 - 12\gamma + 1) \geq 0.$$

The roots of $16\gamma^2 - 12\gamma + 1 = 0$ are $(3 \pm \sqrt{5})/8$. Since $(3-\sqrt{5})/8 > 1/12 \geq \gamma \geq 0$, we are finished. $\square$

Although we don't need it, this paper wouldn't be complete without the following conjecture.

CONJECTURE 2.12. *For all even $m > 0$, for all $0 \leq b < m/4$, the smallest eigenvalue of $J(m, m/2, b)$ is exactly*

$$\beta_1 = \binom{m/2}{b}^2\left[\frac{4b}{m} - 1\right].$$

Possibly this is already known.

Acknowledgments. I am deeply grateful to David Williamson. David originally suggested the problem of analyzing the effect of the triangle constraints on the performance of Algorithm GW. And without David's help in finding a maximum cut and solving (SD) for a particular graph, this paper wouldn't exist. I thank him also for his comments on an earlier draft of this paper.

For the far more general result on the power of additional inequalities than what appeared in an earlier draft, for allowing me to include the result, and for his comments on an earlier draft, I am very thankful to Uri Feige.

For his pointers to the literature on algebraic graph theory, I am grateful to Neil Calkin. I thank Leonard Schulman and Cristina Fernandes for their help in studying the eigenvalues and eigenvectors of the $J(m, t, b)$'s. And I am thankful to Don Knuth for writing his notes on combinatorial matrices.

REFERENCES

[BM] F. Barahona and A. R. Mahjoub, *On the cut polytope*, Math. Programming, 36 (1986), pp. 157–173.

[Delsarte] P. Delsarte, *Hahn polynomials, discrete harmonics, and t-designs*, SIAM J. Appl. Math., 34 (1978), pp. 157–166.

[DL] M. M. Deza and M. Laurent, *Geometry of Cuts and Metrics*, Springer-Verlag, New York, 1997.

[FG] U. Feige and M. X. Goemans, *Approximating the value of two-prover proof systems, with applications to MAX 2SAT and MAX DICUT*, in Proceedings, Third Israel Symposium on the Theory of Computing and Systems, Tel-Aviv, Israel, 1995.

[GW] M. X. Goemans and D. P. Williamson, *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*, J. ACM, 42 (1995), pp. 1115–1145.

[KMS] D. Karger, R. Motwani, and M. Sudan, *Approximate graph coloring by semidefinite programming*, in Proceedings 35th ACM Symposium on the Foundations of Computer Science, 1994, Santa Fe, NM, pp. 2–13.

[K] H. Karloff, *How good is the Goemans-Williamson MAX CUT algorithm?*, in Proceedings 28th ACM Symposium on the Theory of Computing, 1996, Philadelphia, pp. 427–434.

[Knuth] D. Knuth, *Combinatorial Matrices*, Notes, Institut Mittag-Leffler, 1991, revised 1993.

SIAM J. Comput.
Vol. 29, No. 2, pp. 351–377

SOLVABILITY IN ASYNCHRONOUS ENVIRONMENTS II: FINITE INTERACTIVE TASKS[*]

BENNY CHOR[†] AND LEE-BATH NELSON[‡]

Abstract. Identifying what problems can be solved in a given distributed system is a central question in distributed computing. In this series of works, we study this question in the context of asynchronous fault tolerant systems that can execute consensus. These systems can be those executing deterministic protocols with access to a consensus routine or those running randomized error-free protocols. A previous work handled the class of distributed decision tasks. In these tasks, each processor receives one local input and has to respond with one local output.

In an *interactive distributed task* each of n processors receives a sequence of local inputs and has to respond *on line* with an output for every new input (before getting its next input). Different processors can be at different stages concurrently, so that additional inputs are received by fast processors while slow processors are still working on early inputs. An interactive task is called finite if the number of local inputs (and outputs) is finite. Interactive tasks can neither be described as a single huge decision task nor be decomposed into distinct, independent decision tasks.

The main result of this work is an exact characterization of the finite interactive tasks which can be solved by t-resilient protocols in either of the above two models. The major tool we use in the characterization is a directed acyclic graph that is associated with an interactive task. Properties of this graph are used to determine the resiliency of the task and to devise a "generic" resilient algorithm which solves such tasks. This generic algorithm can be viewed as a repeated, deterministic reduction to a consensus subroutine. This implies that any finite interactive task is solvable by randomized error-free protocols iff it is solvable by deterministic protocols with access to consensus.

Key words. solvability, asynchronous distributed systems, fault tolerance, randomized algorithms, consensus, interactive tasks, decision tasks, adversary scheduler

AMS subject classifications. 68Q22, 90D06, 90D43

PII. S0097539795294979

1. Introduction.

1.1. Background.

A central question in distributed computing is identifying what problems can be solved by a given distributed system. In typical systems, each one of n processors starts with some local input and communicates with other processors in order to produce globally meaningful outputs. If the system is perfect, in the sense that all processors are reliable and communication is error-free and is instantaneously relayed, then every well-defined task can be solved (assuming, as usual, that the processors are not computationally limited). However, perfect systems are either rare or nonexistent. Communication links tend to introduce errors and delays. Processors may become slow, stop operating, or even exhibit malevolent behavior.

One of the more popular models of distributed computing is the asynchronous crash failure model. Here, processors may crash without supplying any warning be-

[*]Received by the editors November 20, 1995; accepted for publication (in revised form) June 26, 1998; published electronically October 5, 1999. This research was supported by US–Israel BSF grant 88-00282. A preliminary version of this paper [14] appeared in *Proceedings of the* 10*th ACM Symposium on Principles of Distributed Computing*, 1991, pp. 37–49.
http://www.siam.org/journals/sicomp/29-2/29497.html

[†]Institute of Fundamental Sciences, Private Bag 11-222, Massey University, Palmerston North, New Zealand (B.Chor@massey.ac.nz).

[‡]Graduate School of Business, Stanford University, Palo Alto, CA 94305 (leen@leland.stanford.edu). The work of this author was done while at the Department of Computer Science, Technion, Haifa, Israel.

forehand. Asynchrony implies that there is no way to distinguish a failed processor from a very slow one. An alternative way to describe this is that there is an adversary scheduler that decides which processor moves when. Thus the scheduler controls the pace and failure of processors. A task is called *t-resilient* if it is solvable by a protocol, withstanding up to t processor crash failures. In addition to the important fault-tolerance aspect, crash resilient protocols have other merits: the more resilient the protocol, the less faster processors are delayed by waiting for slower ones. In particular, in a system with n processors, $n-1$ resilient protocols are *wait-free*—every processor can run at its maximum speed [19, 17].

A key result in this area was given by Fischer, Lynch, and Paterson [16], who have shown that the consensus problem is not even 1-resilient in the message passing model with respect to *deterministic* protocols. This result has been extended to the shared memory model [15, 21, 11]. (The conference version of the last reference proves this explicitly for a system of two processors.) This impossibility has motivated the study of *randomized* consensus protocols. A host of consensus protocols for various types of adversaries has been found for both the message passing model (e.g., [7, 24, 12, 6]) and the shared memory model (e.g., [11, 2, 1, 9, 3]). In particular, consensus has efficient *wait-free* (i.e., $n-1$ resilient) solutions in the shared memory model and $\lfloor \frac{n-1}{2} \rfloor$-resilient solutions in the message-passing model, even in the presence of a "strong adversary" [2, 6].

Consensus is an important problem, and in a certain sense it is a complete task, as follows from our results. Still, in order to understand the power and limitations of error-free randomization with respect to arbitrary distributed tasks, consensus alone does not suffice. To clarify this point, it is helpful to compare consensus with the *parity* task. In this latter task, each processor is required to output the XOR (parity) of all n inputs. Error-free randomization is powerful enough to overcome the coordination problems which prevent a deterministic solution to consensus, but it is of no help when facing problems of missing information. In the parity task, any irrevocable output by one processor (before knowing the inputs of all other $n-1$ processors) may later turn out to be inconsistent with additional inputs. Our work suggests a formal framework to capture these notions of missing and consistent/inconsistent inputs. Another aspect of fault tolerant models that is exemplified in this paper is that it is important for the adversary to have adequate powers. In particular, the adversary needs to be able to "resurrect" a processor, thereby turning a processor that previously may have appeared faulty into a merely slow one. Also, the adversary must be able to fail a processor that was previously active.

We study asynchronous fault tolerant systems that execute protocols of either of the following two types:

- deterministic protocols with access to consensus,
- randomized error-free protocols.

While we present most of our results in terms of *randomized error-free protocols*, the proof implies that in fact any finite interactive task is t-resilient in one model iff it is t-resilient in the other. In terms of what can and cannot be solved, it suffices to restrict the use of randomization to consensus. Thus if a consensus mechanism is built in, no randomization at all is needed. (Of course, randomization might still be used to speed up computations.)

This paper is the second in a series of three papers which study solvability by randomized error-free protocols, operating in asynchronous crash failure environments. The first paper in this series [13] deals with distributed decision tasks. The characterization there (as well as in the present work) is of combinatorial nature. Despite the added power of randomization, the characterizations and their proofs are fairly simple and yield effective procedures for testing solvability. This stands in sharp contrast to the recent works on deterministic solvability [8, 18, 27], which develop a methodology for characterizing decision tasks that are t-resilient with respect to deterministic protocols (*without* built-in consensus). These works are fairly complicated, use topological tools, and do not seem to yield effective characterization procedures. To the best of our knowledge, solvability of interactive tasks has not been addressed in the deterministic model.

A different line of research in deterministic solvability has dealt with *initial faults.* In the initial faults model, each processor either is initially crashed or remains active forever. Taubenfeld, Katz, and Moran [28] and Taubenfeld and Moran [29] have characterized t-resilience of distributed decision tasks with respect to deterministic protocols in the initial fault model. Interestingly, the characterization is the same as t-resilience with respect to randomized error-free protocols in the regular crash failure model [13]. This raises the question whether the two models have the same capabilities when richer classes of tasks are considered. In this work, we demonstrate a negative answer to this question. We describe a two-stage task that is solvable in the initial fault model but not in the regular crash failure model. The initial fault model enables the election of a "leader" who collects inputs and assigns outputs in a centralized fashion (and is not allowed to fail while doing this). While for the one round, decision tasks, randomization can be used to yield the effect of a leader, this is no longer the case when two-stage interactive tasks are involved. Intuitively, the models are different because in the initial faults model the adversary's powers are seriously limited.

1.2. Finite interactive tasks—motivation. Finite interactive tasks are an extension of distributed decision tasks [22, 13] (where each processor gets a single input and produces a single output). The number of rounds, namely the number of inputs received by each processor, is specified as part of the task. While one round corresponds to decision tasks, even tasks with two rounds constitute a nontrivial generalization. What distinguishes interactive tasks is the on-line character of the interaction. Each new local input is given only after the corresponding processor has irrevocably produced its previous output. Fast processors can work on late inputs while slower ones are still working on earlier inputs. For example, P_1 can work on its third input, P_2 on its fifth input, and P_3 on its first input concurrently. This implies that interactive tasks can neither be described as a single huge decision task nor be decomposed into distinct, independent decision tasks. Sequential systems as defined by Herlihy [17] and by Plotkin [23], such as stacks and queues, can also be formulated as interactive distributed tasks. (Finite interactive tasks will model such systems that are restricted to a bounded number of accesses per processor.)

We give several examples that should help clarify the expressive power of finite interactive tasks. Consider a multiround task where in each round the processors have to select a unique leader among all candidates. We indicate a processor's candidacy, in a given round, by inputting 1, while 0 indicates that the processor is not a candidate. The selected processor outputs 1 in the appropriate round, while all others output 0. (If there are no candidates in a given round, then nobody is selected.) So far, this leader selection problem can be viewed and solved as a collection of independent decision tasks. However, it may be useful to prevent a fast processor from taking permanent control of the system. Thus we add the requirement that no processor is selected twice. This gives rise to different finite interactive tasks, characterized by the total number of "selection campaigns" (or, equivalently, rounds). In the kth task in this list each processor has k inputs and k outputs. In this revised task, a processor that was selected in some round will output 0 in all subsequent rounds, *even if there are no other candidates.* It is not hard to see (and follows easily from our characterization) that the k round task is $(n-1)$-resilient for any $k \geq 1$. (The task is well defined for any k, although after everyone is selected, it becomes quite boring.)

The second example is a variant of the first. Here, in every round every processor receives as input a "priority," which is a positive integer in case the processor is a candidate and 0 if it is not. A leader can be selected if there are at least ℓ other processors with smaller priorities, among all processors that were not chosen already. (If nobody is interested, then every relevant processor gets a 0 and all inputs are among the least ℓ priorities, and in this case everyone outputs a 0 to indicate nonleadership.) As before, the unique selection is indicated by a 1 in the appropriate output. The selection process is repeated k times. How resilient is this task? For $k = 1$, a prospective leader must see at least ℓ other inputs before "fighting" for candidacy. This argument can be formalized to see that for $k = 1$, the task is $(n-\ell-1)$-resilient. Now consider $k = 2$. At the second round, the prospective candidate must still wait for ℓ other processors, but now these could come from a pool of only $n-1$ processors. This argument can be formalized to see that the task now is $(n-\ell-2)$-resilient. For general k and ℓ, it can be shown that this finite interactive task is $(n-\ell-k)$-resilient. This example can be extended further to describe cases where the selection of the ith round leader may depend on inputs from the $(i+1)$st round (e.g., if future inputs of previous leaders can act as tie breakers for the choice of the current leader), and so on.

As a final example, we now describe a different task, which can be viewed as a resource allocation problem. There are m resources $(m < n)$, denoted by 1 through m. At each of k rounds, any number of no more than m of the n processors can announce their interest in getting one of the resources (it does not matter which specific resource) by receiving an input of 1 (otherwise the processor gets 0). If the processor is allocated resource number j, then its output for that round is j. If no resource is allocated, the processor's output is 0. Once a resource is allocated, the processor can retain it by continuing to get input 1. To release a resource, the processor must get input 0. The task specifies that no resource is allocated at the same round to more than one processor. In addition, all requests should be granted. This task is related to a continued renaming problem [5]. For $k = 1$, it is not hard to see that the task is $(n-1)$-resilient (notice the difference from the deterministic case where the name range has to be larger than m). But this is no longer the case if $k > 1$. Consider a fast processor who was not interested in a resource at round 1 but became interested in one at round 2. This processor cannot simply grab a resource, despite the fact that it is guaranteed that such resource will eventually be freed for round 2: up to m tardy processors can wake up and ask for resources at the first round, and only one of them may release its resource in the second round. Our fast processor must wait until at least $m-1$ of the first round users announce their input, and only then can the processor grab a resource. This will be either one of the resources released by one of the $m-1$ processors or the remaining resource, in case all these processors retained their resource. This implies that if m resources were claimed at round 1, the processor may have to wait for the second round inputs of $m-1$ of them before giving an output in the second round. This argument leads to realizing that for $k \geq 2$, this resource allocation problem is 1-resilient but not 2-resilient.

The examples we give demonstrate the versatility of the finite interactive task formulation. We conclude that this is a rich class of tasks which constitutes a meaningful extension to the class of distributed decision tasks.

1.3. Highlights of the characterization. Denote by ISM_t (resp., IMP_t) the class of finite distributed interactive tasks that are solvable in the asynchronous shared memory (resp., message passing) model by a terminating t-resilient randomized protocol, which never errs and works in the presence of a strong adversary scheduler [2, 6]. (A protocol is terminating if each processor stops participating and halts after producing its last output.) Our main results are necessary and sufficient combinatorial conditions which determine membership in ISM_t (for $0 \leq t < n$) and IMP_t (for $0 \leq t \leq \lfloor \frac{n-1}{2} \rfloor$). A similar characterization, for nonterminating protocols, is also given. These results subsume the previous results of [13], which characterized resiliency of distributed decision tasks. We remark that the proof methods in the present work are substantially more involved than those in [13], due to the more general nature of interactive tasks.

We show that what determines resiliency in the randomized error-free model is the amount of information available (at every possible step) to the active processors and whether this information suffices to make moves that are compatible with every potential future development. In order to capture these properties, we associate every finite interactive task T with a directed acyclic graph (DAG), whose nodes represent states of the distributed system. It turns out that the DAG is a convenient tool with which to express the properties we need. In this subsection we outline how the DAG is defined and used for the characterization (while omitting some of the noncrucial details).

The nodes in the DAG contain partial vectors of (multi) input and (multi) output values that are globally known in the system. (For a concrete example, see section 4.) The nodes are first examined according to their "legality" and "consistency." Legality depends only on the indices in the partial vectors (these are the "S vectors" of section 3) and not on the values themselves. It means that no processor has produced an output in a given round before producing all the outputs of earlier rounds and getting all the inputs of earlier and the current rounds. Consistency is related to the task T and does depend on the values in the partial vector (these are the "Q vectors" of section 3). It means that for every possible completion of input values there is a corresponding completion of output values, such that the complete multi-input multi-output vector belongs to the task T.

We put a directed edge from node v_1 to node v_2 in the graph if v_2 extends the values in v_1 and contains exactly one additional input value. (It may contain one additional output, several additional outputs, or none.) We partition the nodes in the DAG into equivalence classes. Two nodes are called *equivalent* if they have the same sets of indices of revealed *inputs* (output indices do not matter here). This definition is useful in situations when there are directed edges from v to both u_1 and u_2 and the latter two nodes are *nonequivalent*. This implies that u_1 and u_2 extend v in different input indices, implying that two different processors have read an additional input in the corresponding moves.

We further refine each equivalence class and say that two equivalent nodes are *input equal* if the *values* of their revealed inputs are the same (output indices and values may still differ). With these definitions, we can describe the notion of a t-*founded* node (relative to the task T). This notion is central to our characterization.

Essentially, we think of a t-founded node as a "good" node from which there exists a strategy to advance while facing up to t crashes without getting stuck or making errors (relative to the task T).

The definition of t-founded is recursive, and we will now briefly describe its main points while omitting some special cases that correspond to the "boundaries" (where at least $n-t$ processors have already terminated). A node v in the DAG is t-founded if it is either a complete multi-input multioutput vector that belongs to the task T or a nonboundary partial vector which satisfies the following condition: there are at least $t+1$ nonequivalent nodes in the DAG, $u_1, \dots, u_t, u_{t+1}$, such that there is a directed edge from v to each u_j, and all u_j's are t-founded. In addition, for every u with a directed edge from v to u there exists an input equal node u' that also has a directed edge from v to it, and u' is t-founded. (The definition of t-founded for boundary nodes is slightly modified, especially in the requirement for the number of nonequivalent sons.)

Having $t+1$ nonequivalent sons of v essentially implies that even if an adversary scheduler crashes t processors, the remaining ones can still make progress. The second condition (regarding input equal sons) guarantees that the system can advance along a path of consistent nodes (with respect to the task T) no matter which processor is active next. The main theorem states that a finite interactive task T has a t-resilient protocol iff the root of the DAG (the partial vector with no inputs and no outputs) is t-founded.[1]

We show that the condition is necessary by the following argument. Suppose T is a task where the root of the DAG is not t-founded and $\mathcal{A}$ is an algorithm for the task that is claimed to withstand t crashes. We demonstrate a strategy for an adversary scheduler that enables it to force the system to advance (with positive probability) along a path of nodes that are not t-founded. We show that such a path leads to an inconsistent node. This means that the adversary can force the algorithm $\mathcal{A}$ to err (with nonzero probability). Notice that in order to be able to force the processors down this path, the adversary needs to be able to fail a previously active processor as well as cause the processors to think that a slow processor may be faulty.

To show that the condition is sufficient, we design a generic protocol which guarantees that all processors take a coordinated walk along a path of t-founded nodes in the DAG. When all processors terminate, this path must lead to a complete vector that is in T. Such coordination is achieved by applying a consensus subroutine to every step in the walk.

We use randomized consensus algorithms as given by [2, 1, 3, 9, 26] for the shared memory model and by [6] for the message passing model. It is interesting to observe that randomization is needed only for consensus and gives no extra power beyond that. This follows from our generic algorithm, which can be viewed as a repeated *deterministic reduction* of an arbitrary interactive task to consensus (which itself requires randomized solutions). Therefore, our characterizations also can be applied to deterministic systems that have a built-in consensus mechanism, and in this case no randomization at all will be needed (see section 7 for additional discussion).

1.4. Organization. The remainder of this paper is organized as follows. Section 2 describes the computation models. In section 3 we formally define interactive tasks and related notions used in this paper. In section 4 we describe the DAG associated

[1]As stated, the theorem holds for the shared memory model. A slight variation, which corresponds mainly to the final stages of an execution, makes it applicable to the message passing model (the variation is termed *t-valid*).

with an interactive task. This graph plays a central role in section 5, where the characterization theorem for the shared memory model is stated and proven. Section 6 contains the characterization for the message passing model. Finally, in section 7 we present some concluding remarks, including the equivalence of error-free randomized model and the deterministic model with access to consensus.

2. The models. In this section we define the models of asynchronous computation which we use in the sequel, as well as the class of appropriate schedulers (or adversaries) that control our systems.

An asynchronous concurrent system is a collection of n processors. Each processor, P, is a (not necessarily finite) state automaton with an internal input register in_P and an internal output register out_P. The set of all states of the processor P is denoted by S_P. The input register contains a value v taken from a set IN, while the output register has initially the value $\perp$. The value in the output register can be changed to any value in the set OUT ($\perp \notin OUT$). For every input value that the processor gets, it must change the value of the output register (possibly to the same value as before) exactly once before receiving the next input or terminating (after the last input).

The two models we consider differ in the way that processors communicate among themselves. In the *shared memory model*, processors communicate via *shared registers*. Every shared register r is associated with a set of processors R_r, $\mid R_r \mid > 0$, that can read from the register and a set of processors W_r, $\mid W_r \mid > 0$, that can write into the register. These registers are atomic with respect to the read and write operations. (Although our protocols use only the simpler, multireader single-writer registers, the necessity of our conditions holds in the more general setting, presented here.)

In the second model, the *message passing model*, there is a communication link between every two processors. Processors may send and receive messages via these links. The links are atomic with respect to the send-message and receive-message operations, but there is no guarantee on the order in which the messages are received once they are sent.

Processors execute their programs by taking steps. An atomic step consists of one of the following:

1. An internal operation, possibly involving coin tosses.
2. Getting information from other processors (reading from a shared register in the shared memory model, or receiving a message from a link in the message passing model).
3. Giving information to other processors (writing into a shared register in the shared-memory model, or sending via a link in the message passing model).
4. Getting a new local input.
5. Giving a new local output.

Formally, every processor P takes steps according to its *transition function*, T_P. In case the step taken was a read (or receive), the new state of P depends not only on its old state but also on the value read (received) by this action. The transition function T_P could be either deterministic or nondeterministic. In the latter case, the actual step taken is decided via coin tosses (in a nondeterministic step the only difference between the old and new states is the coin toss, and no communication occurs in such a step). Given an asynchronous system as specified above, a *protocol* is a collection of n transition functions $T_1, \dots, T_n$, one per processor.

A *configuration* C of the system, in the shared memory (message passing) model, consists of the state of each processor together with the contents of the shared registers (communication links). In an *initial configuration*, every processor is in an initial state, and all shared registers and output registers contain the default value $\perp$ (all the links are empty). The set of all configurations will be denoted by $\mathcal{C}$. A *step* takes one configuration to another by activating a single processor P. A *run* of length ℓ is a sequence of ℓ steps. Each run has an associated *schedule* which is a sequence of ℓ processors, numbered according to the order of processors that take steps in that run. We denote schedules, finite or infinite, by a list of processor numbers, e.g., $(2, 3, 3, 2, 1)$. We say that processor P is activated k times in a run if P appears k times in the run's schedule. The *history* $\mathcal{H}$ of a run is the sequence obtained by interleaving the sequence of configurations with the steps in the run, starting with the initial configuration. For a finite run, we refer to the last configuration in its history as the *current* configuration.

A *scheduler* $\mathcal{S}$ is a mapping from $\mathcal{H}$ into the set of n processors. Given the configuration of the system, the scheduler picks the next processor that is to take a step. The scheduler could be either a deterministic mapping or a randomized one. The scheduler can be viewed as an adversary which tries to prevent the system from reaching its goal. Under this definition, the adversary is very strong: it has complete knowledge of the state of every processor and of the contents of the shared registers (communication links) during the entire history. In case the processors are randomized, the scheduler could also base its choices on the outcome of past coin flips as well as the current coin flip (if the next step is randomized). We do not allow it, though, to be able to predict *future* randomized moves of the processors. This is a necessary requirement if randomization is to be helpful at all, and it is used in all algorithms where randomization is employed, e.g., [25, 20, 7]. In addition, the scheduler picks the inputs to be given to the processors (from the possible legal inputs).

Finally, we note that both models are asynchronous, meaning that there is no global clock in the system and each processor runs at its own pace. In the message passing model this also means that messages can be delivered with arbitrary delays (and possibly out of order).

In each one of these models there are two submodels according to the nature of the processors. All the processors can be of one of two types:

1. Terminating processors that terminate and halt once their final output has been given.
2. Nonterminating processors that continue to operate even after giving their final output. These processors may help and coordinate among the slower processors that have not yet finished their task.

The second type of processor yields a more robust system, which could solve any task solvable by processors of the first type.

In this work we study the resiliency of tasks in these two models.[2] There is a resiliency parameter, t, that denotes the extent of resiliency required. The *t-resiliency* requirement imposes restrictions both on the scheduler and on the protocol. In every infinite schedule under the scheduler, S, at least $n - t$ processors will be activated infinitely many times. We will call such a scheduler a *t-bounded scheduler*. Intuitively, this means that the scheduler may fail-stop at most t processors.

A *round* of a t-bounded schedule is a minimal schedule of the processors (starting from any configuration) in which at least $n - t$ processors are scheduled at least once.

[2]Most of our results are developed for terminating processors. Once these results are obtained, the case of nonterminating processors is easy to handle. The results for the nonterminating model are stated in section 7.

Each processor which is scheduled in the round is said to be *active* in it. Notice that after $n - t$ processors have terminated, a schedule in which all the terminated processors are scheduled (and thus no move will be made) is also a round.

DEFINITION 1. *We say that an algorithm, A, is a t-resilient algorithm for a task, T, if the following conditions hold:*

- *Safety. For every legal input for T, for any t-bounded scheduler, S, and for any execution of A, if all n processors terminate (or in the nonterminating models, if all n processors have given their last output), then the resulting output is correct with respect to the input for T (see Definition 2).*
- *Liveness. There is a constant, M, such that for every processor the* expected *number of rounds in which it is active, before producing its final output, is bounded by M. For terminating processors this requirement implies that after $n-t$ processors have terminated, the expected number of* steps *taken by each of the remaining t processors in order to terminate is bounded by M (no matter how the steps of this individual processor are interleaved with steps of other processors).*

It should be noticed that we require that protocols never err. Although there could be a positive probability for very long nonterminating runs, this probability should be very small (converging to 0 with the length of the run).

3. Basic definitions.

In this section we define finite interactive tasks and what it means for such a task to be t-resilient. We introduce the notation of partial vectors and use this notation to express states of systems that implement interactive tasks and transitions between such states.

Distributed interactive tasks are a generalization of distributed decision tasks. A decision task is a collection of input-output pairs (each pair is an n-vector). Analogously, we define an *interactive$_k$ task* as a collection of k pairs. Each element of every pair is an n-vector. The first element represents an input vector and the second an output vector. The following formulation turns out to be convenient for our purposes.

DEFINITION 2. *An interactive$_k$* task *(or k-*stage distributed interactive task*), T, is a collection of $2k$-tuples of the form $\langle I^1, \ldots, I^k, O^1, \ldots, O^k\rangle$. Each I^j and O^j is an n-component vector over an arbitrary alphabet, corresponding to the n inputs and n outputs of the jth stage, respectively. The* legal inputs *for T (denoted $IN(T)$) are all the k-tuples of n-component vectors $\langle I^1, \ldots, I^k\rangle$ such that there exists $\langle O^1, \ldots, O^k\rangle$ for which $\langle I^1, \ldots, I^k, O^1, \ldots, O^k\rangle \in T$.*

When an execution of an interactive$_k$ task starts, a processor P_i gets the input for the first stage. After it submits the output for this stage (which is irrevocable) it receives the input for the second stage, and so on. In general, each processor, P_j, gets its input for stage $i + 1$ only after submitting its output for stage i, for all $1 \le i \le k - 1$. Different processors may be in different stages simultaneously. At a certain point during an execution, it is possible that there will already be, for example, seven outputs for the first stage, five for the second, two for the third, and none for the fourth.

We will represent the states in an instance of an algorithm using partial vectors which contain input and output values. If we take a vector $\vec{V}$ of length n over a certain alpha-beth, Σ, and a set of indices, J, where $J \subseteq \{1, \ldots, n\}$, then the partial vector $\vec{V_J}$ is defined as follows:

- If $i \in J$, then $V_J(i) = V(i)$, the ith coordinate of $\vec{V}$.
- Otherwise $V_J(i) = \perp$, where $\perp$ is an agreed sign for "don't know yet."

This means that we know only the values of $\vec{V}$ at the indices indicated by J. Typically,

$\vec{V}_J$ is compatible with more than one vector $\vec{V}$, namely, there may be a $\vec{V}'$ such that $\vec{V}' \neq \vec{V}$ but still $\vec{V}'_J = \vec{V}_J$. For example, the partial vector $\vec{V}_\phi$ is compatible with every vector of the same length.

One property of a $2k$-tuple, representing the state of an interactive$_k$ task, corresponds to the temporal order in which input and output indices are revealed. This property is formalized in the next definition, which uses the notion of *corresponding partial input and output.* We say that a partial input and output $Q = \langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ corresponds to the $2k$-tuple $S = \langle A_1\ ,\ \ldots\ ,\ A_k\ ,\ B_1\ ,\ \ldots\ ,\ B_k\rangle$ if the revealed indices in Q match the index-sets in S.

DEFINITION 3. *Let $S = \langle A_1, \ldots, A_k, B_1, \ldots, B_k\rangle$ be a $2k$-tuple of index-sets. S will be called a* legal 2k-tuple *if the following conditions are satisfied:*

- *For all i, $1 \le i \le k$, it holds that $B_i \subseteq A_i$ and $A_i, B_i \subseteq \{1, ..., n\}$.*
- *For all i, $1 \le i \le k-1$, it holds that $A_{i+1} \subseteq B_i$.*

Let Q be a partial input and output vector, corresponding to S. We say that Q is a legal partial input–output vector *if S is a legal $2k$-tuple.*

The requirements ensure that S represents a possible state of revealed inputs and outputs in an instance of an algorithm for an interactive$_k$ task. By demanding that $B_i \subseteq A_i$ we guarantee that no processor will produce its ith output before receiving its ith input. By demanding that $A_{i+1} \subseteq B_i$ we guarantee that no processor will get its $(i+1)$st input before producing its ith output. However, the legality requirement does not suffice. We also want to guarantee that the "input output contents" matches a state that can fit the requirements of the task itself. This property is stated in the next definition.

DEFINITION 4. *Let $Q = \langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ be a legal $2k$-tuple of partial vectors. Q will be called* T-consistent *if for every extension I^1 to I^k, of $I^1_{A_1}$ to $I^k_{A_k}$ such that $\langle I^1, \ldots, I^k\rangle \in IN(T)$, there exist extensions O^1 to O^k, of $O^1_{B_1}$ to $O^k_{B_k}$ such that $\langle I^1\ ,\ \ldots\ ,\ I^k\ ,\ O^1\ ,\ \ldots\ ,\ O^k\rangle \in T$.*

We will say that output o^i_j is a consistent output *for the $2k$-tuple $\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}$, $O^1_{B_1}\ ,\ \ldots\ ,\ O^k_{B_k}\rangle$ if the $2k$-tuple $\langle I^1_{A_1}\ ,\ \ldots\ ,\ I^k_{A_k}\ ,\ O^1_{B'_1}\ ,\ \ldots\ ,\ O^k_{B'_k}\rangle$, is T-consistent, $B'_k = B_k$ for every $k \neq i$, $B'_i = B_i \cup \{j\}$, and the jth coordinate of $O_{B'_i}$ equals o^i_j.*

An inconsistent $2k$-tuple corresponds to a state where the scheduler, by giving legal inputs, can force the algorithm to err. Thus, intuitively, consistency is a property that the processors wish to maintain.

Let us now formalize the notion of a "one-step" advancement from a legal $2k$-tuple of partial input-output vectors to another legal one. Intuitively, a one-step advancement corresponds to one additional input read by one processor and possibly a resulting extension of one or more outputs. If the new input is from the ith stage $(i \ge 2)$, then, by our convention, the reading processor has already given its $(i-1)$st output. There is an exception to this intuition, which occurs toward the end of a run, when only t or fewer processors still have unrevealed inputs. In this case, a single additional output is also considered an advancement, as the remaining t or fewer processors might be faulty and thus we cannot insist on input extensions here. We first formally define these early and late parts in a run.

DEFINITION 5. *The earlier parts of a run of an algorithm, where the set of processors that have already read their last input, A_k, satisfies $|A_k| < n-t$, are called the* main phase. *The final parts of a run of an algorithm, where $|A_k| \ge n-t$, are called the* concluding phase.

The definition of a one-step advancement is as follows.

DEFINITION 6. *Let*

$$S = \langle A_1, \ldots, A_k, B_1, \ldots, B_k \rangle, S' = \langle A'_1, \ldots, A'_k, B'_1, \ldots, B'_k \rangle$$

be two legal $2k$-tuples and let

$$Q = \langle I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} \rangle,$$

$$Q' = \langle I^1_{A'_1}, \ldots, I^k_{A'_k}, O^1_{B'_1}, \ldots, O^k_{B'_k} \rangle$$

*be partial inputs and outputs corresponding to S and S', respectively. The pair $[S', Q']$ will be called a t-*subsequent *of the pair $[S, Q]$ if the following three conditions hold:*

- *For all i, $1 \le i \le k$ it holds that $B_i \subseteq B'_i$.*
- *Exactly one of the following is true:*
  1. *There is exactly one i ($1 \le i \le k$) and one j ($j \in \{1, \ldots, n\}$) such that $j \notin A_i$, $j \in B_{i-1}$, $A'_i = A_i \cup \{j\}$, and for all $m \ne i$ it holds that $A'_m = A_m$.*
  2. *The system is in the concluding phase ($|A_k| \ge n - t$) and there is exactly one i ($1 \le i \le k$) and one j ($j \in \{1, \ldots, n\}$) such that $j \notin B_i$, $B'_i = B_i \cup \{j\}$, for all $m \ne i$ it holds that $B'_m = B_m$, and for all $1 \le \ell \le k$ it holds that $A_\ell = A'_\ell$. This corresponds to a single new output, given by P_j in the ith stage.*
- *Q' is an extension of Q and there exists $< I^1, \ldots, I^k > \in IN(T)$ such that $< I^1, \ldots, I^k >$ is an extension of $< I^1_{A'_1}, \ldots, I^k_{A'_k} >$.*

When $[S', Q']$ is a t-subsequent of $[S, Q]$ and j is the unique index extending A_i (in case 1) or B_i (in case 2), we say that P_j is the processor associated with the t-advancement.

The number of possible failures, t, is a parameter in the definition. An additional input is considered an advancement in both the main and the concluding phases, while an additional output (on its own) is considered an advancement only in the concluding phase. Notice that this definition can be used independently of consistency.

In [17], Herlihy studied concurrent implementations of sequential data objects. Finite versions of such objects may be represented as interactive$_k$ tasks. This can be done by taking the set of all possible sequences of correct operations on the object. For example, consider a concurrent implementation of a stack. The processors in this implementation are servers that receive in their input one of two possible requests:

- *push(value)*: push the parameter *value* onto the top of the stack and return (output) ℓ, where ℓ is the level of the stack in which the value was put (1 when the stack contains one element, 2 when the stack contains two elements, etc.). In our example the stack level is required as output mainly for didactic reasons.
- *pop*: pop the top of the stack and return (output) its value ("empty" if the stack is empty).

Obviously the outcome of a request given to a processor may depend on previous requests to some of the other processors. We will use the example of a three-processor stack as implemented by an interactive$_2$ task (two requests per processor) to illustrate the definitions. The resiliency parameter will be $t = 2$. Assume that in the first stage P_1 will be requested to push 5, P_2 will be requested to pop, and P_3 will be requested to push 7. These will be the inputs (but the processors do not know them initially). From the initial configuration, where no inputs and no outputs are known, we can advance in one step to a state where one input is known (e.g., P_1 has "push 5" as input) or to a state where one input and one output are known (for instance, P_1 with the input "push 5" and with output "1," or P_2 with input "pop" and output "empty").

4. The corresponding DAG. In this section we define the DAG associated with an interactive$_k$ task. A run of an algorithm can be viewed as a sequence of $2k$-tuples of indices with their corresponding $2k$-tuples of partial inputs and outputs. We associate this sequence with a path in a DAG. There is a directed edge from v to u if u is a possible state representing a one-step advancement from v (namely, u is a t-subsequent of v). More formally, we have the following definition.

DEFINITION 7. *Let T be an interactive$_k$ task. The* DAG associated with T, *which we denote by $D(T)$, has the set of nodes*

$$V = \{[S, Q] | S = \langle A_1 \,,\, \dots \,,\, A_k \,,\, B_1 \,,\, \dots \,,\, B_k \rangle, Q = \langle I^1_{A_1}, \,\dots, \, I^k_{A_k}, \, O^1_{B_1}, \,\dots, \, O^k_{B_k} \rangle,$$

S is a legal $2k$-tuple, Q corresponds to S, and there exists $\langle I^1, \dots, I^k \rangle \in IN(T)$ such that $\langle I^1, \dots, I^k \rangle$ is an extension of $\langle I^1_{A_1} \,,\, \dots \,,\, I^k_{A_k} \rangle\}$. The set of directed edges, E, is the set of all (v_1, v_2) where $v_1 = [S_1, Q_1]$ and $v_2 = [S_2, Q_2]$, such that v_2 is a t-subsequent of v_1 and v_2 is T-consistent.

The root of the DAG is the node $[\langle \phi, \dots, \phi, \phi, \dots, \phi \rangle, \langle I^1_\phi, \dots, I^k_\phi, O^1_\phi, \dots, O^k_\phi \rangle]$. The leaves (nodes from which there are no outgoing edges) are either nodes containing a complete $2k$-tuple of indices ($A_1 = \cdots = A_k = B_1 = \cdots B_k = \{1, \dots, n\}$) with the corresponding full inputs and outputs, or nodes whose $2k$-tuple of indices is incomplete but do not have T-consistent sons. In general, we think of a node in the DAG as representing a possible state of the published input and agreed-upon output values at some point during a possible execution. An edge represents a transition according to the t-subsequency relation. Notice that one can "label" an edge from u to v according to the processor P_j which is associated with the t-advancement.

In our example of the stack, there will be an edge from the node representing the initial state, $[\langle \phi \,,\, \phi \,,\, \phi \,,\, \phi \rangle, \langle I^1_\phi \,,\, I^2_\phi \,,\, O^1_\phi \,,\, O^2_\phi \rangle]$, to the node representing the state where P_1 has seen the input "push 5," which is the node $[\langle \{1\}, \phi, \phi, \phi \rangle, \langle \{\text{"push(5)"}, \perp, \perp\}, I^2_\phi, O^1_\phi, O^2_\phi \rangle]$. There will also be an edge from the initial node to the node representing the state where P_2 has seen "pop" and returned "empty." This node is $[\langle \{2\}, \phi, \{2\}, \phi \rangle, \langle \{\perp, \text{"pop"}, \perp\}, I^2_\phi, \{\perp, \text{"empty"}, \perp\}, O^2_\phi \rangle]$.

The DAG represents all the possible advancements for all possible inputs. It is useful to group the vertices into equivalence classes according to the amount of information in them, disregarding the actual input and output values. This leads to the definition of *t-equivalent* nodes in the DAG.

DEFINITION 8. *Let*

$$v = [\langle A_1 \,,\, \dots \,,\, A_k \,,\, B_1 \,,\, \dots \,,\, B_k \rangle, \langle I^1_{A_1} \,,\, \dots \,,\, I^k_{A_k} \,,\, O^1_{B_1} \,,\, \dots \,,\, O^k_{B_k} \rangle]$$

and

$$v' = [\langle A'_1 \,,\, \dots \,,\, A'_k \,,\, B'_1 \,,\, \dots \,,\, B'_k \rangle, \langle I^1_{A'_1} \,,\, \dots \,,\, I^k_{A'_k} \,,\, O^1_{B'_1} \,,\, \dots \,,\, O^k_{B'_k} \rangle]$$

be two nodes in the DAG corresponding to an interactive$_k$ task, T. These two nodes will be called t-equivalent *if the following two conditions are satisfied:*

- *For all i, $1 \le i \le k$, it holds that $A_i = A'_i$.*
- *If $|A_k| \ge n - t$, then for all i, $1 \le i \le k$, it holds that $B_i = B'_i$.*

In the equivalence relation we are concerned only with the *indices* of the revealed inputs/outputs, while the values in these indices do not matter. From the definition it follows that two nodes are *non-t-equivalent* in one of two cases: either they do not have the same indices of input revealed or they are both in the concluding phase, and their revealed output indices are different (for some i it holds that $B_i \neq B'_i$). As with the definition of t-subsequents, we take the outputs into consideration only in the concluding phase.

In our example, when examining whether the task is 2-resilient ($t = 2$), the nodes

$$v_1 = [\langle \{1,2,3\}, \phi, \{1,2,3\}, \phi\rangle, \\ \langle\{\text{“push(17)”, “push(27)”, “push(37)”}\}, I^2_\phi, \{1,3,2\}, O^2_\phi\rangle]$$

and

$$v_2 = [\langle \{1,2,3\}, \phi, \{1,3\}, \phi\rangle, \\ \langle\{\text{“push(57)”, “push(67)”, “push(87)”}\}, I^2_\phi, \{2,\perp,1\}, O^2_\phi\rangle]$$

are 2-equivalent (both nodes have $A_1 = \{1,2,3\}, A_2 = \phi$). While the nodes

$$u_1 = [\langle \{1,2\}, \{2\}, \{1,2\}, \phi\rangle, \\ \langle\{\text{“push(17)”, “push(27)”}, \perp\}, \{\perp, \text{“pop”}, \perp\}, \{2,1,\perp\}, O^2_\phi\rangle]$$

and

$$u_2 = [\langle \{1,2\}, \{2\}, \{1,2\}, \{2\}\rangle, \\ \langle\{\text{“push(17)”, “push(27)”}, \perp\}, \{\perp, \text{“pop”}, \perp\}, \{1,2,\perp\}, \{\perp, 27, \perp\}\rangle]$$

are *non*-2-equivalent. (These nodes have $|A_2| = 1 = 3 - 2$, but their B_2 sets are not the same.)

Notice that two nodes (collections of partial vectors) with the same input indices are revealed, but different values revealed in these indices *are* equivalent. The reason they are defined as equivalent is that in a specific run we are interested only in the input for this run, which cannot be compatible with both. We refine the t-equivalence relation, thereby partitioning the equivalence classes into subclasses, such that all the nodes in the same subclass will have the same *input values* (not just indices). The nodes in this subclass will be called *input-equal.* Input-equal nodes may differ both in their output indices and in their output values (provided they remain t-equivalent).

DEFINITION 9. *Let*

$$v = [\langle A_1, \dots, A_k, B_1, \dots, B_k\rangle, \langle I^1_{A_1}, \dots, I^k_{A_k}, O^1_{B_1}, \dots, O^k_{B_k}\rangle]$$

and

$$v' = [\langle A'_1, \dots, A'_k, B'_1, \dots, B'_k\rangle, \langle I^1_{A'_1}, \dots, I^k_{A'_k}, O^1_{B'_1}, \dots, O^k_{B'_k}\rangle]$$

be two nodes of the DAG corresponding to an interactive$_k$ task, T. Let t be a resiliency parameter. The nodes v and v' will be called input-equal if the following conditions hold:

- *v and v' are t-equivalent.*
- *For all i, $1 \leq i \leq k$, it holds that $I^i_{A_i} = I^i_{A'_i}$ (which implies $A_i = A'_i$ as well).*

In the example of the stack, assume that the inputs in the first stage are "push 21" for P_1, "push 22" for P_2, and "push 23" for P_3. Now suppose that the input in the second stage for all three processors is "pop." In this situation, the nodes

$$v_1 = [\langle \{1,2,3\}, \{1,2,3\}, \{1,2,3\}, \{2\}\rangle, \langle \{\text{“push(21)”}, \text{“push(22)”}, \text{“push(23)”}\}, \{\text{“pop”}, \text{“pop”}, \text{“pop”}\}, \{2,1,3\}, \{\perp, 23, \perp\}\rangle]$$

and

$$v_2 = [\rangle \{1,2,3\}, \{1,2,3\}, \{1,2,3\}, \{2\}\rangle, \langle \{\text{“push(21)”}, \text{“push(22)”}, \text{“push(23)”}\}, \{\text{“pop”}, \text{“pop”}, \text{“pop”}\}, \{3,2,1\}, \{\perp, 21, \perp\}\rangle]$$

(which correspond to two different schedulings of the same input values) are input-equal.

5. Characterization for the shared memory model. In this section we state and prove the characterization theorem for t-resilient interactive$_k$ tasks in the shared memory model. The following claim will be useful in the proof of the main theorem for this model.

DEFINITION 10. *We say that an algorithm A is an* immediate-input *algorithm if the first step of every processor after writing an output in its private output register (as long as this is not the last (kth) output) is reading the next input from its private input register.*

CLAIM 1. If a distributed interactive$_k$ task, T, is solvable by a t-resilient algorithm, A, in the shared memory model, then T is also solvable by a t-resilient immediate-input algorithm, A'.

Proof. We show how to construct such an immediate-input algorithm, A', on the basis of the given algorithm, A. The algorithm A' will have all the shared registers of A plus an array of k Boolean multireader single-writer registers per processor (one bit for every input in T). The bits in all n arrays are initialized to "false." The state of a processor will consist of a pair (s_A, s_{add}), where s_A is a state of the processor in A and s_{add} reflects the additional parts, performed by A'. The algorithm A' will simulate algorithm A (changing s_A according to algorithm A) except at the following points:

- After a processor, P, writes the ℓth output ($\ell < k$), P's next step is to read the next, $(\ell+1)$st, input. However, P will leave the indicator corresponding to the $(\ell+1)$st input with the value "false," meaning that it did not want to read its input yet (in A). This will change only s_{add}, not s_A.
- When P is supposed to read its input (according to algorithm A) it will, instead, change the indicator corresponding to this input to "true." It will also change s_A according to algorithm A (and the value of this input).

It is not hard to verify that if A is t-resilient, then so is A'. ∎

We now define the notion of a t-founded node relative to a task T. Intuitively, a t-founded node is a "good" node from which a t-resilient algorithm can progress to correct outputs, regardless of the scheduling and of future revealed inputs. The definition of t-foundedness will be recursive.

DEFINITION 11. *Let*

$$v = [\langle A_1, \ldots, A_k, B_1, \ldots, B_k\rangle, \langle I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k}\rangle]$$

be a node in the DAG. The node v will be called t-founded *relative to the interactive$_k$ task T if*

- *the node v is a "completed leaf" $v = [\langle\{1,\ldots,n\}_1, \ldots, \{1,\ldots,n\}_{2k}\rangle, \langle I^1,\ldots,I^k,O^1,\ldots,O^k\rangle]$ and $\langle I^1,\ldots,I^k,O^1,\ldots,O^k\rangle \in T$,*

or

- *the node $v = [\langle A_1, \ldots, A_k, B_1, \ldots, B_k\rangle, \langle I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k}\rangle]$ is not a completed leaf (namely $B_k \neq \{1,\ldots,n\}$), and the following conditions hold:*
  1. *The node v has at least α non-t-equivalent sons, where $\alpha = min\,(t+1\,,\, n-|B_k|)$.*
  2. *For every possible t-subsequent, u, of v (u is not necessarily T-consistent), there exists an input-equal node u' that is a t-founded son of v.*

As we mentioned earlier, t-equivalent nodes have the same input indices revealed. Therefore, if two nodes are t-equivalent sons of some node, then the same processor made the t-advancement to both sons. Since the scheduler may fail up to t processors, t-resiliency necessitates, in the main phase, at least $t+1$ processors that can advance from any given situation. However, during the concluding phase there may be fewer than t active processors. (These are the processors that have not yet submitted their final output.) The number of processors able to make a t-advancement surely cannot exceed the number of active processors. This argument motivates the definition of α. If we require that there be at least α processors that are able to advance (i.e., at least α non-t-equivalent sons), some processor will always be able to advance from this state, no matter what the scheduling sequence is.

In addition, we need to have a legal consistent step available no matter what the input *values* are. This is embodied in the requirement for a t-founded input-equal son for every possible t-subsequent. Notice that the specific input values in a run cannot be chosen by the processors, while the output values *are* chosen by the processors. We remark that by the definition of t-foundedness, every path in the DAG which starts at the root, ends at some leaf, and proceeds only through t-founded nodes must end in a leaf belonging to T.

We are now in position to state and prove the characterization theorem for the shared memory model.

THEOREM 1. *In the terminating shared memory model, for $0 \leq t \leq n-1$, an interactive$_k$ task T is t-resilient iff the root of its DAG is t-founded relative to T.*

Proof. First we prove the $\Rightarrow$ direction: if the root of the DAG, $R_0 = [\langle\phi,\ldots,\phi, \phi,\ldots,\phi\rangle, \langle I^1_\phi, \ldots, I^k_\phi, O^1_\phi, \ldots, O^k_\phi\rangle]$ is *non*-t-founded relative to T, then T is not t-resilient. A high-level description of the proof is as follows: Assume that the root of the DAG is non-t-founded and yet there exists a t-resilient algorithm for the task T. We will show how the scheduler can force the system to make transitions along a sequence, s, of non-t-founded nodes in the DAG. Each node in s reflects the input values that were read and the output values that were produced by the processors. In every transition, at least one new input is read or one new output is produced. Since the number of inputs and outputs is finite, $2nk$, the number of such transitions will also be finite. This process will terminate either at a leaf of the DAG that is non-t-founded or in a state that is not represented by a node in the DAG. In the latter case, the inputs and outputs represent an inconsistent state. In either case, the scheduler has forced the algorithm to err.

A processor that had either terminated or read an input for which it has not yet given an output will be called a *nonreading* processor. Let v be a node in the sequence s. The scheduler's strategy will preserve two invariants regarding v:

1. The node v is non-t-founded, and every node in the DAG, u, that is input-equal to v is either inconsistent or is both consistent and non-t-founded.
   Notice that this invariant holds at the root, as we assume that the root is non-t-founded and there are no other nodes that are input-equal to the root.
2. The node, v, belongs to one of the following two categories:
   (a) The state represented by v is in the main phase and the number of non-reading processors in this state is smaller than $n - t$. (Notice that this invariant holds at the root, as there are no nonreading processors at the root.)
   (b) The state represented by v is in the concluding phase and at least $n - t$ processors have terminated (given their final output).

The scheduler will allow the system to advance by making a t-advancement while keeping these invariants. We will denote them as advancements from node R_ℓ to node $R_{\ell+1}$ ($\ell = 0$ at the root). In such an advancement there are two possibilities.

1. In the main phase, a t-advancement corresponds to an additional input and possibly several additional outputs.
2. In the concluding phase an advancement can be either an additional single input (only) or a single additional output.

We now present the detailed proof. Suppose, by way of contradiction, that T *is* t-resilient, namely, there exists a t-resilient algorithm, A, which implements T. By Claim 1 we can assume that this algorithm is an immediate-input algorithm. The root of the DAG corresponding to T, R_0, is *non*-t-founded. By Definition 11 there are two possible reasons for a node R_ℓ to be non-t-founded:

(A) R_ℓ has less than α non-t-equivalent sons.
(B) R_ℓ has a t-subsequent, $R'_{\ell+1}$, such that for every $R_{\ell+1}$ which is input-equal to $R'_{\ell+1}$ and is a son of R_ℓ in the DAG, $R_{\ell+1}$ is *non*-t-founded.

We describe an adversary scheduler which, as long as the system is at a node that is non-t-founded due to possibility (B), will force the algorithm to advance by choosing one of the non-t-founded input-equal sons. Notice that when the system moves from R_ℓ to $R_{\ell+1}$, the first invariant is maintained. ($R_{\ell+1}$ is non-t-founded and all of it's input-equals are either inconsistent (not a node in the DAG) or consistent (in the DAG) but non-t-founded.)

Let P_i be the processor associated with the t-advancement from R_ℓ to $R'_{\ell+1}$ (Definition 6). If the system is in the main phase, this advancement corresponds to P_i reading an input. If the system is in the concluding phase, this advancement corresponds to either P_i reading an input or P_i writing an output (case 2 of Definition 6).

In case P_i's step is reading an input, in the concluding phase, the adversary scheduler will activate P_i once and P_i will read this input, since A is an immediate-input algorithm. The resulting configuration will be $R_{\ell+1}$. In case P_i's step is reading an input in the main phase, the scheduler activates P_i (which reads its input) as above, but then the scheduler may perform additional activations according to the following cases.

If $n-t$ or more inputs of the final stage are known, then the scheduler will activate the $n - t$ processors that have read their final input (say, in round-robin order) until *all* of them give an output and terminate. By Definition 1, within a finite expected number of steps all $n-t$ processors will give outputs and terminate. The configuration reached after all $n - t$ processors have given their final output is $R_{\ell+1}$. Notice that this case preserves invariant 2(b), as we reach the concluding phase not only with

$n-t$ inputs of the final stage but also with $n-t$ outputs and terminated processors.

If, however, there are fewer than $n-t$ inputs of the final stage, then we know that the system remains in the main phase. In this case, let S_p denote the set of nonreading processors (after the read step of P_i). Notice that by invariant 2(a) it holds that $|S_p| \leq n-t$ because before P_i read its input, there were fewer than $n-t$ nonreading processors. The scheduler will do one of the following:

1. If $|S_p| < n-t$, then the resulting configuration is $R_{\ell+1}$ (no additional activations). In this case, the second invariant holds ($R_{\ell+1}$ is in the main phase and $|S_p| < n-t$).
2. If $|S_p| = n-t$, then the scheduler will activate the processors in S_p (say, in round-robin order) until exactly one of them gives an output. By Definition 1, such an output will be given within a finite expected number of steps. The resulting configuration will be $R_{\ell+1}$. This configuration adheres to the second invariant as now there are only $n-t-1$ nonreading processors.

When examining $R_{\ell+1}$ we see there are now two possibilities. The first is that the outputs given were inconsistent, in which case the algorithm has erred. The second possibility is that $R_{\ell+1}$ is consistent. The configurations $R_{\ell+1}$ and $R'_{\ell+1}$ are input equal. $R_{\ell+1}$ is a son of R_ℓ in the DAG. Since we assumed R_ℓ is non-t-founded due to (B), this implies that $R_{\ell+1}$ is non-t-founded.

The other t-advancement possible is that P_i's step in advancing from R_ℓ to $R'_{\ell+1}$ is writing an output. In this case the system must be in the concluding phase, and due to the second invariant we know that $n-t$ processors have already terminated. By Definition 1, since A is assumed to be a t-resilient algorithm, P_i will write an output, with probability 1, after being scheduled a finite number of times. Thus, the scheduler will activate P_i until it writes an output. In this case, the resulting configuration is the desired $R_{\ell+1}$. Again, if $R_{\ell+1}$ is inconsistent the algorithm has erred and otherwise, $R_{\ell+1}$ is non-t-founded (since all the t-advancements made by P_i in this case are input equal and thus non-t-founded). As the system was in the concluding phase and the second invariant was true before the advancement, it will also hold after the advancement.

Hence, in all cases, within a finite expected number of steps the scheduler can force the system to reach $R_{\ell+1}$ without failing any processor. The system is now at a non-t-founded node, $R_{\ell+1}$. The argument used for R_ℓ applies to $R_{\ell+1}$ as well, and can therefore be repeated.

Since the depth of the DAG is finite, after a finite number of such t-advancements we will reach a node R_f that is non-t-founded due either to possibility (A) or to the fact that it is a complete leaf which is not in T ($\langle I^1, \ldots, I^k, O^1, \ldots, O^k \rangle \notin T$).

In the main phase, a node v cannot be non-t-founded due to possibility (A): from the second invariant, less than $n-t$ processors are nonreading. Therefore, at least $t+1$ processors can read an additional input and thus v must have at least α non-t-equivalent sons. (Remember that $\alpha \leq t+1$ and that due to Definition 4 an additional input preserves consistency.)

If R_f is a completed leaf not in T, then the algorithm has erred, contradicting the assumption that algorithm A implements T without erring. Otherwise, R_f is non-t-founded due to possibility (A). This means that R_f has less than α non-t-equivalent sons. As $\alpha \leq t+1$, we know that at most t processors can be associated with a (consistent) t-advancement from R_f (Definitions 11 and 7). Denote the set of these processors by C_f.

For every output that is consistent for R_f (Definition 4) there exists a consistent son of R_f which includes this output. Therefore, in the concluding phase when it holds that $n - t$ processors have already terminated (due to the second invariant), the fact that there are fewer than α processors able to make a t-advancement implies that the number of these processors is smaller than the number of active processors ($\alpha = min\,(t+1\ ,\ n - |B_k|)$, Definition 11). It follows that there exists at least one additional active processor (not in C_f) which cannot give a consistent output. Our adversary scheduler will activate all the processors not in C_f. Since the algorithm is t-resilient, one of the remaining (activated) processors must give an output within a finite expected number of steps. However, this processor's output is not consistent (the processor is not in C_f). This means that with this output, the system is at an inconsistent state and therefore (by Definition 4) there is an input that the scheduler can give the system for which it will err.

We have shown that the scheduler can force the system to advance along a sequence of non-t-founded nodes, and hence we have arrived at a contradiction in every possible case. This proves the $\Rightarrow$ direction of the theorem.

Now we will prove the $\Leftarrow$ direction; namely, if the root of the DAG corresponding to T is t-founded, then T is a t-resilient task. This is proven by presenting a generic t-resilient algorithm which implements T. In this algorithm, every processor will publish (in a shared register that the rest of the processors can read) its input as soon as it reads it and will also publish the output that it has chosen just before writing it in the output register. The frame of this algorithm will be a walk along a path in the DAG starting at its root, proceeding according to the inputs revealed and the produced outputs, and ending at a legal leaf that represents a full $2k$-tuple in T. This walk is executed commonly by all processors, and the way to achieve this is by applying consensus to every move.

There is a certain difference between the use of the DAG in this direction and its use in the previous direction. While in the first direction the current node represented the system's current state, in this direction the node represents a state the system "aspires" to. If a certain node, v, was agreed upon, it means that v represents input values that were published. However, not all the output values in v have necessarily been decided upon (and published) because they could belong to dormant processors. Such outputs will be decided upon and published by the respective processors when they are activated.

The algorithm we present uses as a subroutine an *extended consensus* protocol. This protocol allows consensus among processors when some of the participating processors do not offer a value but rather adopt one of the values offered by other processors. Processors that do not suggest values for the extended consensus protocol are called *passive processors* while processors that do offer a value for the extended consensus protocol are called *agile processors*. Given an $n-1$ resilient consensus protocol, and allowing β passive processors, we can build a $n-1-\beta$-resilient extended consensus protocol as follows:

- An agile processor will publish its offered value in a shared register and then proceed to execute the wait-free consensus protocol.
- A passive processor will wait until some agile processor suggests a value, adopt that value as its own, and then join the execution of the wait-free consensus protocol.

The algorithm will be carried out in "virtual rounds." In round $r+1$ each processor starts with node v_r, the t-founded node (in the DAG) that was agreed upon in round r (via consensus). Each processor then proceeds to pick, according to the DAG, a son of v_r representing a $2k$-tuple, S_{i+1}. This node, u_r, is a t-subsequent of v_r, is t-founded, and includes the processor's new input, if such an input is available (i.e., P_i has not yet published its jth input even though it has given its $(j-1)$st output). If the system is in the concluding phase and no such node, u_r, exists, the processor P will look for a t-founded son of v_r that includes an additional output for P. The processor proposes the node it has found to the rest of the processors to agree upon. Then, the processors run consensus to agree on one of the proposed options. Since the option includes outputs (not necessarily outputs of the originating processor, i.e., the one whose option was chosen), each processor now checks the agreed node to see if it is required to output a value. Each of the relevant processors will output its required value immediately after receiving the result, v_{r+1}, of the $(r+1)$st consensus. The processor then proceeds to the next virtual round. Formally the algorithm for processor P_i is as follows:

```
INIT:
  $v_r \leftarrow$ root(DAG).          /* $v_r$ is the current node in virtual round number $r$ */
  $a \leftarrow 1$                                      /* $a$ holds $P_i$'s current stage */
  $r \leftarrow 1$                                                    /* round counter */
  $in_i \leftarrow input$

VIRTUAL ROUNDS:
  do while $a \le k$
    denote $v_r$'s components by $< I^1_{A_1}, \dots, I^k_{A_k}, O^1_{B_1}, \dots, O^k_{B_k} >= v_r$
      if there exists $u_r =< I^1_{A'_1}, \dots, I^k_{A'_k}, O^1_{B'_1}, \dots, O^k_{B'_k} >$ s.t.
        $u_r$ is a son of $v_r$ and $u_r$ is $t$-founded
        and
          ( either
              $I^a_{A'_a} = I^a_{A_a} \cup \{in_i\}$     /* main or concluding phase - $in_i$ is $P_i$'s input */
              and $A'_j = A_j \;\; j \ne a$
              and $B_j \subseteq B'_j \;\; \forall j$
          or
              $B'_a = B_a \cup \{i\}$               /* concluding phase - only another output */
              and $B'_j = B_j \;\; j \ne a$
              and $A'_j = A_j \;\; \forall j$
          )
        then $v_{r+1} \leftarrow extended\text{-}consensus(u_r, r)$
        else $v_{r+1} \leftarrow extended\text{-}consensus(passive, r)$
      if $i \in B'_a$                     /* an additional output of $P_i$ was decided upon */
        then
          output($O^a_{\{i\}}$)                          /* output the corresponding value */
          if $a = k$
            then terminate
          $a \leftarrow a + 1$
          $in_i \leftarrow input$
      $r \leftarrow r + 1$
    end                                                              /* of while */
```

The different consensus rounds are separate and use separate sets of registers. We will use a multivalue consensus protocol with the addition of "passive inputs" (belonging to passive processors). The requirement from the protocol is that its output will be the input of one of the agile processors. Any known wait-free protocol for consensus (e.g., [1, 2, 9, 26]) can easily be modified to comply with this requirement. Notice that if up to β processors can be passive in a given round, then the modified consensus protocol is $(n-\beta-1)$-resilient. In the concluding phase, it is possible that the processors that have missing inputs are faulty, and so every remaining processor must be able to advance on its own (without waiting for values offered by other processors).

Let us now prove that the given algorithm is correct and t-resilient. According to the remark from the end of section 4, following a t-founded path leads to a leaf in T. This means (by the definition of a t-founded leaf in Definition 11) that the outputs given by the system match the inputs it received, according to the task T. Hence the algorithm always gives correct outputs for any legal input. As every call to the consensus subroutine takes bounded expected time, and the other operations are finite, within bounded expected time the algorithm will terminate. Therefore the algorithm is correct.

To show t-resiliency we will show that the system can always advance (even if t processors have failed, as long as some processor is active), i.e., there will always be an agile processor that can find a u_r as required. Notice that v_r is always t-founded (the root is t-founded by the assumption, and only t-founded nodes are chosen by the algorithm during the execution). Also, for a given node, the number of passive processors is less than $n-\alpha$ as at least α processors can find a u_r as required. Now let us show that there are indeed enough agile processors. We will do this for two different cases, according to the phase that the system is in:

- In the main phase and in the concluding phase when less than $n-t$ processors have terminated, $\alpha = t+1$. Therefore, even if t processors have failed, there exists at least one additional processor that can make a t-advancement from this t-founded node. This processor will be agile in the consensus round for v_r and thus the extended consensus will be computed (in bounded expected time) in this round.
- In the concluding phase when $n-t$ processors have already terminated, α equals the number of not-yet-terminated processors. Therefore, every processor that has not yet terminated and will be activated can find a u_r as required. Such a processor will be agile in the respective extended consensus round.

Altogether, we have shown that for every node on the path there are enough processors that can make a t-advancement from it. Thus, the algorithm is t-resilient, as claimed. ▯

6. Characterization for message passing. In this section we will examine the resiliency of interactive tasks in the message passing model, where it is possible to perform t-resilient consensus only for $t \leq \left\lfloor \frac{n-1}{2} \right\rfloor$. In the terminating message passing model a processor terminates after giving its final output. After $n-t$ processors have terminated, the remaining t processors could be disconnected from each other by the scheduler, and thus will have no way of "coordinating" amongst themselves. The disconnection is possible since all t processors might be faulty and so no processor can wait for a message from another processor. (Recall that in an asynchronous system the processors have no way of distinguishing between a slow and a faulty processor.) Each of the t slow processors will know what the fast $n-t$ processors have decided but will have no way of knowing what any one of the other slow processors has decided. In order for a task to be t-resilient, the slow processors must be able to decide on their value in a "consistent" manner, each without knowing the others' decisions. Notice that a particularly slow processor may be at the beginning of its work, i.e., it has not yet read even its first input. Such a processor must be able to function correctly through *all* the stages of the task. Following these considerations, we focus on nodes in the DAG, $D(T)$, that have at least $n-t$ outputs of the final stage revealed. We call these nodes *semiterminal* nodes. In these nodes each additional input *or* output is considered a t-advancement (by Definition 6). Notice that for semiterminal nodes we require at least $n-t$ *outputs* of the final stage, while in the concluding phase we required only $n-t$ *inputs* of the final stage.

DEFINITION 12. *A tree R_i that is a subgraph of $D(T)$ will be called* terminal *for P_i if the following conditions hold:*

- *The root of R_i, $v=[S,Q]$ where $S=\langle A_1 , \dots , A_k , B_1 , \dots , B_k\rangle$ and $Q=\langle I^1_{A_1}, \dots, I^k_{A_k}, O^1_{B_1}, \dots, O^k_{B_k}\rangle$, is a semiterminal node ($|B_k|\geq n-t$) such that $i\notin B_k$. (Processor P_i has not yet terminated.)*
- *Let ℓ denote the last stage for which P_i gave an output. When we look at the inputs and outputs for processor P_i, the following condition holds: For every input for the $(\ell+1)$st stage there is an output for the $(\ell+1)$st stage such that for every input for the $(\ell+2)$nd stage there is ... for every input for the kth stage there is an output for the kth stage such that the t-advancements corresponding to these inputs and outputs form a path in R_i. In the case that the $(\ell+1)$st input is already given in v, then the condition starts with the existence of an output for the $(\ell+1)$st stage such that for every input for the $(\ell+2)$nd stage, etc.*

Intuitively, the tree R_i represents a possible strategy for P_i to react to every possible sequence of inputs presented to it, when the system is at the state represented by the root, v. The next definition captures the requirement that terminal trees of different processors should be compatible.

DEFINITION 13. *Let v be a semiterminal node and let $\{i_1,\dots,i_j\}=\{1,\dots,n\}\backslash B_k$ (i.e., the set of processors that have not yet terminated in v). Let $R_{i_1},\dots,R_{i_j}$ be j terminal trees for processors $P_{i_1},\dots,P_{i_j}$, respectively. These trees are called T-compatible if the following conditions hold:*

- *$R_{i_1},\dots,R_{i_j}$ have the same semiterminal root v:*
  $v=[\langle A_1 , \dots , A_k , B_1 , \dots , B_k\rangle,\langle I^1_{A_1} , \dots , I^k_{A_k} , O^1_{B_1} , \dots , O^k_{B_k}\rangle]$.
- *For every $\langle I^1,\dots,I^k\rangle\in IN(T)$ which is an extension of $\langle I^1_{A_1},\dots,I^k_{A_k}\rangle$, the outputs produced along the set of paths (one path per tree) that correspond to this input give a full output vector $\langle O^1,\dots,O^k\rangle$ such that $\langle I^1,\dots,I^k, O^1,\dots,O^k\rangle\in T$ and $\langle O^1,\dots,O^k\rangle$ is an extension of $\langle O^1_{B_1} , \dots , O^k_{B_k}\rangle$.*

We use these definitions to describe our "building block," t-validity. It will play the role that t-foundedness played in the shared memory model (section 5).

DEFINITION 14. *Let*

$$v=[\langle A_1 , \dots , A_k , B_1 , \dots , B_k\rangle,\langle I^1_{A_1} , \dots , I^k_{A_k} , O^1_{B_1} , \dots , O^k_{B_k}\rangle]$$

be a node in the DAG $D(T)$, and let $\alpha=min\,(t+1 , n-|B_k|)$. We say that v is t-valid *relative to the task T when*

- *The node v is a leaf ($v=[\langle\{1,\dots,n\}_1 , \dots , \{1,\dots,n\}_{2k}\rangle, \langle I^1,\dots,I^k, O^1,\dots,O^k\rangle]$), and $\langle I^1,\dots,I^k,O^1,\dots,O^k\rangle\in T$;*

or

- *The node v is an internal node (not a leaf) and the following conditions hold:*
  1. *v has at least α non-t-equivalent sons.*

2. *For every t-subsequent, u, of v there exists an input-equal node u' that is a t-valid son of v.*
3. *If v is semiterminal, then for every $i \notin B_k$ there exists a terminal tree, R_i, for P_i, such that the trees $\{R_i\}_{i \notin B_k}$ are T-compatible. We will call one of these sets, say, the first in canonical order that complies with the above condition, the* chosen set *for v, denoted by $ch(v)$.*

The characterization theorem is formulated in the same way as Theorem 1, but the different definitions of t-foundedness versus t-validity make the difference between the shared memory and the message passing models. The additional requirement (3) explains why certain tasks, solvable in the shared memory model, cannot be solved in the message passing model.

THEOREM 2. *Consider the terminating message passing model, and let t satisfy $t \leq \lfloor \frac{n-1}{2} \rfloor$. An interactive$_k$ task, T, is t-resilient iff the root of its corresponding DAG is t-valid relative to T.*

Proof. The proof follows the proof of Theorem 1 closely. The differences between these proofs are due to the nature of the communication model. For the $\Rightarrow$ direction, we use an algorithm which, starting at a t-valid node, proceeds along a path of t-valid nodes until a semiterminal node is reached. The algorithm uses an *extended consensus* subroutine (see proof of Theorem 1) which guarantees that all the processors agree on the same nodes along the path. In the message passing model, the resiliency of the extended consensus protocol is $min(\lfloor \frac{n-1}{2} \rfloor, n-1-\beta)$, where β is an upper bound on the number of passive processors (processors that take part in the consensus protocol but do not suggest a value). In our algorithm, no processor terminates before a semiterminal node is reached. That is, no processor writes its final (kth) output before a node which includes the final outputs of at least $n-t$ processors has been agreed upon. This precaution guarantees that at least $n-t$ processors remain active throughout the main phase of the algorithm. This number is large enough to enable each round of the extended consensus algorithm to terminate, as there is at least one agile processor in each round, and at least $\lfloor \frac{n+1}{2} \rfloor$ processors take part in the sequence of consensus executions until a semiterminal node is reached and agreed upon.

We associate with every t-valid semiterminal node, v, its chosen set, $ch(v)$ (the existence of this set follows from Definition 14). Once the system reaches and agrees upon such a semiterminal node, all processors P_i with $i \in B_k$ can produce their final output and terminate. Each remaining processor can now produce a local output as a response to every new local input it receives by proceeding along the corresponding path in its tree. This requires no communication and coordination with the remaining processors. The compatibility of the terminal trees guarantees that this process leads to a leaf satisfying the input-output relations of the task T. In order to simplify the algorithm, we will use a pruned DAG, denoted by $D'(T)$. This DAG will be the same as D(T) for all nodes that are not semiterminal. For semiterminal nodes, $D'(T)$ will include only their chosen set. That is, for semiterminal node v, $D'(T)$ will include only $ch(v)$ (and not other alternative advancements from v). The algorithm will run on $D'(T)$ rather than $D(T)$.

Formally, the algorithm for P_i is as follows.

```
INIT:
  $v_r \leftarrow$ root($D'(T)$).          /* $v_r$ is the current node in virtual round number $r$ */
  $a \leftarrow 1$                                          /* $a$ holds $P_i$'s current stage */
  $r \leftarrow 1$                                                        /* round counter */
  $in_i \leftarrow input$
```

```
VIRTUAL ROUNDS:
 do while $a \leq k$
   denote $v_r$'s components by $< I^1_{A_1}, \ldots, I^k_{A_k}, O^1_{B_1}, \ldots, O^k_{B_k} >= v_r$
     if there exists $u_r =< I^1_{A'_1}, \ldots, I^k_{A'_k}, O^1_{B'_1}, \ldots, O^k_{B'_k} >$ s.t.
       $u_r$ is a son of $v_r$ and $u_r$ is $t$-valid
       and
         ( either
             $I^a_{A'_a} = I^a_{A_a} \cup \{in_i\}$          /* main phase - $in_i$ is the new input */
             and $A'_j = A_j$  for all $j$, $j \neq a$
             and $B_j \subseteq B'_j$ for all $j$
         or
             $B'_a = B_a \cup \{i\}$          /* concluding phase - only another output */
             and $B'_j = B_j$ for all $j$, $j \neq a$
             and $A'_j = A_j$ for all $j$
         )
       then if $|B_k| < n - t$/* consensus is possible until $n - t$ processors terminate */
             then $v_{r+1} \leftarrow extended - consensus(u_r, r)$
             else $v_{r+1} \leftarrow u_r$        /* no consensus - proceed along terminal tree */
       else $v_{r+1} \leftarrow extended - consensus(passive, r)$   /* this case is only possible
                                                                   in the main phase */
     if $i \in B'_a$                 /* an additional output of $P_i$ was decided upon */
       then (if $a < k$
           then
             output($O^a_{\{i\}}$)                  /* output the corresponding value */
             $a \leftarrow a + 1$
             $in_i \leftarrow input$
           elseif $|B'_k| \geq n - t$
             then
               output($O^a_{\{i\}}$)                /* output the corresponding value */
               terminate
     $r \leftarrow r + 1$
   end                                                              /* of while */
```

The consensus protocol used by the extended consensus must be $\lfloor \frac{n-1}{2} \rfloor$-resilient and terminating. The different consensus rounds are separate and use separate message numbers. (Any consensus protocol that complies with these requirements can be used as a subroutine. Examples of such consensus protocols are [7, 6]). Notice that passive processors are counted for the number of processors participating in the consensus as they can "help out." However, at least one agile processor must participate in the consensus (to suggest the next value). There will always be such a processor for the same reasons as in the shared memory model (in the main phase at least $\alpha = t+1$ processors can find the required u_r as the current node is always t-valid, and in the concluding phase there is no need for consensus).

The $\Leftarrow$ direction is also similar to the shared memory case. As in Claim 1, we can assume that the algorithms used are immediate input algorithms. (The proof for the message passing model is identical to that of the shared memory case.) We present a scheduler that forces the system to always remain in a non-t-valid node. By Definition 14, the possible cases at node v (which is non-t-valid) are

0. v is a non-t-valid leaf;
1. v has less than α non-t-equivalent sons;
2. v has a t-subsequent, v_1, such that for every u' that is input-equal to v_1 and is a son of v, it holds that u' is *non-t*-valid;
3. v is semiterminal and one of the following is true:
   (a) There exists $i \notin B_k$ for which there is no terminal tree rooted at v.
   (b) Every collection of terminal trees $\{R_i\}_{i \notin B_k}$ rooted at v is not compatible.

As in the shared memory model, let us assume toward a contradiction that there exists a t-resilient immediate-input algorithm for T. We will start at the root of the DAG and the processors will follow this algorithm. As long as the node representing the system's current state, v, is non-t-valid due to possibility 2, the scheduler forces the processors to advance to a state represented by u' that is input-equal to v_1. This will continue until we reach a node v_f that is non-t-valid either due to possibility 1 or possibility 3 or because v_f is a non-t-valid leaf. If v_f is non-t-valid due to possibility 1 or is a leaf, then for exactly the same reasons as in the proof for the shared memory model we have shown a contradiction.

It remains to show that the other alternative (non-t-validity due to possibility 3) also leads to contradiction. The first case is when there exists an i for which there is no terminal tree. By Definition 12, this means that there exists a strategy for the scheduler (supplying inputs) such that every strategy for the processor P_i leads to a partial $2k$-tuple which is not a partial vector of any full $2k$-tuple in T. This contradicts the correctness of the algorithm.

The second case is when each selection of terminal trees is not compatible (i.e., every set of terminal trees, one for each active processor, is incompatible). In this case the scheduler activates these active processors and withholds all messages sent between them. Now each one must act on its own. Let us now show that there is a strategy for the scheduler which causes the processors to err with positive probability. In order to do that we will construct one set of terminal trees for all the remaining active processors, using a "roll-back" technique. The construction starts with the system at the state in which it arrived at the semiterminal node v. Now for every processor P_i where $i \notin B_k$, the tree T_i is built recursively, as follows: For every node, u, in the tree (starting the recursion at v), the scheduler determines the son by presenting the processor P_i with a legal input at u and the grandson by activating P_i until it produces an output (which must happen within bounded time, by the requirements of t-resilient algorithms). The process is repeated in the next recursion level, with respect to the grandson. To determine the continuation from u with respect to other inputs, the scheduler "rolls back" the system to the same state it was in u and then supplies another input. This process is done with respect to every legal input for P_i at u. The end of the recursion along each path is when P_i supplies its last (kth) output.

This process builds a set of terminal trees. Every branch along each tree represents a strategy for the corresponding processor, which is used with positive probability. Probabilities of different processors are independent, since their random inputs are independent, and the processors cannot communicate. By our assumption, the terminal trees are incompatible. Therefore, there exist input sequences (one per processor) such that the output sequences resulting by following the corresponding trees yield a full input-output $2k$-tuple that is not in T. The scheduler will supply these inputs and then the processors will give incorrect answers (due to the definition of incompatible terminal trees) with positive probability. This contradicts the assumption that the algorithm implements T with no error. Therefore the condition in the theorem is necessary. $\square$

7. Concluding remarks. Denote by REF the class of randomized error-free protocols and by DC the class of deterministic protocols with access to consensus. Since consensus has wait-free REF solution, it follows that any finite interactive task solvable in the DC model is also solvable in the REF model. The proof of Theorems 1 (for the shared memory model) and 2 (for the message passing model) implies that any finite interactive task solvable in the REF model is also solvable in the DC model. Stated formally, we have the following theorem.

THEOREM 3. *Consider the following two families of protocols:*

- *deterministic protocols with access to consensus (DC),*
- *randomized error-free protocols (REF).*

An interactive$_k$ task T is t-resilient in the DC family iff it is t-resilient in the REF family, where the range of t is

- $0 \leq t \leq n-1$ *in the terminating shared memory model,*
- $0 \leq t \leq \left\lfloor \frac{n-1}{2} \right\rfloor$ *in the terminating message passing model.*

Results concerning the structure of the "resiliency hierarchy" in both the shared memory and the message passing models can easily be inferred from our characterizations. They extend similar results for decision tasks, proven by Chor and Moscovici in [13]. Some of these implications are as follows:

- Shared memory is strictly more powerful than message passing for the same resiliency (except at the two lowest resiliencies): $IMP_t \subsetneq ISM_t$ $(1 < t < n)$, while $IMP_0 = ISM_0$ and $IMP_1 = ISM_1$.
- With respect to difference resiliencies, message passing and shared memory, namely IMP_t and ISM_{t+i}, are incompatible for $0 < i < n-t$. (The results in [13] already prove this.)
- In the range of resiliency where network partition is not possible ($t \leq \left\lfloor \frac{n-1}{2} \right\rfloor$), *nonterminating* protocols in the shared memory and message passing models have the same capabilities: namely, nonterminating $IMP_t =$ nonterminating $ISM_t = ISM_t$. For t in the range $\frac{n}{2} \leq t \leq n-1$, nonterminating $ISM_t = ISM_t$.

7.1. Related work. Bar-Noy and Dolev [6], and consequently Attiya, Bar-Noy and Dolev [4], have investigated emulation strategies of shared memory in nonterminating message passing systems. One consequence of their work is that $ISM_{n-1} \subseteq$ nonterminating $IMP_{\left\lfloor \frac{n-1}{2} \right\rfloor}$. In fact, their construction yields the stronger result $ISM_{\left\lfloor \frac{n-1}{2} \right\rfloor} \subseteq$ nonterminating $IMP_{\left\lfloor \frac{n-1}{2} \right\rfloor}$. However, no characterization can be derived from their techniques. Also, it is not clear if their "local" approach can be extended to yield the equality between these classes (which does follow from our characterization).

Plotkin [23] and Herlihy [17] have studied the implementation of concurrent objects in the shared memory model. They have shown that every sequential system has a concurrent wait-free implementation, given access to a wait-free consensus subroutine. We note that our results give a natural way to implement finite versions of concurrent objects, and so they shed light on the use of randomization for wait-free concurrent objects. In fact, the concurrent objects in [17] refer to $t = n-1$ (wait-free objects) while interactive tasks can implement a wider range of fault-tolerance.

Taubenfeld, Katz, and Moran [28] and Taubenfeld and Moran [29] have studied a weaker type of crash failures—initial faults. In the initial faults model, each processor either is initially crashed or remains active forever. Taubenfeld, Katz, and Moran found necessary and sufficient conditions for solvability of distributed *decision* tasks with respect to deterministic protocols in the initial fault message passing and shared memory models. Surprisingly, the characterization is the same as the one for general crash faults using randomized protocols [13]. Indeed, these equalities no longer remain valid when interactive, rather than decision, tasks are considered. This is due to the scheduler's limited powers in the initial faults model. Therefore, it is not surprising that there are interactive tasks that are t-resilient in the initial fault model but are not t-resilient for general crash faults. As an example, consider the following interactive$_2$ task: Inputs for both stages are all binary n-vectors. The value in each component of the first output equals the sum of the first inputs over some subset S of size $n - t$ (the subset is not predetermined). The value in each component of the second output equals the sum of the second inputs over the *same* subset S. It is easily seen that this task is resilient to t *initial* faults but is not in ISM_t since in ISM_t the scheduler can fail a previously active processor, thus leaving the second input unknown. This confirms the intuition that initial faults alone are not sufficient to represent a realistic fault model. (In the other direction, every task in ISM_t can be solved in a system with at most t crashes that can implement a consensus subroutine. Since consensus is solvable deterministically in the initial faults model, this implies that ISM_t is strictly contained in the class of t initial faults.)

The modular way in which consensus is used implies that our characterization is also a sufficient condition for t-resilience in *deterministic* systems augmented with *stronger* mechanisms that make consensus possible, for example, the failure detectors of Chandra and Toueg [10]. However, the question whether our conditions are also necessary requires a closer look. For example, suppose one has at his possession an accurate and reliable failure detector. If the detector announces that a processor has crashed, then we are guaranteed this processor will not become active later. In such a case we may be able to solve a natural modification of, for example, the parity task. We assign $\perp$ as the input of processors that crashed before supplying an input, and the $\perp$ values do not influence the output. Under such conditions, the parity is no longer unsolvable in the presence of one failure. This example implies that exact characterization of resilience in the presence of failure detectors strongly depends on the specific properties of the detector.

Despite substantial differences, there is a common feature to the initial faults model and the one with strong failure detectors. In both, the power of the adversary is severely restricted, further than what is "needed" to enable deterministic consensus. Any task which satisfies our characterization for t-resilience will also be t-resilient in these restricted adversary models. But whether the condition is also necessary crucially depends on the strength (or weakness) of the adversary.

Acknowledgments. Thanks to Hagit Attiya and Oded Goldriech for helpful discussions and suggestions and for their comments on earlier versions of this manuscript, to Moshi Molcho for his help and support, and to the anonymous referees for helpful comments.

REFERENCES

[1] J. Aspnes, *Time- and Space-Efficient Randomized Consensus*, J. Algorithms, 14 (1993), pp. 414–431.

[2] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[3] J. Aspnes and O. Waarts, *Randomized Consensus in Expected $O(n \log^2 n)$ Operations per Processor*, SIAM J. Comp., 25 (1996), pp. 1024–1044.

[4] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message passing systems*, J. ACM, 42 (1995), pp. 124–142.

[5] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, *Renaming in an asynchronous environment*, J. ACM, 37 (1990), pp. 524–548.

[6] A. Bar-Noy and D. Dolev, *A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment*, Math. Systems Theory, 26 (1993), pp. 21–39.

[7] M. Ben-Or, *Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols*, in Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983, pp. 27–30.

[8] E. Borowsky and E. Gafni, *Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations*, in Proceedings of the 25th Symposium on the Theory of Computing, 1993, pp. 91–100.

[9] G. Bracha and O. Rachman, *Randomized Consensus in Expected $O(n^2 \log n)$ Operations*, TR–671, Technion, Haifa, Israel, 1991.

[10] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, J. ACM, 43 (1996), pp. 225–267.

[11] B. Chor, A. Israeli, and M. Li, *Wait-Free Consensus Using Asynchronous Hardware*, SIAM J. Comput., 23 (1994), pp. 701–712; conference version in Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, 1987, pp. 86–97.

[12] B. Chor, M. Merritt, and D. Shmoys, *Simple constant time consensus protocols in realistic failure models*, J. ACM, 36 (1989), pp. 591–614.

[13] B. Chor and L. Moscovici, *Solvability in Asynchronous Environments*, in Proceedings of the 30th Symposium on Foundations of Computer Science, 1989, pp. 422–427.

[14] B. Chor and L. Nelson, *Resiliency of Interactive Distributed Tasks*, in Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, 1991, pp. 37–49.

[15] D. Dolev, Dwork C., and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. ACM, 34 (1987), pp. 77–97.

[16] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distriburted consensus with one faulty process*, J. ACM, 32 (1985), pp. 374–382.

[17] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Languages Systems, 13 (1991), pp. 124–149.

[18] M. Herlihy and N. Shavit, *The Asynchronous Computability Theorem for t-Resilient Tasks*, in Proceedings of the 25th Symposium on the Theory of Computing, 1993, pp. 111–120.

[19] L. Lamport, *On interprocess communication*, Distrib. Comput., 1 (1986), pp. 77–101.

[20] D. Lehmann and M. Rabin, *On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem*, in Proceedings of the 8th Principles of Programming Languages, 1981, pp. 133–138.

[21] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, in Advances in Computing Research, JAI press, Greenwich, CT, 1987, pp. 163–183.

[22] S. Moran and Y. Wolfstahl, *Extended impossibility results for asynchronous complete networks*, Inform. Process. Lett., 26 (1987), pp. 145–151.

[23] S. Plotkin, *Sticky Bits and the Universality of Consensus*, in Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, 1989, pp. 159–176.

[24] M. Rabin, *Randomized Byzantine Generals*, in Proceedings of the 24th Symposium on Foundations of Computer Science, 1983, pp. 403-409.

[25] M. Rabin, *The choice coordination problem*, Acta Inform., 17 (1984), pp. 121–134.

[26] M. Saks, N. Shavit, and H. Woll, *Optimal Time Randomized Consensus—Making Resilient Algorithms Fast in Practice*, in Proceedings of the 2nd ACM Symposium on Discrete Algorithms, 1991, pp. 351–362.

[27] M. Saks and F. Zaharoglou, *Wait-Free k-set Agreement is Impossible: The Topology of Public Knowledge*, in Proceedings of 25th Symposium on the Theory of Computing, 1993, pp. 111–120.

[28] G. Taubenfeld, S. Katz, and S. Moran, *Initial failures in distributed computations*, Internat. J. Parallel Programming, 18 (1989), pp. 255–276.

[29] G. Taubenfeld and S. Moran, *Possibility and impossibility results in a shared memory environment*, Acta Inform., 33 (1996), pp. 1–20.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 378–386

WEAKLY TRIANGULATED COMPARABILITY GRAPHS[*]

ELAINE ESCHEN[†], RYAN B. HAYWARD[‡], JEREMY SPINRAD[§], AND R. SRITHARAN[¶]

Abstract. The class of weakly triangulated comparability graphs and their complements are generalizations of interval graphs and chordal comparability graphs. We show that problems on these classes of graphs can be solved efficiently by transforming them into problems on chordal bipartite graphs. We show that recognition and independent set on weakly triangulated comparability graphs can be solved in $O(n^2)$ time in this manner, and that the number of weakly triangulated comparability graphs is $2^{\Theta(n\log^2 n)}$. We also give algorithms to compute transitive closure and transitive reduction in $O(n^2\log\log n)$ time if the underlying undirected graph of the transitive closure is a weakly triangulated comparability graph.

Key words. weakly triangulated, comparability, partial orders, algorithm

AMS subject classifications. 05C70, 05C85, 06A06, 68Q25

PII. S0097539797322334

1. Introduction.

A variety of interesting graph classes correspond to requiring that the graph and/or its complement be a triangulated or a comparability graph. For example, permutation graphs are equal to graphs which are both comparability and cocomparability graphs, split graphs correspond to triangulated cotriangulated graphs, and interval graphs are exactly the chordal cocomparability graphs. These two properties are one of the themes which unify Golumbic's book on perfect graph classes [17]. We extend the study of the intersection of pairs of properties by generalizing from triangulated graphs to weakly triangulated graphs.

Triangulated graphs [28], also known as chordal graphs, are those graphs which contain no induced cycles of length greater than three. Weakly triangulated graphs allow induced four-cycles but do not allow any induced cycles of length greater than four in either the graph or its complement. We refer to an induced cycle of length five or more as a *hole*. We refer to the complement of a hole as an *antihole*. Weakly triangulated graphs properly contain chordal graphs and are a subclass of perfect graphs [18]. The fastest known recognition algorithm for weakly triangulated graphs takes $O(n^4)$ time [30]. The best algorithms for clique, independent set, chromatic number, and clique cover on weakly triangulated graphs also take $O(n^4)$ time [19, 3].

Comparability graphs [16], also known as transitively orientable graphs, are undirected graphs with the property that directions can be assigned to edges in such a way that whenever there is an edge from x to y and from y to z in the directed graph, there is also an edge from x to z. This assignment of direction to edges is called a

[*]Received by the editors June 2, 1997; accepted for publication (in revised form) June 25, 1998; published electronically October 5, 1999.
http://www.siam.org/journals/sicomp/29-2/32233.html

[†]Mathematics and Computer Science Department, Fisk University, 1000 17th Avenue North, Nashville, TN 37208 (eschen@dubois.fisk.edu). The work of this author was supported by the Kellogg Foundation grant "Building on Excellence in the Sciences" at Fisk University.

[‡]Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, AB T1K 3M4, Canada (hayward@cs.uleth.ca). The work of this author was supported by the Natural Science and Engineering Research Council of Canada.

[§]Department of Computer Science, Vanderbilt University, Nashville, TN 37235 (spin@vuse.vanderbilt.edu).

[¶]Department of Mathematics and Computer Science, Indiana State University, Terre Haute, IN 47809 (rst@berge.indstate.edu). The work of this author was supported by the University Research Committee, Indiana State University.

transitive orientation. Comparability graphs are also a subset of perfect graphs. The fastest known recognition algorithm for comparability graphs has two distinct stages. It is possible to find an orientation which is transitive if and only if the input is a comparability graph in $O(n+m)$ time [25]. However, it is not known how to determine whether this orientation is actually transitive faster than performing a matrix multiplication, for which the current asymptotically fastest algorithm takes $O(n^{2.376})$ time [7]. There are $O(n+m)$ algorithms for clique and chromatic number [17, 25] on comparability graphs, if a transitive orientation is given. Maximum independent set on comparability graphs is essentially equivalent to bipartite matching [13], for which the best known algorithms have complexities of $O(n^{1.5}\sqrt{m/\log n})$ [2] and $O(\frac{n^{2.5}}{\log n})$ [11].

One of the most famous classes of intersection graphs is the class of interval graphs [4, 5]. A graph G is an interval graph if and only if both G is a chordal graph and the complement of G is a comparability graph [16]. Since weakly triangulated graphs are a natural generalization of triangulated graphs, we are interested to see whether the weakly triangulated cocomparability graphs are well behaved. Since weakly triangulated graphs are closed under complementation, this class is exactly the complement of weakly triangulated comparability graphs.

Weakly triangulated comparability graphs generalize the triangulated comparability graphs, studied in [23]. We note that both triangulated comparability graphs and triangulated cocomparability graphs can be recognized in linear time, which is faster than the algorithm obtained by separately testing whether G is triangulated and whether G is comparability/cocomparability. For weakly triangulated comparability and weakly triangulated cocomparability graphs, we give recognition algorithms which are faster than testing either whether G is weakly triangulated or whether G is a comparability/cocomparability graph.

Another class contained in the weakly triangulated comparability graphs is the class of permutation graphs [17]. This follows from the fact that cocomparability graphs cannot contain induced cycles of length greater than four [15], and permutation graphs are equal to graphs which are both comparability and cocomparability graphs.

Both the classes of comparability graphs and triangulated graphs contain $2^{\Theta(n^2)}$ graphs on n vertices, but the classes of triangulated comparability graphs and triangulated cocomparability graphs contain $2^{\Theta(n\log n)}$ graphs on n vertices. We show that the number of weakly triangulated comparability graphs on n vertices is $2^{\Theta(n\log^2 n)}$, which makes this class much smaller, for example, than the triangulated cotriangulated graph class, which has $2^{\Theta(n^2)}$ members.

The primary method used here to design efficient algorithms for weakly triangulated comparability graphs is to convert them into bipartite graphs and use the property that this transformation yields a chordal bipartite graph. The ideas are similar to those used in recognizing circular-arc graphs [10] and trapezoidal graphs [24] efficiently. Chordal bipartite graphs, studied in [21], are bipartite graphs in which every cycle of length greater than four has a chord. The bipartite adjacency matrix of a graph is the 0/1 matrix formed by making one color class the rows, the other color class the columns, and placing a 1 at row x column y if and only if (x,y) is an edge of G. A 0/1 matrix has a Γ if there is some pair of rows $r_1 < r_2$ and columns $c_1 < c_2$ such that $(r_1,c_1) = (r_1,c_2) = (r_2,c_1) = 1$, while $(r_2,c_2) = 0$. The crucial theorem for this paper is that a graph is chordal bipartite if and only if the rows and columns of the bipartite adjacency matrix can be permuted to form a Γ-free matrix, and this ordering can be found in $O(m\log n)$ or $O(n^2)$ time [21, 27, 32].

2. Recognition. Let G be an arbitrary directed graph. Consider the undirected bipartite graph G' formed by making two copies x_1, x_2 of each vertex x and adding an edge from u_1 to v_2 if and only if there is an edge from u to v in G. We will call G' the *bipartite transformation* of G. Identical constructions have been made frequently; the most closely related work that uses this construction is due to Ford and Fulkerson [13], who use it for solving the independent set problem on partially ordered sets.

THEOREM 1. *If G is a transitively oriented graph, then the underlying undirected graph of G is weakly triangulated if and only if the bipartite transformation G' of G is chordal bipartite.*

Proof. First, we note that complements of comparability graphs cannot have induced cycles of length greater than four [15]. Thus, we do not have to be concerned with long antiholes in the underlying undirected graph of G.

Suppose that the underlying undirected graph of G is weakly triangulated. Consider a chordless cycle C of length at least six in G'. There cannot be a vertex x such that both x_1 and x_2 are in C; otherwise, there would be edges (y_1,x_2) and (x_1,u_2) in C, where $u_2 \neq y_2$ and (y_1,u_2) is not in C, and thus transitivity in G would imply that C has the chord (y_1,u_2). If x_1 and y_1 are both in C, then there cannot be an edge from x to y in G; otherwise, since G is transitively oriented, the neighborhood of x_1 would contain the neighborhood of y_1, and the vertices could not be part of the same chordless cycle. Similarly, if x_2 and y_2 are both in C, then there cannot be an edge from x to y in G. There cannot be any nonadjacent vertices u_1, v_2 in C such that there is an edge from v to u in G; otherwise, the edge (w_1,v_2) of C would imply an edge from w to u in G. Therefore, any chordless cycle C of G' corresponds to a chordless cycle of the same length in the underlying undirected graph of G; if the underlying graph of G is weakly triangulated, then G' must be chordal bipartite.

Suppose that G' is chordal bipartite. Let C be a chordless cycle in the underlying undirected graph of G. No vertex of C can have an edge directed to vertex u of C and an edge directed into it from vertex v of C, since v would have an edge to u by transitivity of the orientation. Let 1 be a vertex of C with edges directed to other vertices of C. If the cycle is $1, 2, \ldots, k$, then $1, 3, \ldots$ have edges directed outward with respect to other vertices of C, while $2, 4, \ldots$ have edges directed towards them. There will be a chordless cycle $1_1,\ 2_2,\ 3_1,\ 4_2, \ldots,\ (k-1)_1, k_2, 1_1$ in G', contradicting our assumption that G' is chordal bipartite. $\square$

Theorem 1 immediately reduces the complexity of weakly triangulated comparability graph recognition to the time of comparability graph recognition and orientation plus the time of chordal bipartite graph recognition. Since comparability graphs can be recognized and transitively oriented in $O(n^{2.376})$ time using matrix multiplication [31], and chordal bipartite graphs can be recognized in $O(n^2)$ time [21, 27, 32], we have an $O(n^{2.376})$ algorithm for recognizing weakly triangulated comparability graphs. If fast matrix multiplication is not desired due to its complexity, we get an $O(n^3)$ recognition algorithm. These time complexities are an improvement over existing algorithms, since the best known algorithms for recognizing weakly triangulated graphs take $\Theta(n^4)$ time [30].

However, we can improve on these time bounds using properties of the comparability graph recognition algorithm. It is possible to find a transitive orientation of a comparability graph in $O(n+m)$ time [25]; the reason that this does not give a linear time algorithm for recognizing comparability graphs is that the algorithm will assign some direction to edges, even if no transitive orientation is possible. Therefore, we will use the following strategy for recognizing weakly triangulated comparability graphs.

We find an orientation of the graph which is transitive if and only if the graph has a transitive orientation. We determine whether the bipartite transformation of the directed graph is chordal bipartite; if not, then we know that G is either not a comparability graph or G is not weakly triangulated. We then use properties of chordal bipartite graphs, and use the bipartite transformation to verify that the orientation is transitive. If the orientation is transitive, we use Theorem 1 to answer that G is a weakly triangulated comparability graph.

We construct a Γ-free ordering of the bipartite transformation, which can be done in $O(n^2)$ time [32]. Algorithms for constructing a Γ-free ordering work by using a technique called doubly lexical ordering of the matrix. Details of the procedure are not important to this paper, but if the neighborhood of a vertex x properly contains the neighborhood of vertex y, then doubly lexical ordering guarantees that the column or row corresponding to x comes after the column or row corresponding to y in the Γ-free ordering of the bipartite adjacency matrix.

THEOREM 2. *Suppose that the bipartite transformation of a directed graph G gives a chordal bipartite graph G'. We can determine whether G is transitive in* $O(n^2)$ *time.*

Proof. For each pair of vertices x, y we can determine whether there is a transitivity violation $x \to y \to z$ in constant time. If x does not have an edge to y, clearly no such violation exists. So suppose there is an edge from x to y and x_1 comes before y_1 in the doubly lexical ordering of G'. There must be a vertex z such that $y \to z$ while x has no edge to z, or the neighborhood of y_1 would be properly contained in the neighborhood of x_1 (the containment is proper since x_1 has an edge to y_2 while y_1 does not), so we can immediately answer that G is not transitive. Now suppose that x_1 comes after y_1 in the ordering. Let z_2 be the first column of the Γ-free ordering of G' which has a 1 in row y_1. If x_1 does not have a 1 in this position, there is a transitivity violation $x \to y \to z$. If there is a 1 in this position, then there is no transitivity violation $x \to y \to u$, or the rows x_1, y_1 and columns z_2, u_2 would form a Γ. ◻

COROLLARY 3. *Weakly triangulated comparability graphs can be recognized in* $O(n^2)$ *time.*

Proof. Given a graph G, we find an orientation of the edges which is transitive if and only if G is a comparability graph using the algorithm of [25, 31]. We construct the bipartite transformation G' of the directed version of G. By Theorem 1, if G is in the class, then G' must be a chordal bipartite graph; we test this in $O(n^2)$ time using the algorithms in [21, 32]. If G' is chordal bipartite, we use Theorem 2 to test whether the orientation is actually transitive. If the orientation is transitive and G' is chordal bipartite, we use Theorem 1 to conclude that G is a weakly triangulated comparability graph. ◻

COROLLARY 4. *The number of weakly triangulated comparability graphs is* $2^{\Theta(n \log^2 n)}$.

Proof. Every chordal bipartite graph G is clearly a weakly triangulated comparability, since there are no induced cycles of length greater than four in G, and there cannot be any set of three vertices from a single color class in any hole of the complement. Any weakly triangulated comparability graph G can be reconstructed from the bipartite transformation of the transitive orientation of G, which is a chordal bipartite graph on $2n$ vertices. Therefore, the number of weakly triangulated comparability graphs on n vertices is some number between the number of chordal bipartite graphs on n vertices and the number of chordal bipartite graphs on $2n$ vertices. Since there are $2^{\Theta(n \log^2 n)}$ chordal bipartite graphs on n vertices [29], the result follows. ◻

3. Independent set, transitive closure, and transitive reduction. We know that there are polynomial algorithms for solving the clique, independent set, chromatic number, and clique cover problems on weakly triangulated comparability graphs, since such algorithms exist for these problems on both comparability graphs and weakly triangulated graphs. Clique and chromatic number can be solved in linear time on comparability graphs [17, 25], so we cannot hope to do better for these problems. Since these graphs are perfect, and the size of the largest independent set in G is equal to the clique cover number of G, we will consider only the independent set problem.

The best known algorithm for solving independent set on weakly triangulated graphs has time complexity $\Theta(n^4)$ [3, 19] and the best algorithm for computing the maximum independent set on comparability graphs has the same time complexity as finding a maximum matching in a bipartite graph [13], as described below. We show that this problem can be solved in $\mathrm{O}(n^2)$ time on weakly triangulated comparability graphs, as a simple consequence of Theorem 1. We note that it will be difficult to improve the time bound for independent set on general comparability graphs, since every bipartite graph is a comparability graph, and any improvement in the time complexity of computing maximum independent set on comparability graphs will improve the time complexity of computing the cardinality of a maximum matching in bipartite graphs.

The best algorithm for solving the independent set problem on comparability graphs is most easily phrased as solving the vertex cover problem on the class; to solve the maximum independent set problem, we simply take the vertices which are not part of the minimum vertex cover.

The algorithm for solving this problem takes a transitive orientation of the graph G, performs the bipartite transformation, and solves the vertex cover problem on the bipartite transformation G' using well known ideas for transforming bipartite vertex cover problem to bipartite matching problem. A vertex x is placed in the vertex cover of G if and only if either x_1 or x_2 is in the minimum vertex cover of G'; this will be the minimum vertex cover of G. The bottleneck step is solving the maximum matching problem on G'; the current best time bounds are $\mathrm{O}(n^{1.5}\sqrt{m/\log n})$ [2] and $\mathrm{O}(\frac{n^{2.5}}{\log n})$ [11].

For weakly triangulated comparability graphs, we make the same transformation, but instead of using a general matching algorithm we use the fact that G' is chordal bipartite, and use a simpler matching algorithm for G'. It is known that a maximum cardinality matching in a chordal bipartite graph can be found in linear time, given a Γ-free ordering of the bipartite adjacency matrix of the graph [6]. We include the algorithm here for the sake of completeness.

LEMMA 5. *The maximum cardinality matching problem on a chordal bipartite graph can be solved in* $\mathrm{O}(n^2)$ *time.*

Proof. We create a Γ-free ordering of the bipartite adjacency matrix for the chordal bipartite graph. We match the vertex corresponding to the first row with the vertex whose column corresponds to the first 1 entry in that row (if such a column exists), and delete that row and column. We then continue to the next row and repeat until all the rows and columns have been deleted. Consider any row r and column c which we choose for our matching. Suppose that we cannot choose to match these in a maximum cardinality matching that includes all previous matches. If r or c is unmatched in the optimal matching, then matching r with c cannot decrease the size of the matching. If r is matched with c_2 and c is matched with r_2, there must be an edge (r_2, c_2) or rows r, r_2 and c, c_2 induce a Γ. Therefore, we can match r with c and r_2 with c_2 and still get an optimum matching. ☐

COROLLARY 6. *The independent set problem can be solved in* $O(n^2)$ *time for weakly triangulated comparability graphs.*

Proof. Use Ford and Fulkerson's transformation [13] to reduce this to bipartite matching. Theorem 1 shows that this yields a chordal bipartite graph, and Lemma 5 lets us solve this problem in $O(n^2)$ time. ☐

Transitive closure is normally defined on a directed acyclic graph G; an edge from x to y is part of the transitive closure of G if there is a directed path of length ≥ 1 from x to y in G. The best algorithms known for general transitive closure involve matrix multiplication; relationships between matrix multiplication and transitive closure are examined in [12, 14, 26].

We will solve the transitive closure problem efficiently for any directed acyclic graph G with the property that the underlying undirected graph of the transitive closure is a weakly triangulated comparability graph. This problem can be solved in $O(n+m_t)$ time for chordal comparability graphs [23], where m_t is the number of edges of the transitive closure. We show that the problem can be solved in $O(n^2\log\log n)$ time for weakly triangulated comparability graphs.

The algorithm most closely resembles the algorithm for transitive closure of two dimensional partial orders [22]. The fundamental idea is to use divide and conquer. The set of vertices is partitioned into two sets S_1, S_2 of size $\frac{n}{2}$ with the property that no edge goes from S_2 to S_1; the transitive closure is solved recursively within each set. The key task is to add edges which are implied by transitivity from S_1 to S_2; this is done by using the fact that the transitive closures of the two pieces can be transformed to chordal bipartite graphs using the bipartite transformation.

LEMMA 7. *Suppose that G is a directed acyclic graph such that the underlying undirected graph is a weakly triangulated comparability graph. Assume that we are given the doubly lexical ordering of the bipartite transformation of G, and adjacency lists of each vertex ordered by column number in the ordering. Let x be a vertex which has edges to some vertices of G. We can find all edges out of x in the transitive closure of $\{x\} \cup G$ in* $O(n\log\log n)$ *time.*

Proof. Let $Y = y_1, y_2, \ldots, y_k$ be the vertices of G which have an edge from x. We place the vertices of Y in a priority queue, where the value associated with y_i is the column number of the first 1 in row y_i.

We repeatedly look at the vertex y_i with the smallest value in the priority queue; let z be the vertex with the column number that is currently associated with y_i. Suppose there is another vertex y_j on the queue with the same priority value as y_i. One of the vertices $\{y_i, y_j\}$ has smaller row number in the bipartite transformation; call the vertex with smaller row number y_a and the vertex with larger row number y_b. Since we are given a Γ-free ordering, for any vertex z' which comes after z in the ordering, if y_a has an edge to z', then y_b has an edge to z'. Therefore, we can eliminate y_a from our priority queue when looking for later edges implied by transitivity from x.

Therefore, the algorithm is as follows. Remove from the priority queue the vertex y_i with the smallest value. Let z be as above and suppose z's column number is k. Add an edge from x to z. Next examine the current smallest value on the queue. Suppose it is associated with the vertex y_j. If the value associated with y_j is also k, remove y_j from the priority queue and eliminate from further consideration whichever of y_i and y_j has the smaller row number. Repeat this step until the current smallest value in the queue is greater than k. The vertex with the largest row number among those with priority value k is then added back to the priority queue with associated value equal to the column index of the next 1 after column k in its row. Repeat this entire process until the priority queue is empty.

There are $\mathrm{O}(n)$ operations which correspond to adding an edge from x, and each priority queue operation which does not add an edge from x must delete one vertex from the priority queue. Therefore, the number of priority queue operations is $\mathrm{O}(n)$. If the priority queue is a heap, each operation takes $\mathrm{O}(\log n)$ time, and the total time spent is $\mathrm{O}(n \log n)$. However, each value in the queue is an integer in the range $1, \ldots, n$, which allows us to use van Emde Boas's data structure [33] with cost $\mathrm{O}(\log\log n)$ per operation. Therefore, all edges can be added in $\mathrm{O}(n \log\log n)$ time. $\square$

A similar procedure can be used to add a new vertex y with edges into it from vertices of G, and to add all edges implied by transitivity in $\mathrm{O}(n\log\log n)$ time.

THEOREM 8. *If the underlying undirected graph of the transitive closure of a directed acyclic graph G is a weakly triangulated comparability graph, then the transitive closure of G can be found in* $\mathrm{O}(n^2 \log\log n)$ *time.*

Proof. Let T be a topological sort of G. Divide the vertices of G into two sets S_1, S_2, where S_1 consists of the first $\frac{n}{2}$ vertices in T and S_2 is the remainder. Recursively find the transitive closures C_1, C_2 of the subgraphs induced by S_1 and S_2. Construct the Γ-free matrix of the bipartite transformation of C_1 and C_2, and order the adjacency lists of each vertex of C_1 and C_2 so that vertices appear in increasing order of their columns in the Γ-free ordering.

For each vertex x in S_1, add all edges implied by transitivity of the form $x \to y_1 \to y_2$, where y_1 and y_2 are in S_2. By Lemma 7, this step takes $\mathrm{O}(n^2\log\log n)$ time overall. For each vertex y in S_2, add all edges implied by transitivity of the form $x_1 \to x_2 \to y$, x_1, $x_2 \in S_1$; this also takes $\mathrm{O}(n^2\log\log n)$ time.

At this point, we claim that we have the transitive closure of G. Consider any path from x to y in G. If x and y are both in S_1 or both in S_2, then the path must be entirely within S_1 or within S_2 as we divided vertices along a topological sort of G, and therefore, the edge from x to y will be added in the recursive step. Consider a path of G from x in S_1 to y in S_2. Let x_2 be the last vertex of S_1 on the path, and let y_2 be the first vertex of S_2 on the path. There must be an edge from x to x_2 in C_1, and an edge from y_2 to y in C_2. When we are adding edges implied by transitivity from x_2, we will add an edge from x_2 to y. When we are adding edges implied by transitivity into y, since the edge from x_2 to y has already been added, we will add an edge from x to y.

The running time of this algorithm is governed by the recurrence relation $\mathrm{T}(n) = 2\mathrm{T}(\frac{n}{2}) + \mathrm{O}(n^2\log\log n)$. $\mathrm{T}(n)$ can be shown to be $\mathrm{O}(n^2\log\log n)$; one automatic method for proving this can be found in [8]. $\square$

The transitive reduction of a directed acyclic graph G is formed by removing from G all edges $x \to z$ such that there exists a path of length greater than 1 from x to z in the transitive closure of G. The relationship between computing transitive closure and computing transitive reduction is discussed in [1]. We show that Lemma 7 can be used to find the transitive reduction of G in $\mathrm{O}(n^2\log\log n)$ time in the case that the underlying undirected graph of G's transitive closure is a weakly triangulated comparabilty graph.

THEOREM 9. *If the underlying undirected graph of the transitive closure of a directed acyclic graph G is a weakly triangulated comparability graph, then the transitive reduction of G can be found in* $\mathrm{O}(n^2\log\log n)$ *time.*

Proof. We first find the transitive closure of G and find a Γ-free ordering of the bipartite transformation of the transitive closure. Consider any vertex x. Let $y_1, y_2, \ldots, y_k$ be the vertices which have edges in from x in G. Consider adding a new vertex z which has edges to these same vertices. By Lemma 7, we can find all edges implied by transitivity from z in $\mathrm{O}(n\log\log n)$ time. An edge $x \rightarrow w$ of the transitive closure is implied by transitivity and therefore is not in the transitive reduction of G if and only if the edge $z \rightarrow w$ is found by the procedure of Lemma 7. Therefore, repeating the procedure for each vertex x in G, the transitive reduction of G can be found in $\mathrm{O}(n^2\log\log n)$ time. ∎

If the input graph is an arbitrary directed acyclic graph, Corollary 3 and Theorem 9 give us an $\mathrm{O}(n^2\log\log n)$-time algorithm which either finds the transitive reduction or declares that the underlying undirected graph of transitive closure of the input is not a weakly triangulated comparability graph. While computing the transitive closure of the input graph, we simply verify at each recursive step that the underlying undirected graph of the transitive closure of each subgraph is a weakly triangulated comparability graph.

4. Conclusions and open problems.

We have shown that weakly triangulated comparability graphs can be dealt with more efficiently than either of the classes weakly triangulated or comparability individually, by transforming problems on these graphs to an associated chordal bipartite graph. This technique allows us to recognize and find maximum independent set in $\mathrm{O}(n^2)$ time for graphs in the class and to find transitive closure and reductions in $\mathrm{O}(n^2\log\log n)$ time if the transitive closure is in the class.

It would be interesting to find a geometric intersection model for the class of graphs or its complement. Extensions of the models for the subclasses of permutation graphs and interval graphs are possible, but the fact that there are $2^{\Theta(n\log^2 n)}$ weakly triangulated comparability graphs means that the models would have to be made considerably more complex.

Weakly triangulated cocomparability graphs are natural extensions of interval graphs, using the fact that a graph G is an interval graph if and only if G is a chordal cocomparability graph. Interval graphs are also equivalent to chordal asteroidal triple-free graphs [20], and cocomparability graphs are a proper subset of asteroidal triple-free graphs [9]. Thus, it also seems natural to consider weakly triangulated asteroidal triple-free graphs to see whether problems remain tractable on this larger class.

REFERENCES

[1] A. Aho, M. Garey, and J. Ullman, *The transitive reduction of a directed graph*, SIAM J. Comput., 1 (1972), pp. 131–137.

[2] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, *Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$*, Inform. Process. Lett., 37 (1991), pp. 237–240.

[3] S. Arikati and C. Rangan, *An efficient algorithm for finding a two-pair, and its applications*, Discrete Appl. Math., 31 (1991), pp. 71–74.

[4] S. Benzer, *On the topology of the genetic fine structure*, Proc. Nat. Acad. Sci. U.S.A., 45 (1959), pp. 1607–1620.

[5] K. S. Booth and G. S. Leuker, *Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[6] M.-S. Chang, *Algorithms for maximum matching and mininum fill-in on chordal bipartite graphs*, in Algorithms and Computation, Lecture Notes in Comput. Sci. 1178, Springer-Verlag, Berlin, 1996, pp. 146–155.

[7] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, in Proceedings of the 19th Annual Symposium on the Theory of Computation, 1987, pp. 1–6.

[8] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1989.

[9] D. Corneil, S. Olariu, and L. Stewart, *Asteroidal triple-free graphs*, in Graph-Theoretic Concepts in Computer Science, Lecture Notes in Comput. Science 790, Springer-Verlag, Berlin, 1994, pp. 211–224.

[10] E. Eschen and J. Spinrad, *An $O(n^2)$ algorithm for circular-arc graph recognition*, in Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 128–137.

[11] T. Feder and R. Motwani, *Clique partitions, graph compression and speeding up algorithms*, Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 123–133.

[12] M. J. Fischer and A. R. Meyer, *Boolean matrix multiplication and transitive closure*, in Conference Record, Twelfth Annual Symposium on Switching and Automata Theory, East Lansing, MI, 1971, pp. 129–131.

[13] L. Ford and D. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[14] M. E. Furman, *Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph*, Dokl. Akad. Nauk SSSR, 11 (1970), p. 1252.

[15] T. Gallai, *Transitiv orientbare graphen*, Acta Math. Acad. Sci. Hung. Tom., 18 (1967), pp. 25–66.

[16] P. C. Gilmore and A. J. Hoffman, *A characterization of comparability graphs and interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.

[17] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[18] R. Hayward, *Weakly Triangulated Graphs*, J. Combin. Theory Ser. B, 39 (1985), pp. 200–209.

[19] R. B. Hayward, C. T. Hoàng, and F. Maffray, *Optimizing weakly triangulated graphs*, Graphs Combin., 5 (1989), pp. 339–349; erratum in 6 (1990), pp. 33–35.

[20] C. G. Lekkerkerker and J. Boland, *Representation of a finite graph by a set of intervals on the line*, Fund. Math., 51, pp. 45–64.

[21] A. Lubiw, *Doubly lexical orderings of matrices*, SIAM J. Comput., 16 (1987), pp. 854–879.

[22] T.-H. Ma and J. Spinrad, *Transitive closure for restricted classes of partial orders*, Order, 8 (1991), pp. 175–183.

[23] T.-H. Ma and J. Spinrad, *Cycle-free partial orders and chordal comparability graphs*, Order, 8 (1991), pp. 49–61.

[24] T.-H. Ma and J. Spinrad, *on the 2-chain subgraph cover and related problems*, J. Algorithms, (1994), pp. 251–268.

[25] R. McConnell and J. Spinrad, *Linear-time transitive orientation*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1997, pp. 19–25.

[26] I. Munro, *Efficient Determination of the Strongly Connected Components and the Transitive Closure of a Graph*, unpublished manuscript, University of Toronto, 1971.

[27] R. Paige and R. E. Tarjan, *Three partition refinement algorithms*, SIAM J. Comput., 16 (1987), pp. 973–989.

[28] D. J. Rose, R. E. Tarjan, and G. S. Leuker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[29] J. Spinrad, *Nonredundant ones and chordal bipartite graphs*, SIAM J. Discrete Math., 8 (1995), pp. 251–257.

[30] J. Spinrad and R. Sritharan, *Algorithms for weakly triangulated graphs*, Discrete Appl. Math., 19 (1995), pp. 181–191.

[31] J. P. Spinrad, *On comparability and permutation graphs*, SIAM J. Comput., 14 (1987), pp. 658–670.

[32] J. P. Spinrad, *Doubly lexical ordering of dense 0-1 matrices*, Inform. Process. Lett., 45 (1993), pp. 229–235.

[33] P. Van Emde Boas, *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Process. Lett., 6 (1977), pp. 80–82.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 387–400

ON APPROXIMATELY COUNTING COLORINGS OF SMALL DEGREE GRAPHS [*]

RUSS BUBLEY[†], MARTIN DYER[†], CATHERINE GREENHILL[†], AND MARK JERRUM[‡]

Abstract. We consider approximate counting of colorings of an n-vertex graph using rapidly mixing Markov chains. It has been shown by Jerrum and by Salas and Sokal that a simple random walk on graph colorings would mix rapidly, provided the number of colors k exceeded the maximum degree Δ of the graph by a factor of at least 2. We prove that this is not a necessary condition for rapid mixing by considering the simplest case of 5-coloring graphs of maximum degree 3. Our proof involves a computer-assisted proof technique to establish rapid mixing of a new "heat bath" Markov chain on colorings using the method of path coupling. We outline an extension to 7-colorings of triangle-free 4-regular graphs. Since rapid mixing implies approximate counting in polynomial time, we show in contrast that exact counting is unlikely to be possible (in polynomial time). We give a general proof that the problem of exactly counting the number of proper k-colorings of graphs with maximum degree Δ is $\#P$-complete whenever $k \geq 3$ and $\Delta \geq 3$.

Key words. graph colorings, rapidly mixing Markov chains, approximate counting, transportation problems

AMS subject classifications. 60J10, 05C15, 11Y16, 90C08

PII. S0097539798338175

1. Introduction.

The problem of properly coloring the vertices of an n-vertex graph with some given number of colors k has been widely studied [19]. Of equal theoretical interest has been the problem of *counting the number* of proper k-colorings of a graph, the values of the so-called *chromatic polynomial* [19, p. 247]. Unfortunately this is a $\#P$-hard counting problem, even for graphs with a fixed bound Δ on the vertex degrees. (See section 6 for a proof of this fact in the context of this paper.) Consequently, exact counting is unlikely to be possible, and attention turns to *approximate* counting, to some given proportional error ε. It is well known [15] that *randomized* approximate counting in this sense is equivalent to *approximate uniform generation* of a coloring; it is this problem we address here, using the *Monte Carlo Markov chain* (MCMC) method [14]. The use of the MCMC method originated in statistical physics (see, for example, [5, 9, 17]), but rigorous analysis of the method has been a more recent development, with notable contributions from the computer science community [6, 13]. This analysis involves using one of a few available methods to prove that a Markov chain converges quickly to its stationary distribution.

In this context, Jerrum [12] and Salas and Sokal [18] independently proved that a simple random walk on the k-colorings of an n-vertex graph would mix rapidly, provided the number of colors k exceeded the maximum degree Δ of the graph by a factor of more than 2. These proofs were based on entirely different techniques, *coupling* and *Dobrushin uniqueness*, respectively. The two results had different merits.

[*]Received by the editors May 4, 1998; accepted for publication (in revised form) October 30, 1998; published electronically October 5, 1999. The research of this paper was supported by the ESPRIT working group RAND2. An earlier version of this paper appeared in the 9th Annual Symposium on Discrete Mathematics, 1998.

http://www.siam.org/journals/sicomp/29-2/33817.html

[†]School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK (russ@scs.leeds.ac.uk, dyer@scs.leeds.ac.uk, csg@scs.leeds.ac.uk). The work of the first author was supported in part by an ESPRC Studentship.

[‡]Department of Computer Science, University of Edinburgh, James Clerk Maxwell Building, King's Buildings, Edinburgh EH9 3JZ, UK (mrj@dcs.ed.ac.uk).

Jerrum's result was subsequently extended (but with an $\Omega^*(n^2)$[1] increase in running time) to the case $k = 2\Delta$, whereas Salas and Sokal's result was extended to the closely related *Potts model* [3]. However, the similarity of these bounds led to a natural question: whether the condition $k \geq 2\Delta$ is necessary for rapid mixing of the underlying Markov chain on colorings. (Note that the chain is known to converge eventually for $k \geq \Delta + 2$.) There is still no general refutation of this question. Here we examine only specific cases.

In a recent paper, Bubley and Dyer [3], using a new technique called *path coupling*, showed how a coupling proof could be extended to the Potts model (and beyond). Subsequently, Dyer and Greenhill [8] used this technique to analyze a more rapidly mixing chain on colorings. This reduces the running time for the case $k = 2\Delta$ by an $\Omega^*(n^2)$ factor, but still does not demonstrate rapid mixing with fewer than 2Δ colors.

Here we show that it is possible to have rapid mixing with fewer than 2Δ colors by considering the simplest case of 5-coloring graphs of maximum degree 3. To establish our result we analyze a new Markov chain on colorings, from the so-called *heat bath* family, using the technique of path coupling. The proof idea is somewhat novel in this area: we establish the existence of certain required couplings by solving a large number of large linear programs (in fact, *transportation problems*). The analysis therefore cannot be carried out "by hand" and requires the use of linear programming software to solve the many subproblems. Thus our analysis is a "computer proof" of a mathematical theorem (in the spirit of [2, 4]).

We also extend our results to 7-colorings of *triangle-free* 4-regular graphs, again demonstrating rapid mixing with fewer than 2Δ colors. (Observe that this includes the physically relevant case of planar grids.) However, our methods are subject to combinatorial explosion in terms of Δ, and we have currently not succeeded in pushing them further.

Using results presented in [7] it is possible to deduce that the simple Jerrum/Salas–Sokal Markov chain of colorings is rapidly mixing for the same values of k and the same families of graphs as our new chains; that is, when $k = 5$ and the graphs have maximum degree 3 or when $k = 7$ and the graphs are triangle-free and 4-regular. (Note, however, that the mixing rate which is established in this way, while still polynomial, is much larger than the mixing rate of our chains.)

The plan of the paper is as follows. In section 1.1 of the paper we review the path coupling method, and in section 1.2 we briefly review recent work on approximately counting colorings. In section 2 we give a rigorous mathematical definition of a *heat bath Markov chain.* (As far as we are aware, this is the first general definition of this concept.) In section 3 we show that the analysis of a heat bath Markov chain can be reduced (using path coupling) to the task of solving a set of related transportation problems. This leads to a computational method for analyzing heat bath Markov chains. In section 4 this approach is applied to a Markov chain on 5-colorings of graphs of maximum degree 3. A table needed for the computation is provided. In section 5, we outline the extension to 7-colorings of triangle-free 4-regular graphs. Finally, in section 6 we prove that the problem of exactly counting the number of proper k-colorings of graphs with maximum degree Δ is $\#P$-complete when $k \geq 3$ and $\Delta \geq 3$.

1.1. Path coupling. Let Ω be a finite set and let $\mathcal{M}$ be a Markov chain with state space Ω, transition matrix P, and unique stationary distribution π. If the initial

[1]$\Omega^*(\cdot)$ is the notation which hides factors of $\log n$.

state of the Markov chain is x, then the distribution of the chain at time t is given by $P_x^t(y) = P^t(x,y)$. The *total variation distance* of the Markov chain from π at time t with initial state x is defined by

$$\mathrm{d_{TV}}(P_x^t, \pi) = \frac{1}{2}\sum_{y\in\Omega} |P^t(x,y) - \pi(y)|.$$

A Markov chain is only useful in terms of almost uniform sampling or approximate counting if its total variation distance tends to zero relatively quickly from some (easily obtainable) initial state. Let $\tau_x(\varepsilon)$ denote the least value T such that $\mathrm{d_{TV}}(P_x^t, \pi) \leq \varepsilon$ for all $t \geq T$. The *mixing rate*[2] of $\mathcal{M}$, denoted by $\tau(\varepsilon)$, is defined by $\tau(\varepsilon) = \max\{\tau_x(\varepsilon) : x \in \Omega\}$. A Markov chain is said to be *rapidly mixing* if the mixing rate is bounded above by some polynomial in n and $\log(\varepsilon^{-1})$, where n is a measure of the size of the elements of Ω. All logarithms are to the base e.

There are relatively few methods available to prove that a Markov chain is rapidly mixing. One such method is *coupling*. A *coupling* for $\mathcal{M}$ is a stochastic process (X_t, Y_t) on $\Omega \times \Omega$ such that each of (X_t), (Y_t), considered independently, is a faithful copy of $\mathcal{M}$. The coupling lemma (see, for example, Aldous [1]) states that the total variation distance of $\mathcal{M}$ at time t is bounded above by $\mathrm{Prob}[X_t \neq Y_t]$, the probability that the process has not coupled. The difficulty in applying this result lies in obtaining an upper bound for this probability. In the *path coupling* method, introduced by Bubley and Dyer [3], one need only define and analyze a coupling on a subset S of $\Omega \times \Omega$. Choosing the set S carefully can simplify considerably the arguments involved in proving rapid mixing of Markov chains by coupling. The path coupling method is described in the next lemma (taken from [8]). We use the term *path* to refer to a sequence of elements of Ω, which need not be a path of possible transitions in the Markov chain.

LEMMA 1.1. *Let δ be an integer-valued metric defined on $\Omega \times \Omega$ which takes values in $\{0, \ldots, D\}$. Let S be a subset of $\Omega \times \Omega$ such that for all $(X_t, Y_t) \in \Omega \times \Omega$ there exists a path*

$$X_t = Z_0, Z_1, \ldots, Z_r = Y_t$$

between X_t and Y_t where $(Z_l, Z_{l+1}) \in S$ for $0 \leq l < r$ and $\sum_{l=0}^{r-1} \delta(Z_l, Z_{l+1}) = \delta(X_t, Y_t)$. Define a coupling $(X, Y) \mapsto (X', Y')$ of the Markov chain $\mathcal{M}$ on all pairs $(X, Y) \in S$. Suppose that there exists $\beta < 1$ such that

$$\mathbf{E}\left[\delta(X', Y')\right] \leq \beta\, \delta(X, Y)$$

for all $(X, Y) \in S$. Then the mixing rate $\tau(\varepsilon)$ of $\mathcal{M}$ satisfies

$$\tau(\varepsilon) \leq \frac{\log(D\varepsilon^{-1})}{1-\beta}.$$

Remark 1. The set S is often taken to be

$$S = \{(X, Y) \in \Omega \times \Omega : \delta(X, Y) = 1\}.$$

Here a coupling only need be defined for pairs at distance 1 apart.

Remark 2. Notice that Lemma 1.1 does not assume that the Markov chain $\mathcal{M}$ is reversible.

[2]Elsewhere, the *mixing rate* is sometimes defined to be $\tau_x(\varepsilon)$ for some fixed x.

1.2. Colorings of graphs. Let $G = (V, E)$ be a graph on n vertices and let Δ be the maximum degree of G. Let k be a positive integer and let C be a set of size k. A map from V to C is called a *k-coloring*. A vertex v is said to be *properly colored* in the coloring X if v is colored differently from all of its neighbors. A coloring X is called *proper* if every vertex is properly colored in X. A necessary and sufficient condition for the existence of a proper k-coloring of all graphs with maximum degree Δ is $k \geq \Delta + 1$. Denote by $\Omega_k(G)$ the set of all proper k-colorings of G. The color assigned to a vertex v in the coloring X is denoted by $X(v)$.

Jerrum [12] and Salas and Sokal [18] independently defined a Markov chain with state space $\Omega_k(G)$ which is irreducible for $k \geq \Delta + 2$ and rapidly mixing for $k \geq 2\Delta$. One version of this chain has the following transition procedure: from current state X, choose a vertex v uniformly at random and choose a color c uniformly at random from the set of those colors which properly color v in X. Then recolor v with c to give the new state. A new Markov chain of colorings, denoted by $\mathcal{M}_1(\Omega_k(G))$, was introduced in [8]. This new chain is irreducible for $k \geq \Delta+1$ and is also rapidly mixing for $k \geq 2\Delta$. Moreover, the new chain is provably faster than the Jerrum chain when G is Δ-regular or (after a slight adjustment to the chain) when $2\Delta \leq k \leq 3\Delta - 1$. When $k = 2\Delta$ the new chain is $\Omega^*(n^2)$ times faster than the Jerrum chain (see [8]). The transitions of $\mathcal{M}_1(\Omega_k(G))$ are defined by the following procedure: choose an edge $\{v, w\}$ uniformly at random from E and choose an ordered pair of colors $(c(v), c(w))$ uniformly at random from the set of all those such that both v and w are properly colored when v is recolored $c(v)$ and w is recolored $c(w)$. The resulting coloring is the new state.

2. Definition of a heat bath Markov chain. The notation is adapted from that used in [3]. Let V and C be finite sets and let Ω be a subset of C^V, the set of functions from V to C. Let L be a subset of the power set of V. We shall refer to the elements of L as *lines*. For $X \in \Omega$, $\ell \in L$, and $c \in C^\ell$, let $X_{\ell\to c}$ denote the element of C^V defined by

$$X_{\ell\to c}(u) = \begin{cases} c(u) & \text{if } u \in \ell, \\ X(u) & \text{otherwise.} \end{cases}$$

If $X_{\ell\to c} \in \Omega$ then we say that c is *acceptable* at ℓ in X. Finally let $\mathcal{S}_X(\ell)$ be the set of all elements of C^ℓ which are acceptable at ℓ in X. Let π be a distribution on Ω and, for all $\ell \in L$ and all $X \in \Omega$, let $\pi^*_X(\ell)$ be defined by

$$\pi^*_X(\ell) = \sum_{c\in\mathcal{S}_X(\ell)} \pi(X_{\ell\to c}).$$

The set of lines L and the distribution π can be used to define a Markov chain $\mathcal{M}(L, \pi)$ with state space Ω. The transition procedure from the current state X is as follows:

1. choose ℓ uniformly at random from L,
2. choose $c \in \mathcal{S}_X(\ell)$ with probability $\pi(X_{\ell\to c})/\pi^*_X(\ell)$ and move to $X_{\ell\to c}$.

Clearly the Markov chain $\mathcal{M}(L, \pi)$ is aperiodic.

The definition of a heat bath Markov chain can now be stated.

DEFINITION 2.1. *A Markov chain $\mathcal{M}$ with state space $\Omega \subseteq C^V$ is said to be a heat bath Markov chain if there exists a set of lines L and a distribution π on Ω such that $\mathcal{M} = \mathcal{M}(L, \pi)$.*

Suppose that $\mathcal{M}(L, \pi)$ satisfies Definition 2.1. The transition matrix P of $\mathcal{M}(L, \pi)$ has entries

$$P(X, Y) = \pi(Y) \sum_{\ell\supseteq D} \left(|L||\pi^*_X(\ell)|\right)^{-1},$$

where $D = \{v \in V : X(v) \neq Y(v)\}$. If $\ell \supseteq D$, then $\mathcal{S}_X(\ell) = \mathcal{S}_Y(\ell)$ and $\pi^*_X(\ell) = \pi^*_Y(\ell)$. Therefore P is reversible with respect to π. If $\mathcal{M}(L, \pi)$ is also irreducible, then it is an ergodic chain with stationary distribution π. Say that the set of lines L is *sufficient* if $\mathcal{M}(L, \pi)$ is irreducible.

In the special case that π is the uniform distribution on Ω we will write $\mathcal{M}(L)$ instead of $\mathcal{M}(L, \pi)$. Here the transition procedure of $\mathcal{M}(L)$ involves choosing a line $\ell \in L$ uniformly at random and a color map $c \in \mathcal{S}_X(\ell)$ uniformly at random and making a transition to $X_{\ell \to c}$. All the heat bath Markov chains considered in this paper involve the uniform distribution. Two examples are the two Markov chains of k-colorings of a graph described in section 1.2. Both fit Definition 2.1, where π is the uniform distribution on $\Omega_k(G)$. For the Jerrum/Salas–Sokal chain the set L of lines is merely the set of singleton vertices. In the case of the chain $\mathcal{M}_1(\Omega_k(G))$ the set of lines L is the edge set E. In both cases, the set of lines L is sufficient, so the corresponding Markov chain is ergodic.

Let $H(X, Y)$ denote the *Hamming distance* between X and Y, defined by

$$H(X, Y) = |\{v \in V : X(v) \neq Y(v)\}|.$$

Suppose we want to analyze the mixing rate of $\mathcal{M}(L)$ using path coupling on pairs at Hamming distance 1 apart. Let X and Y be two elements of Ω which differ only at $v \in V$. A coupling $(X, Y) \mapsto (X', Y')$ must be defined for every $\ell \in L$. Call a coupling at ℓ *optimal* if it minimizes the expected value of $H(X', Y')$. Note that the *optimal* coupling defined here is not the same as the *maximal* coupling which is referred to elsewhere in the literature. Refer to $\mathbf{E}\,[H(X', Y') - 1]$ as the *cost* of the coupling. If $\mathcal{S}_X(\ell) = \mathcal{S}_Y(\ell)$, then an optimal coupling at ℓ is obtained by choosing the same element $c \in \mathcal{S}_X(\ell)$ in both X and Y. Here $H(X', Y') = 0$ if v lies on the line ℓ and $H(X', Y') = 1$ otherwise. Suppose now that $\mathcal{S}_X(\ell) \neq \mathcal{S}_Y(\ell)$; it follows that $v \notin \ell$. Given $c_X \in \mathcal{S}_X(\ell)$ and $c_Y \in \mathcal{S}_Y(\ell)$, let $P(c_X, c_Y)$ denote the probability that the pair (c_X, c_Y) is chosen by the coupling and let

$$h(c_X, c_Y) = |\{u \in \ell : c_X(u) \neq c_Y(u)\}|,$$

the number of elements of ℓ which are assigned different values by c_X and c_Y. Then the expected value of $H(X', Y') - 1$ is given by

$$\sum P(c_X, c_Y) h(c_X, c_Y),$$

where the sum is over all $(c_X, c_Y) \in \mathcal{S}_X(\ell) \times \mathcal{S}_Y(\ell)$.

3. The related transportation problems. In this section it will be shown that an optimal coupling for a heat bath Markov chain is equivalent to the optimal solutions of a related set of *transportation problems.* (For more information on transportation problems see, for example, [16].) Let m and n be positive integers and let K be an $m \times n$ matrix of nonnegative integers (the *cost matrix*). Let a be a vector of m positive integers and let b be a vector of n positive integers such that $\sum_{i=1}^m a_i = \sum_{j=1}^n b_j = N$. An $m \times n$ matrix Z of nonnegative numbers is a solution of the transportation problem defined by a, b, and K if $\sum_{j=1}^n Z_{i,j} = a_i$ for $1 \leq i \leq m$ and $\sum_{i=1}^m Z_{i,j} = b_j$ for $1 \leq j \leq n$. The *cost* of this solution is measured by $\sum_{i=1}^m \sum_{j=1}^n K_{i,j} Z_{i,j}$. The elements of a are called *row sums* and the elements of b are called *column sums.* An *optimal solution* of this transportation problem is a solution which minimizes the cost. Every entry of an optimal solution is a nonnegative integer. Efficient algorithms exist for solving transportation problems.

Returning to the problem of defining an optimal coupling for the heat bath Markov chain $\mathcal{M}(L)$ at the line ℓ, let $N_X = |\mathcal{S}_X(\ell)|$ and $N_Y = |\mathcal{S}_Y(\ell)|$. Let a be the N_X-dimensional vector with each entry equal to N_Y and let b be the N_Y-dimensional vector with each entry equal to N_X. Let K be the matrix with N_X rows and N_Y columns, corresponding to the elements of $\mathcal{S}_X(\ell)$ and $\mathcal{S}_Y(\ell)$, respectively, such that the (c_X, c_Y)th entry of K is $h(c_X, c_Y)$. A coupling at ℓ defines an $N_X \times N_Y$ matrix Z with (c_X, c_Y)th entry given by

$$N_X N_Y \operatorname{Prob}\left[X' = X_{\ell \to c_X}, Y' = Y_{\ell \to c_Y}\right].$$

The matrix Z is a solution of the transportation problem defined by a, b, and K. An optimal solution of this transportation problem corresponds to a optimal coupling at ℓ. The cost of an optimal solution equals $N_X N_Y$ times the cost of an optimal coupling. Therefore one can attempt to prove that $\mathcal{M}(L)$ is rapidly mixing by adapting the path coupling method as follows. Here “nonisomorphic” means “nonisomorphic in some appropriate sense.”

1. Compile a complete list of nonisomorphic pairs (X, Y) such that $H(X, Y) = 1$.
2. For each such pair, calculate the contribution of all lines ℓ such that $\mathcal{S}_X(\ell) = \mathcal{S}_Y(\ell)$.
3. For all other lines ℓ, solve the corresponding transportation problem to find the cost of the optimal coupling.
4. Combine all this information to determine whether the expected value of $H(X', Y') - 1$ is negative in all cases. If this is the case, then let the maximum of these values be $\beta - 1$. The mixing time $\tau(\varepsilon)$ of $\mathcal{M}(L)$ satisfies
$$\tau(\varepsilon) \leq \frac{\log(n\varepsilon^{-1})}{1 - \beta}$$
by Lemma 1.1. If $1 - \beta$ is only polynomially small, then $\mathcal{M}(L)$ is rapidly mixing.

Remark 3. In order to prove that the Markov chain is rapidly mixing it may not be necessary to find an optimal solution in each case. Instead, it may be sufficient to find a solution with a low enough cost in each case.

Remark 4. Certainly the list of pairs (X, Y) used in the above procedure must contain at least one element from each isomorphism class so that the calculations are conclusive. If the list is a transversal, then no unnecessary calculations are performed. In many cases, however, the amount of effort required to find a transversal of the isomorphism classes is prohibitive and ruling out obviously isomorphic pairs will suffice. Moreover, in most cases one need only consider a certain “neighborhood” around the line ℓ rather than the entire maps X, Y.

4. Five-coloring degree-three graphs.

Let G be a graph with maximum degree 3 and let $\Omega_k(G)$ denote the set of proper k-colorings of G. The two known Markov chains on k-colorings described in section 1.2 are rapidly mixing for $k \geq 6$. In this section a Markov chain on $\Omega_5(G)$ is defined and a computational proof that the chain is rapidly mixing is given following the method described in section 3.

Let $C = \{1, \ldots, 5\}$ be the color set and let $\mathcal{M}$ be the following Markov chain on $\Omega_5(G)$. If w is a vertex with degree d then let the neighbors of w be denoted by $u_1(w), \ldots, u_d(w)$. Denote by $\tilde{w}$ the set defined by

$$\tilde{w} = \{w, u_1(w), \ldots, u_d(w)\}.$$

Define the set of lines L by

$$L = \{\tilde{w} : w \in V\}.$$

Then $\mathcal{S}_X(\tilde{w})$ is

$$\mathcal{S}_X(\tilde{w}) = \left\{c \in C^{\tilde{w}} : X_{\tilde{w}\to c} \in \Omega_5(G)\right\}. \tag{4.1}$$

The Markov chain $\mathcal{M} = \mathcal{M}(L)$ is a heat bath Markov chain in the sense of Definition 2.1, where π is the uniform distribution on $\Omega_5(G)$. The transitions of $\mathcal{M}$ from current state X follow the pattern described in section 2: a line $\tilde{w} \in L$ is chosen uniformly at random, a corresponding color mapping $c \in \mathcal{S}_X(\tilde{w})$ is chosen uniformly at random, and a transition is made to $X_{\tilde{w}\to c}$. The chain $\mathcal{M}$ is irreducible; for example, it can perform all the moves of the Jerrum/Salas–Sokal chain. Therefore the set of lines L is sufficient and the chain $\mathcal{M}$ is ergodic with uniform stationary distribution.

The mixing rate of $\mathcal{M}$ will now be analyzed using path coupling on pairs at Hamming distance 1 apart. Let $X, Y \in \Omega_5(G)$ and let $d = H(X, Y)$. It is not always possible to form a path

$$X = Z_0, Z_1, \ldots, Z_d = Y$$

of length d between $X, Y \in \Omega_5(G)$ where $H(Z_l, Z_{l+1}) = 1$ for $0 \le l < d$ and $Z_l \in \Omega_5(G)$ for $0 \le l \le d$; however, we can always form such a path where $Z_l \in C^V$ for $0 \le l \le d$. If $X \in \Omega_5(G)$, then the definition of the set $\mathcal{S}_X(\tilde{w})$ given in (4.1) is equivalent to the following definition:

$$\mathcal{S}_X(\tilde{w}) = \left\{c \in C^{\tilde{w}} : \text{ all elements of } \tilde{w} \text{ are properly colored in } X_{\tilde{w}\to c}\right\}.$$

The latter definition makes sense for all $X \in C^V$. Using the latter definition, we can extend the chain $\mathcal{M}$ to act on the state space C^V. It is easy to see that the extended chain has the stationary distribution

$$\pi(X) = \begin{cases} |\Omega_5(G)|^{-1} & \text{if } X \in \Omega_5(G), \\ 0 & \text{otherwise.} \end{cases}$$

The extended chain is no longer reversible; however, by Remark 2 we may apply Lemma 1.1 to the extended chain. If the extended chain is rapidly mixing, then the original chain is also rapidly mixing with the same upper bound on the mixing rate. This follows since mixing time is defined as the maximum over all initial states, and the original chain involves only a subset of these.

Now suppose that X and Y are two colorings which differ only at v. Without loss of generality, suppose that $X(v) = 1$ and $Y(v) = 2$. Denote the degree of v by d_v. Let w be a vertex for which $\tilde{w}$ corresponds to the line chosen in the transition procedure. If $w = v$ or $\{w, v\} \in E$, then $\mathcal{S}_X(\tilde{w}) = \mathcal{S}_Y(\tilde{w})$. Therefore the same choice of c may be made in X and Y. It follows that these vertices contribute $-(d_v+1)/n$ to the expected value of $H(X', Y') - 1$. The only other lines which may make nonzero contributions to the expected value of $H(X', Y') - 1$ are those corresponding to vertices w which satisfy the following criteria:

1. $w \ne v$ and $\{w, v\} \notin E$,
2. there exists $u \in V$ such that $\{w, u\} \in E$ and $\{u, v\} \in E$.

Call such a vertex a *critical* vertex. Define the *multiplicity* $\mu(w)$ of a critical vertex w to be the number of neighbors of w which are neighbors of v. Now w may have up to three neighbors. If w has degree 2 or 3, then label its second neighbor by z_1; if w has degree 3, then label its third neighbor by z_2. If z_i is present, then it may have up to three neighbors, for $i = 1, 2$. Let the second and third neighbors of z_i be labeled $z_{i,1}$, $z_{i,2}$, if they are present, for $i = 1, 2$. Finally u may have a third neighbor

Table 4.1
The 15 configurations.

i	d_w	edges	$\mu_i(w)$	λ_i	$M_i(w)$
1	1	NNNN	1	3	5/12
2	2	NNNN	1	20	13/28
3	2	NNNY	1	3	73/156
4	2	YNNN	2	8	25/29
5	2	YNNY	2	1	2/3
6	3	NNNN	1	112	381/748
7	3	YNNN	2	59	2230/2343
8	3	YYNN	3	17	94/71
9	3	NNYN	1	31	527/1081
10	3	YNYN	2	11	1888/2115
11	3	YYYN	3	3	10/9
12	3	NNNY	1	20	250/483
13	3	YNNY	2	5	67/90
14	3	NNYY	1	3	25/57
15	3	NYYY	2	1	3/4

which we denote by z_3 if it is present. There are 15 nonisomorphic configurations on these vertices, depending on the degree of w and on whether the edges

$$\{v, z_1\}, \{v, z_2\}, \{z_1, z_2\}, \{z_1, u\}$$

are present or absent in E. The details are given in Table 4.1, where the second column holds the degree of w and the third column holds a string of four characters, each equal to "Y" or "N." The characters refer to the four edges listed above—in that order—and "Y" indicates that the edge is present in E while "N" indicates that it is absent. The fourth column of Table 4.1 holds $\mu_i(w)$, the multiplicity of w in the ith configuration. The last two columns of Table 4.1 are explained below. Note that any identification or edges between the vertices $z_{1,1}, z_{1,2}, z_{2,1}, z_{2,2}, z_3$ can be ignored, as this situation can be modeled by coloring the affected vertices with the same (respectively, different) colors. The only vertices which affect the sets $\mathcal{S}_X(\tilde{w})$, $\mathcal{S}_Y(\tilde{w})$ are the vertices

$$v, z_3, z_{1,1}, z_{1,2}, z_{2,1}, z_{2,2}.$$

Call these vertices the *rim vertices* and call an assignation of colors to these vertices a *rim coloring*. Following Remark 4, it suffices to consider a complete list of nonisomorphic rim colorings for each configuration. Let λ_i be the number of nonisomorphic rim colorings of the ith configuration, $1 \leq i \leq 15$. These values are listed in the fifth column of Table 4.1, while a complete list of the nonisomorphic rim colorings for each configuration can be found in Table 4.2. Each rim coloring is shown as a list of six characters representing the colors of the six rim vertices in the order given above. If a rim vertex equals v, then its color is given as 0. Similarly, if a rim vertex equals u, z_1, or z_2, then its color is given as ρ. If a rim vertex is absent, then its color is given as ω. The first element of each rim coloring is always 0, but it is included so that any symmetry in the configuration is apparent.

Suppose that in a given configuration the vertex z_1 is present and is not joined by an edge to v, u (or z_2 if present). Then it suffices to assume that z_1 has degree 3, as follows. If z_1 has degree 2, then its third neighbor $z_{1,2}$ may be added and colored with the same color as $z_{1,1}$ without affecting the sets $\mathcal{S}_X(\tilde{w})$, $\mathcal{S}_Y(\tilde{w})$. If z_1 has degree 1, then the sets $\mathcal{S}_X(\tilde{w})$, $\mathcal{S}_Y(\tilde{w})$, respectively, contain exactly four times as many elements as those obtained in the case that vertex z_1 is not present. Let a, b, and K be the

Table 4.2
The rim colorings for configurations 1–15.

Config 1:	$01\omega\omega\omega\omega$	$03\omega\omega\omega\omega$	$0\omega\omega\omega\omega\omega$				
Config 2:	$0111\omega\omega$	$0112\omega\omega$	$0113\omega\omega$	$0122\omega\omega$	$0123\omega\omega$	$0133\omega\omega$	$0134\omega\omega$
	$0311\omega\omega$	$0312\omega\omega$	$0313\omega\omega$	$0314\omega\omega$	$0333\omega\omega$	$0334\omega\omega$	$0344\omega\omega$
	$0345\omega\omega$	$0\omega11\omega\omega$	$0\omega12\omega\omega$	$0\omega13\omega\omega$	$0\omega33\omega\omega$	$0\omega34\omega\omega$	
Config 3:	$0\rho1\rho\omega\omega$	$0\rho3\rho\omega\omega$	$0\rho\omega\rho\omega\omega$				
Config 4:	$0101\omega\omega$	$0103\omega\omega$	$0104\omega\omega$	$0303\omega\omega$	$0304\omega\omega$	$0\omega01\omega\omega$	$0\omega03\omega\omega$
	$0\omega0\omega\omega\omega$						
Config 5:	$0\rho0\rho\omega\omega$						
Config 6:	011111	011112	011113	011122	011123	011133	011134
	011212	011213	011222	011223	011233	011234	011313
	011314	011322	011323	011324	011333	011334	011344
	011345	012222	012223	012233	012234	012323	012324
	012333	012334	012344	012345	013333	013334	013344
	013345	013434	013435	031111	031112	031113	031114
	031122	031123	031124	031133	031134	031144	031145
	031212	031213	031214	031233	031234	031244	031245
	031313	031314	031323	031324	031333	031334	031344
	031345	031414	031415	031424	031425	031433	031434
	031435	031444	031445	031455	033333	033334	033344
	033345	033434	033435	033444	033445	033455	034444
	034445	034455	034545	$0\omega1111$	$0\omega1112$	$0\omega1113$	$0\omega1122$
	$0\omega1123$	$0\omega1133$	$0\omega1134$	$0\omega1212$	$0\omega1213$	$0\omega1233$	$0\omega1234$
	$0\omega1313$	$0\omega1314$	$0\omega1323$	$0\omega1324$	$0\omega1333$	$0\omega1334$	$0\omega1344$
	$0\omega1345$	$0\omega3333$	$0\omega3334$	$0\omega3344$	$0\omega3345$	$0\omega3433$	$0\omega3435$
Config 7:	010111	010112	010113	010122	010123	010133	010134
	010211	010212	010213	010233	010234	010311	010312
	010313	010314	010322	010323	010324	010333	010334
	010344	010345	030311	030312	030313	030314	030333
	030334	030344	030345	030411	030412	030413	030415
	030433	030434	030435	030455	$0\omega0111$	$0\omega0112$	$0\omega0113$
	$0\omega0122$	$0\omega0123$	$0\omega0133$	$0\omega0134$	$0\omega0311$	$0\omega0312$	$0\omega0313$
	$0\omega0314$	$0\omega0333$	$0\omega0334$	$0\omega0344$	$0\omega0345$	$0\omega0\omega11$	$0\omega0\omega12$
	$0\omega0\omega13$	$0\omega0\omega33$	$0\omega0\omega34$				
Config 8:	010101	010102	010103	010203	010303	010304	030303
	030304	030405	$0\omega0101$	$0\omega0102$	$0\omega0103$	$0\omega0303$	$0\omega0304$
	$0\omega0\omega01$	$0\omega0\omega03$	$0\omega0\omega0\omega$				
Config 9:	$011\rho1\rho$	$011\rho2\rho$	$011\rho3\rho$	$012\rho2\rho$	$012\rho3\rho$	$013\rho3\rho$	$013\rho4\rho$
	$031\rho1\rho$	$031\rho2\rho$	$031\rho3\rho$	$031\rho4\rho$	$033\rho3\rho$	$033\rho4\rho$	$034\rho4\rho$
	$034\rho5\rho$	$01\omega\rho1\rho$	$01\omega\rho2\rho$	$01\omega\rho3\rho$	$03\omega\rho1\rho$	$03\omega\rho3\rho$	$03\omega\rho4\rho$
	$01\omega\rho\omega\rho$	$03\omega\rho\omega\rho$	$0\omega1\rho1\rho$	$0\omega1\rho2\rho$	$0\omega1\rho3\rho$	$0\omega3\rho3\rho$	$0\omega3\rho4\rho$
	$0\omega\omega\rho1\rho$	$0\omega\omega\rho3\rho$	$0\omega\omega\rho\omega\rho$				
Config 10:	$010\rho1\rho$	$010\rho2\rho$	$010\rho3\rho$	$030\rho1\rho$	$030\rho3\rho$	$030\rho4\rho$	$010\rho\omega\rho$
	$030\rho\omega\rho$	$0\omega0\rho1\rho$	$0\omega0\rho3\rho$	$0\omega0\rho\omega\rho$			
Config 11:	$010\rho0\rho$	$030\rho0\rho$	$0\omega0\rho0\rho$				
Config 12:	$0\rho1\rho11$	$0\rho1\rho12$	$0\rho1\rho13$	$0\rho1\rho22$	$0\rho1\rho23$	$0\rho1\rho33$	$0\rho1\rho34$
	$0\rho3\rho11$	$0\rho3\rho12$	$0\rho3\rho13$	$0\rho3\rho14$	$0\rho3\rho33$	$0\rho3\rho34$	$0\rho3\rho44$
	$0\rho3\rho45$	$0\rho\omega\rho11$	$0\rho\omega\rho12$	$0\rho\omega\rho13$	$0\rho\omega\rho33$	$0\rho\omega\rho34$	
Config 13:	$0\rho0\rho11$	$0\rho0\rho12$	$0\rho0\rho13$	$0\rho0\rho33$	$0\rho0\rho34$		
Config 14:	$0\rho\rho\rho1\rho$	$0\rho\rho\rho3\rho$	$0\rho\rho\rho\omega\rho$				
Config 15:	$0\rho\rho\rho0\rho$						

row sums, column sums, and cost matrix obtained when z_1 is absent and let a', b', and K' be those obtained when z_1 is present. Then K' has the block form

$$K' = \begin{pmatrix} K & * & * & * \\ * & K & * & * \\ * & * & K & * \\ * & * & * & K \end{pmatrix}$$

and the vector a' (respectively, b') consists of four copies of the vector $4a$ (respectively, $4b$) concatenated together. It is not hard to see that the cost of an optimal solution of the transportation problem defined by a', b', and K' is bounded above by $16k$, where k is the cost of an optimal solution of the transportation problem defined by a, b, and K. Therefore the cost of an optimal coupling when z_1 is present is bounded above by the cost of an optimal coupling when z_1 is absent. The same argument holds with the roles of z_1 and z_2 reversed and reduces the number of rim colorings for certain configurations.

For each rim coloring of a given configuration the sets $\mathcal{S}_X(\tilde{w})$, $\mathcal{S}_Y(\tilde{w})$ can be formed. From these the matrix K of costs can be obtained and given as input to an algorithm for the transportation problem. The cost of an optimal solution corresponds to the value of $N_X N_Y \mathbf{E}\,[H(X',Y')-1]$ for an optimal coupling of X and Y at w. Let $M_i(w)$ be the greatest cost of an optimal coupling over all possible rim colorings for the ith configuration, $1 \le i \le 15$. These values are listed in the sixth column of Table 4.1. These results lead to the following theorem.

THEOREM 4.1. *The Markov chain $\mathcal{M}$ is rapidly mixing with mixing rate*

$$\tau(\varepsilon) \le \frac{161}{144} n \log(n\varepsilon^{-1}).$$

Proof. The maximum contribution of a critical vertex in the ith configuration is denoted by $M_i(w)$. These values were calculated using an algorithm for the transportation problem and are listed in Table 4.1. Denote by d_v the degree of the vertex v. If $d_v = 0$, then the expected value of $H(X',Y')-1$ is $-1/n$. If $d_v > 0$, then the expected value of $H(X',Y')-1$ in the ith configuration is given by

$$\mathbf{E}\,[H(X',Y')-1] \le -\frac{(d_v+1)}{n} + \sum_w \frac{M_i(w)}{n}$$

for $1 \le i \le 15$, where the sum is over all critical w. By inspection of Table 4.1 it is clear that

$$M_i(w) < \frac{2\mu_i(w)}{3} \le \frac{(d_v+1)\mu_i(w)}{2d_v}$$

for $1 \le i \le 15$. Therefore the expected value of $H(X',Y')-1$ is less than

$$\frac{(d_v+1)}{2d_v n}\left(\sum_w \mu_i(w) - 2d_v\right) \le 0$$

for $1 \le i \le 15$. To compute an upper bound for the mixing rate, let

$$T = \max\left\{M_i(w)/\mu_i(w) : 1 \le i \le 15\right\}.$$

By inspection of Table 4.1 one can see that $T = 250/483$, corresponding to configuration 12. Then

$$\mathbf{E}\,[H(X',Y')-1] \le -\frac{4}{n} + \frac{6T}{n} = -\frac{144}{161n}.$$

Therefore, by Lemma 1.1 the mixing rate of $\mathcal{M}$ is as stated. $\square$

Remark 5. In order to prove that $\mathcal{M}$ is rapidly mixing, it suffices to establish the inequality $T < 2/3$, as shown in the proof of Theorem 4.1. To show this, the exact value of $M_i(w)$ was calculated for $1 \le i \le 15$. As mentioned in Remark 3, much calculation can be saved by halting the algorithm for the transportation problem in each case as soon as a feasible solution with low enough cost has been found. In two separate calculations using different heuristics, the authors established the two bounds $T \leq 1174/1767$ and $T \leq 515/966$ using this method. The first gives rise to the upper bound

$$\tau(\varepsilon) \leq \frac{589}{8} n \log(n\varepsilon^{-1})$$

on the mixing rate of $\mathcal{M}$, which is more than 65 times greater than the bound given in Theorem 4.1. This illustrates that the increased efficiency which results in this way may incur a significant loss of tightness in the bound on the mixing rate. However, the second calculation gives rise to the bound

$$\tau(\varepsilon) \leq \frac{161}{129} n \log(n\varepsilon^{-1}),$$

which is very close to the bound provided by Theorem 4.1.

5. Further applications. In principle it is not difficult to extend the result of section 4 to graphs of larger (bounded) degree—the algorithm for generating all the nonisomorphic configurations readily extends to this case. However, there is a problem that prevents us from doing this: the number of rim colorings suffers from combinatorial explosion, as does the size of the transportation problems to be solved. This may be offset in some fashion by restricting attention to smaller classes of graphs. The high degree of symmetry inherent in lattice graphs makes these ideal candidates.

It would be interesting to see just how general these classes of graphs can be made before the computation becomes intractable. Determining the cut-off point is a largely subjective matter and dependent on available resources. One additional computation that we discovered to be tractable is the proof of rapid mixing of $\Omega_7(G)$, where G is a 4-regular, triangle-free graph. It should be clear where the simplifications arise from these two additional restrictions over the case considered in section 4: the regularity condition ensures that we need not consider the cases where some of the rim vertices are absent, and demanding that the graph is triangle-free means that we do not need to consider the cases where some of the rim vertices are adjacent to the critical vertex that they rim.

To further save on the amount of computational time needed, we did not actually find the optimum solutions of the transportation problems. Instead we made use of Remark 3, since it transpired that using the matrix minimum heuristic was sufficient.

We will use some of the notation of section 4 for the remainder of this section.

In order to ensure that the chain was rapidly mixing, we needed to show that we had

$$\mathbf{E}\left[H(X', Y') - 1\right] \leq -\frac{5}{n} + \sum_{w} \frac{M_i(w)}{n}$$

for all critical vertices w. (The 5 arises from the fact that there are five choices of vertex, v and all of its neighbors, whose choice would ensure that $H(X', Y') = 0$.)

It is sufficient therefore to show that for each rim coloring Q we have

$$\text{cost}(Q) \times \frac{12}{\mu(w)} - 5 \leq 0, \quad \text{i.e., } \text{cost}(Q) \leq \frac{5}{12}\mu(w), \tag{5.1}$$

where $\text{cost}(Q)$ is the cost of the optimal solution to the transportation problem associated with Q.

TABLE 5.1
The worst configurations for 7-coloring 4-regular triangle-free graphs.

Rim coloring	$\mu(w)$	Proven bound
0 3 4 5 5 6 5 5 6 5 6 7	1	62/189
0 3 4 5 5 6 5 6 6 5 6 7	1	62/189
0 3 4 0 5 6 3 3 7 4 4 7	2	950/1489
0 3 4 0 5 6 3 3 7 4 7 7	2	950/1489
0 3 4 0 5 6 3 3 7 5 5 7	2	950/1489
0 3 4 0 5 6 3 3 7 5 7 7	2	950/1489
0 3 4 0 5 6 3 7 7 4 7 7	2	950/1489
0 3 4 0 5 6 3 7 7 5 7 7	2	950/1489
0 3 4 0 3 5 0 6 7 3 4 5	3	914/969
0 3 4 0 3 5 0 6 7 3 4 6	3	914/969
0 3 4 0 3 5 0 6 7 3 6 7	3	914/969
0 3 4 0 3 4 0 3 5 0 6 7	4	1212/997
0 3 4 0 3 5 0 3 6 0 4 7	4	1212/997

It was necessary to consider 42574 nonisomorphic rim colorings. The worst cases for the different values of $\mu(w)$ are listed in Table 5.1. Each rim coloring is shown as a list of 12 numbers in four groups of three. Each group of three numbers corresponds to a neighbor z of w and represents the colors of the neighbors of z other than w. The notation for the colors is as explained in section 4 (although here the set of colors is $\{1, \ldots, 7\}$, with seven elements). By inspection, all of these worst-case rim colorings satisfy (5.1), from which the rapid mixing result follows.

The mixing rate may be bounded by considering the largest value of the proven bound over $\mu(w)$ in Table 5.1. This is 62/189. Thus $\mathbf{E}\left[H(X', Y') - 1\right] \leq -67/63n$. It follows then from Lemma 1.1 that the Markov chain $\mathcal{M}$ is rapidly mixing with mixing rate bounded above by

$$\frac{63}{67} n \log(n\varepsilon^{-1}).$$

Remark 6. A method is outlined in [7] whereby, under certain conditions, the rapid mixing of a given Markov chain can be used to deduce the rapid mixing of a second Markov chain with the same state space and stationary distribution as the first. Using this and the results of section 4 it is possible to prove that the simple Jerrum/Salas–Sokal chain is rapidly mixing when $k = 5$ for graphs of maximum degree 3. Similarly, using the results of this section it is possible to prove that the simple chain is rapidly mixing when $k = 7$ for triangle-free 4-regular graphs. The mixing rate of the Jerrum/Salas–Sokal chain is bounded above by $O(n^8 \log(n))$ in both cases.

6. A #*P*-completeness proof.

In this section we present a proof that the problem of counting the number of k-colorings of graphs with maximum degree Δ is #P-complete for fixed k, Δ such that $k \geq 3$ and $\Delta \geq 3$. First we must establish a preliminary result involving the path with $r+1$ vertices $u_1, \ldots, u_{r+1}$ and edges $\{u_i, u_{i+1}\}$ for $1 \leq i \leq r$.

LEMMA 6.1. *Let C_r denote the path with $r+1$ vertices $u_1, \ldots, u_{r+1}$ and r edges $\{u_i, u_{i+1}\}$ for $1 \leq i \leq r$. Let γ, γ' be two fixed, distinct colors and let σ_r, δ_r be defined by*

$$\sigma_r = |\{X \in \Omega_k(C_r) : X(u_1) = \gamma,\ \ X(u_{r+1}) = \gamma\}|,$$
$$\delta_r = |\{X \in \Omega_k(C_r) : X(u_1) = \gamma,\ \ X(u_{r+1}) = \gamma'\}|$$

for $r \geq 1$. *Then*

$$\sigma_r = \frac{(k-1)^r - (-1)^{r-1}(k-1)}{k} \qquad \textit{and} \qquad \delta_r = \frac{(k-1)^r + (-1)^{r-1}}{k}$$

for $r \geq 1$.

Proof. This result follows easily by induction, using the following observations. When $r = 1$, the equations give the correct values $\sigma_1 = 0$ and $\delta_1 = 1$. If $r > 1$, then

$$\sigma_r = (k-1)\delta_{r-1} \qquad \text{and} \qquad \delta_r = \sigma_{r-1} + (k-2)\delta_{r-1},$$

giving the inductive step. □

We now show that it is $\#P$-complete to count the number of proper k-colorings of arbitrary graphs of maximum degree Δ when $k \geq 3$ and $\Delta \geq 3$.

THEOREM 6.2. *Denote by* $\#kColMaxDeg\Delta$ *the problem of computing* $|\Omega_k(G)|$ *for an arbitrary graph* G *of maximum degree* Δ. *If* $k \geq 3$ *and* $\Delta \geq 3$, *then the problem* $\#kColMaxDeg\Delta$ *is* $\#P$*-complete.*

Proof. Let $\#kCol$ denote the problem of computing the number of proper k-colorings for an arbitrary graph. If $k \geq 3$, then by an immediate corollary of Vertigan's thesis (see [11]) the problem $\#kCol$ is $\#P$-complete.

Let Π be the problem which takes as an instance a graph G with maximum degree Δ and bipartition of the edge set $E = M \cup B$, and asks how many k-colorings of G are there which make every edge in M monochromatic and every edge in B bichromatic. There is an easy reduction from $\#kCol$ to Π, which we now describe. Given an instance H of $\#kCol$, define an instance G of Π as follows. Let the vertex set of G equal the vertex set of H initially and let B equal the edge set of H, $M = \emptyset$. Let Δ' be the maximum degree of H. Suppose first that $\Delta' > \Delta$. Then replace each vertex v with degree greater than Δ by a path of new vertices, each of degree at most Δ, joined together by edges in M. Ensure that each neighbor of v in H is connected (by an edge in B) to exactly one of these new vertices and that at least one of the new vertices has degree Δ. Next suppose that $\Delta' < \Delta$. In this case choose a vertex v of degree Δ' in H and make v a vertex of degree Δ in G, as follows. Add $\Delta - \Delta'$ vertices w to the vertex set of G, and make each new vertex a neighbor of v by adding the edge $\{v, w\}$ to M. The number of k-colorings of G which make all edges in M monochromatic and all edges in B bichromatic is given by $|\Omega_k(H)|$. This shows that $\#kCol$ is polynomial-time reducible to Π, so Π is $\#P$-complete.

We now give a polynomial-time reduction of Π to $\#kColMaxDeg\Delta$. Let H be an instance of Π with edge bipartition $E = M \cup B$. Let $m = |M|$ and let C_r be the path with $r+1$ vertices and r edges, as in Lemma 6.1. For $r \geq 1$, let G_r be the r-stretch of H with respect to M. That is, G_r is obtained from H by replacing each edge $\{v, w\}$ in M by a copy of the path C_r, with the identifications $u_1 = v$, $u_{r+1} = w$. Let N_s be the number of k-colorings of H where s edges in M are monochromatic and the remaining $m - s$ edges in M (together with all edges in B) are bichromatic. Then

$$|\Omega_k(G_r)| = \sum_{s=0}^{m} N_s \, {\sigma_r}^s \, {\delta_r}^{m-s} = {\delta_r}^m \sum_{s=0}^{m} N_s \, {x_r}^s, \tag{6.1}$$

where $x_r = \sigma_r/\delta_r$. We may express (6.1), for $1 \leq r \leq m+1$, as a matrix equation

$$\begin{pmatrix} {\delta_1}^{-m}|\Omega_k(G_1)| \\ {\delta_2}^{-m}|\Omega_k(G_2)| \\ \vdots \\ {\delta_{m+1}}^{-m}|\Omega_k(G_{m+1})| \end{pmatrix} = \begin{pmatrix} 1 & x_1 & \cdots & {x_1}^m \\ 1 & x_2 & \cdots & {x_2}^m \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m+1} & \cdots & {x_{m+1}}^m \end{pmatrix} \begin{pmatrix} N_0 \\ N_1 \\ \vdots \\ N_m \end{pmatrix}. \tag{6.2}$$

Now, using Lemma 6.1,

$$x_r = 1 - \frac{k}{(k-1)(1-k)^{r-1}+1}.$$

It follows that the values x_r are distinct. If we know the values $|\Omega_k(G_r)|$, for $1 \leq r \leq m+1$, then the values $N_0, N_1, \dots, N_m$ may be obtained in polynomial time by inverting the Vandermonde matrix in (6.2). Since N_m is the answer which satisfies Π, this completes the reduction. Therefore $\#kColMaxDeg\Delta$ is $\#P$-complete. □

By Theorem 6.2, the counting problem associated with the Jerrum/Salas–Sokal chain [12, 18] and the Dyer–Greenhill chain [8] is $\#P$-complete. Also, by Theorem 6.2 with $\Delta = 3$ and $k = 5$, the counting problem associated with the Markov chain of section 4 is $\#P$-complete.

Further $\#P$-completeness results for graph colorings can be found in [10]. For example, we show that the problem of counting k-colorings in graphs with maximum degree Δ remains $\#P$-complete when restricted to triangle-free Δ-regular graphs, so long as $k \geq 3$ and $\Delta \geq 3$. Thus, the specific counting problem associated with the Markov chain of section 5 is $\#P$-complete.

REFERENCES

[1] D. Aldous, *Random walks on finite groups and rapidly mixing Markov chains*, in Séminaire de Probabilités XVII 1981/1982, Lecture Notes in Math. 986, A. Dold and B. Eckmann, eds., Springer-Verlag, New York, 1983, pp. 243–297.

[2] K. Appel and W. Haken, *Every Planar Map is Four Colorable*, Contemp. Math. 98, American Mathematical Society, Providence, RI, 1989.

[3] R. Bubley and M. Dyer, *Path coupling: A technique for proving rapid mixing in Markov chains*, in 38th Annual Symposium on Foundations of Computer Science, Los Alamitos, CA, IEEE, 1997, pp. 223–231.

[4] E. G. Coffman, D. S. Johnson, P. W. Shor, and R. R. Weber, *Markov chains, computer proofs and average-case analysis of best fit bin packing*, in 25th Annual Symposium on the Theory of Computing, New York, ACM, 1993, pp. 412–421.

[5] M. Creutz, *A Monte Carlo study of quantised SU(2) gauge theory*, Phys. Rev. D, 21 (1980), pp. 2308–2315.

[6] M. Dyer, A. Frieze, and R. Kannan, *A random polynomial-time algorithm for approximating the volume of convex bodies*, J. ACM, 38 (1991), pp. 1–17.

[7] M. Dyer and C. Greenhill, *On Markov Chains for Independent Sets*, preprint, 1997.

[8] M. Dyer and C. Greenhill, *A more rapidly mixing Markov chain for graph colorings*, Random Structures Algorithms, 13 (1998), pp. 285–317.

[9] K. Fredenhagen and M. Marcu, *A modified heat bath method suitable for Monte Carlo simulations on vector and parallel processing*, Phys. Lett. B, 193 (1987), pp. 486–488.

[10] C. Greenhill, *The complexity of counting colorings and independent sets in sparse graphs and hypergraphs*, Comput. Complexity, to appear.

[11] F. Jaeger, D. L. Vertigan, and D. Welsh, *On the computational complexity of the Jones and Tutte polynomials*, Math. Proc. Cambridge Philos. Soc., 108 (1990), pp. 35–53.

[12] M. Jerrum, *A very simple algorithm for estimating the number of k-colorings of a low-degree graph*, Random Structures Algorithms, 7 (1995), pp. 157–165.

[13] M. Jerrum and A. Sinclair, *Approximating the permanent*, SIAM J. Comput., 18 (1989), pp. 1149–1178.

[14] M. Jerrum and A. Sinclair, *The Markov chain Monte Carlo method: An approach to approximate counting and integration*, in Approximation Algorithms for NP-Hard Problems, D. Hochbaum, ed., PWS Publishing, Boston, 1996, pp. 482–520.

[15] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani, *Random generation of combinatorial structures from a uniform distribution*, Theoret. Comput. Sci., 43 (1986), pp. 169–188.

[16] B. Krekó, *Linear Programming*, Sir Isaac Pitman and Sons Ltd., London, 1968.

[17] R. Y. Rubenstein, *Simulation and the Monte Carlo Method*, John Wiley, New York, 1981.

[18] J. Salas and A. D. Sokal, *Absence of phase transition for antiferromagnetic Potts models via the Dobrushin uniqueness theorem*, J. Statist. Phys., 86 (1997), pp. 551–579.

[19] B. Toft, *Coloring, stable sets and perfect graphs*, in Handbook of Combinatorics, R. L. Graham, M. Grötschel, and L. Lovász, eds., Elsevier, Amsterdam, 1995, pp. 233–288.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 401–415

RANKING PRIMITIVE RECURSIONS: THE LOW GRZEGORCZYK CLASSES REVISITED*

STEPHEN J. BELLANTONI† AND KARL-HEINZ NIGGL‡

Abstract. Traditional results in subrecursion theory are integrated with the recent work in "predicative recursion" by defining a simple ranking ρ of all primitive recursive functions. The hierarchy defined by this ranking coincides with the Grzegorczyk hierarchy at and above the linear-space level. Thus, the result is like an extension of the Schwichtenberg–Müller theorems. When primitive recursion is replaced by recursion on notation, the same series of classes is obtained except with the polynomial time computable functions at the first level.

Key words. subrecursion, computational complexity, predicativity, ramified recursion, Heinermann classes, polynomial time, linear space

AMS subject classifications. 03D20, 68Q15, 03D15

PII. S009753979528175X

1. Introduction.

A variety of restricted recursion schemes has been successfully used to characterize some common complexity classes; see, e.g., Clote [6] for a survey. At first characterized using explicit bounds on the value computed by a recursion, some of these function classes were later characterized by restricting the way in which values can be accessed during the recursion. These latter schemes are called "predicative recursion" [4] or "ramified recurrence" [11] schemes; the phrase "safe recursion" has also been used. Taking the predicative viewpoint, one has two types of values: values which are known in their entirety and which therefore can be examined completely, e.g., by being recursed upon; and those values which are still emerging and which therefore can be accessed only in a more restricted way, e.g., by examining a few low-order bits. One then develops a class of functions over these two types. Although one can develop a formal mechanism having more types of values, doing so does not usually allow one to define more functions: Leivant showed that strictly predicative systems having more than two types collapse to the system having only two types [11].

A different way to look at ramification is in terms of the amount or structure of the recursions performed. One thinks of a loose analogy with the number of comprehension levels used to define a particular set in ramified set theory. A definition of the function is given first, then one examines that definition to see how many "ramification levels" are used by it. In this sense, ramification is "implicit" in the derivation of the function as built up from the initial functions of the class using the derivation rules of composition and recursion. Rather than explicitly controlling the type or quality of the values appearing during the calculation, one measures the amount or structure of the work that, implicitly, must have been performed in order to produce the value. Of course, in order to use this approach one must restrict attention to func-

*Received by the editors January 24, 1995; accepted for publication (in revised form) August 11, 1998; published electronically October 5, 1999.
http://www.siam.org/journals/sicomp/29-2/28175.html

†151 Howland Avenue, Toronto, ON M5R 3B4, Canada (sjb@cs.toronto.edu).

‡Technische Universität Ilmenau, Fakultät für Informatik und Automatisierung, PF 100565, 98684 Ilmenau, Germany (niggl@theoinf.tu-ilmenau.de). The author was affiliated with Ludwig-Maximilans-Universität, München, while doing this work. The research of this author was supported by a DFG grant.

tion definitions for which one can always ascribe ramification levels; in the present work, the primitive recursive functions are used. A major purpose of this paper is to propose a specific way of measuring the amount or structure of primitive recursive derivations. The measure should classify derivations by specifying the number of ramification levels that are implicitly used.

An obvious proposal for such a classification would be, according to the minimal degree of derivations of the function, using an idea which dates back to the early 1960s: the degree, *deg*, of a derivation concluded by a recursion rule is one more than the maximum degree of the subderivations; for a composition rule the degree is just the maximum of the degrees of the subderivations. This simple definition has the advantage of coinciding with the Grzegorczyk hierarchy $\mathcal{E}^r$ at and above the elementary functions: $\mathcal{E}^{r+1} = \mathcal{D}^r$ for $r \geq 2$, where $\mathcal{D}^r$ are the primitive recursive derivations with degree at most r. As discussed by Clote [6], the characterization for $r \geq 3$ was shown by Schwichtenberg [23], and later the case of $r = 2$ was shown by Müller [15]. Another possible classification was given prior to Müller's result by Parsons [20], who examined whether the step function (i.e., the function h in $f(x+1) = h(x, f(x))$) accesses the critical value. This result also characterized the Grzegorczyk hierarchy at the elementary functions and above. A more detailed discussion of other earlier work is given after the statement of the results in section 4.

In this paper we propose a new ranking ρ of the primitive recursive functions. Like *deg*, the ranking ρ characterizes the Grzegorczyk hierarchy at and above the elementary level; but at level 1 it characterizes the linear-space computable functions —by Ritchie's result [22], this is $\mathcal{E}^2$. Thus, $\mathcal{E}^{r+1} = \mathcal{PR}_1^r$ for $r \geq 1$, where $\mathcal{PR}_1^r$ consists of the primitive recursive derivations with rank at most r. In this sense, the result is like an extension of the Schwichtenberg–Müller theorems. Unlike the Schwichtenberg and Müller proofs, the proof of this result does not refer directly to any computation model. A natural series of classes $\mathcal{PR}_2^r$ is also obtained down to the first level, when primitive recursion is replaced with recursion on notation. In fact, $\mathcal{E}^{r+1} = \mathcal{PR}_2^r$ for $r \geq 2$, but $\mathcal{PR}_2^1$ characterizes the polynomial-time computable functions.

The μ measure of Niggl [17], a slight modification of [16], provides similar characterizations of the complexity classes discussed here with respect to classes $\mathcal{R}_1^n$ and $\mathcal{R}_2^n$. The former are based on primitive recursion, the latter on recursion on notation. In fact, the μ measure operates on algorithms given as lambda terms over ground-type variables. The measure μ accounts for redundant input positions and therefore is able to distinguish between proper and improper recursions, i.e., recursions in which the critical input position of the recurrence function is not used. Thus μ does not require any initial functions other than zero and its successor. Remarkably, the proofs of $\mathcal{R}_2^1 = \text{FPTIME}$ and $\mathcal{E}^{r+1} = \mathcal{R}_1^n$ for $r \geq 1$ use the same method, thus emphasizing the uniform method of determining the computational complexity of primitive recursive algorithms.

The measure ρ is convenient because it classifies derivations in an arguably more natural way than *deg* does. For example, the natural derivation of the multiplication function uses two recursions. The Schwichtenberg–Müller approach classifies this derivation in the same way as the exponential, as level 2. Although there is a derivation of multiplication that has *deg* equal to 1, one typically must pass through a Turing machine simulation in order to find it. In contrast, the ranking ρ proposed here classifies the natural derivation of multiplication as level 1, and exponentiation as level 2.

A parallel of these developments can be carried out in the setting of formal logic.

This paper is a companion to [1], which shows how to define weak subsystems of arithmetic by defining the "rank" of proofs instead of by bounding the induction formulas. Rank r functions turn out to be exactly those having a rank r proof of convergence. The intuition of ramification levels is made more precise by the model-theoretic analysis in [1]; also see Leivant [10].

The results of this paper should be of some interest to complexity theorists as well as recursion theorists. Complexity characterizations based on the structure of recursive derivations, without explicit bounds on time or memory resources, can help to ground the concepts of computational complexity by providing a reference point other than the original resource-based definitions.

Notation. Iterations of a function f are defined by $f^{(0)}(x) = x$ and $f^{(n+1)}(x) = f(f^{(n)}(x))$. The binary length of x is $|x| = (\mu n \geq 0)(p^{(n)}(x) = 0)$, where $p(x) \equiv \lfloor x/2 \rfloor$. Vector notation is used freely throughout this work; for example, $|\vec{x}|$ means the vector $|x_1|, \ldots, |x_m|$, where m is the number of elements (perhaps zero) in vector $\vec{x}$. Sometimes vectors are treated as if they were sets, and vice versa. When $\vec{x}$ is the empty vector, one assumes $\max \vec{x} = 0$. The arity of function f is indicated by m_f. Throughout, the domain and range are assumed to be the nonnegative numbers. We avoid writing λ and instead casually use expressions such as "2^x" to mean the function $\lambda x.2^x$.

2. Existing subrecursive classes. First we would like to review some basic and well-known facts about subrecursive function classes defined over the nonnegative integers.

The primitive recursive functions, $\mathcal{D}$, are usually defined by closing the set of initial functions 0, $Sx = x + 1$, and $\pi^n_k(\vec{x}) = x_k$ (for $1 \leq k \leq n$, all $n \geq 0$) under the following primitive recursion and composition rules. Let $P(x) = \max(0, x - 1)$.

• Composition: given h and $g_1, \ldots, g_m$ with $m = m_h \geq 1$, derive f such that $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$.

• Primitive recursion: given g and h, derive f such that $f(0, \vec{y}) = g(\vec{y})$ and $f(x, \vec{y}) = h(Px, \vec{y}, f(Px, \vec{y}))$ for $x \neq 0$.

Researchers have frequently used definitions similar to the following degree *deg* for function derivations in $\mathcal{D}$. The definition of *deg* can be applied equally well to the classes $\mathcal{PR}_1$ and $\mathcal{PR}_2$ defined later in this paper.

• If f is an initial function, then $deg(f) = 0$.

• If f is defined by (any form of) composition from h and $g_1, \ldots, g_m$, then $deg(f) = \max\{deg(h), deg(g_1), \ldots, deg(g_m)\}$.

• If f is defined by (any form of) recursion with step function h and base function g, then $deg(f) = \max\{deg(g), 1 + deg(h)\}$.

Let $\mathcal{D}^r = \{f \in \mathcal{D} : deg(f) \leq r\}$, the primitive recursive derivations with *deg* at most r.

The degree of functions is closely related to the Grzegorczyk hierarchy, with classes $\mathcal{E}^n$. The definition of $\mathcal{E}^n$ refers to the Ackermann branches $A_n(x, y) = A(n, x, y)$, where A is the Ackermann function

$$\begin{aligned} A(0, x, y) &= y + 1, \\ A(n + 1, x, 0) &= \text{"if } n = 0, \text{ then } x \text{ else if } n = 1, \text{ then } 0 \text{ else } 1\text{"} \\ A(n + 1, x, y + 1) &= A(n, x, A(n + 1, x, y)). \end{aligned}$$

Note that $A_0(x, y) = Sy$, $A_1(x, y) = x + y$, $A_2(x, y) = x \cdot y$, and $A_3(x, y) = x^y$. Let $\mathcal{E}^r$ consist of the initial functions 0, π^n_k, S, and A_r closed under composition and "bounded primitive recursion," stated next. For later use we also define "bounded recursion on notation."

• Bounded primitive recursion: given g, h, and k, obtain f such that $f(0,\vec{y}) = g(\vec{y})$ and $f(x,\vec{y}) = h(Px,\vec{y},f(Px,\vec{y}))$ for $x \neq 0$, provided that for all $x,\vec{y},\ f(x,\vec{y}) \leq k(x,\vec{y})$.

• Bounded recursion on notation: given g, h_0, h_1, and k, obtain f such that $f(0,\vec{y}) = g(\vec{y})$ and $f(x,\vec{y}) = h_i(px,\vec{y},f(px,\vec{y}))$ for $x \neq 0$, where $i = (x \bmod 2)$, provided that for all $x,\vec{y},\ f(x,\vec{y}) \leq k(x,\vec{y})$.

The following central results were established early on. See Clote [6] for a definition of the Kalmar elementary functions and for discussion of these theorems.

THEOREM 2.1 (see Schwichtenberg [23]; Müller [15]). *For $r \geq 2$, $\mathcal{E}^{r+1} = \mathcal{D}^r$.*

THEOREM 2.2 (see Ritchie [22]). *$\mathcal{E}^2$ consists of the linear-space computable functions.*

THEOREM 2.3 (see Cobham [7]). *The polynomial-time computable functions are exactly those in the class obtained from the initial functions 0, π_i^n, $s_0x = 2x$, $s_1x = 2x+1$, and $x\#y \equiv 2^{|x|\cdot|y|}$ using composition and the rule of bounded recursion on notation.*

THEOREM 2.4 (see Grzegorczyk [8]). *$\mathcal{E}^3$ consists of the Kalmar elementary functions.*

3. Ranking recursions.

The development begins by defining function classes $\mathcal{PR}_1$ and $\mathcal{PR}_2$.

Let I_1 consist of the following initial functions: constant 0; successor $Sx = x+1$; predecessor $P(x) = \max(0, x-1)$; and conditional $C(x,y,z) =$ "if $x = 0$, then y else z."

Let I_2 consist of constant 0; successors $s_0x = 2x$ and $s_1x = 2x+1$; predecessor $p(x) = \lfloor x/2 \rfloor$; and conditional $c(x,y,z) =$ "if $x \bmod 2 = 0$, then y else z."

Define $\mathcal{PR}_1$ to be the smallest set of function derivations containing a derivation for each function in I_1 and closed under the following rules of "full" composition and recursion. These full closure rules differ slightly from the ordinary ones stated earlier; this minor difference will be compensated for by the difference in the initial functions. We use the full rules in order to simplify the definition of ρ and to improve the similarity between $\mathcal{PR}_2$ and $\mathcal{PR}_1$.

• Full composition: given derivations h and $g_1,\ldots,g_m$ with $m = m_h \geq 1$, derive f such that $f(\vec{x}) = h(g_1(\vec{x}^1),\ldots,g_m(\vec{x}^m))$, where each $\vec{x}^j$ is a vector consisting of elements from the vector $\vec{x}$ (possibly with repetitions, omissions, or changes in order).

• Full primitive recursion: given derivations g and h, derive f such that $f(0,\vec{y}) = g(\vec{y})$ and $f(x,\vec{y}) = h(x,\vec{y},f(Px,\vec{y}))$ for $x \neq 0$.

Similarly, define $\mathcal{PR}_2$ to be the least set of function derivations containing I_2 and closed under the same rules, except using p instead of P in the recursion rule (now called "full recursion on notation" instead of "full primitive recursion").

For convenience, we have defined $\mathcal{PR}_1$ and $\mathcal{PR}_2$ to be sets of derivations rather than sets of functions, because we will be working most often with the derivation rather than the function itself. Given a derivation f, we sometimes ambiguously also use f to refer to the function defined by the derivation.

Now we assign a *rank*, $\rho(f,i)$, for each derivation f and for each position $1 \leq i \leq m_f$. The rank of an input position is supposed to be an indication of the amount of recursion on that input; or, it is the number of ramification levels of "knowledge" about that input which are required in order to "know" the result of the function. In defining ρ, one uses the idea that when f is derived from subfunctions $\vec{h}$, then there is some sense in which the rank of input positions of $\vec{h}$ contributes to the rank of input positions of f. For example, if $f(x) = h(0, g(x))$, then one requires $\rho(f,1) \geq \rho(h,2)$ and $\rho(f,1) \geq \rho(g,1)$. The key point is that recursion makes it strictly more difficult to understand how a value is used. If $f(x) = h(x, f(px))$, then one requires $\rho(f,1) \geq 1 + \rho(h,2)$ as well as $\rho(f,1) \geq \rho(h,1)$.

The rank $\rho(f,i)$ is defined for both $\mathcal{PR}_2$ and $\mathcal{PR}_1$ as follows:

• If f is an initial function, then $\rho(f,i) = 0$ for $1 \leq i \leq m_f$.

• If f is defined by the full composition $h(g_1(\vec{x}^1), \ldots, g_m(\vec{x}^m))$, then $\rho(f,i) = \max(\{\rho(h,k) : x_i \in \vec{x}^k\} \cup \{\rho(g_k, j) : x_i \equiv (\vec{x}^k)_j\})$ for $1 \leq i \leq m_f$.

• If f is defined by full recursion from base function g and step function h, then $\rho(f,1) = \max\{\rho(h,1),\, 1 + \rho(h, m_h)\}$ and $\rho(f,i) = \max\{\rho(h,i),\, \rho(h,m_h),\, \rho(g,i-1)\}$ for $2 \leq i \leq m_f$.

Define the rank of function derivations f by $\rho(f) = \max\{\rho(f,i) : 1 \leq i \leq m_f\}$.

Define $\mathcal{PR}_1^r = \{f \in \mathcal{PR}_1 : \rho(f) \leq r\}$ and $\mathcal{PR}_2^r = \{f \in \mathcal{PR}_2 : \rho(f) \leq r\}$.

One can see that the formulation of full composition has been designed to give rank 0 to input positions that appear in the defined function $f(\vec{x})$ but which do not appear in any of the sublists $\vec{x}^1, \ldots, \vec{x}^m$. A spuriously higher rank would have been assigned if we had used projection functions and ordinary composition.

It is shown below that, without loss of generality, every $\mathcal{PR}_i^r$ function derivation has all subderivations also in $\mathcal{PR}_i^r$. On the other hand, $\mathcal{PR}_i^r$ is closed under full composition, because the rank of a function defined by full composition is bounded by the greater of the ranks of the subderivations. It follows that $\mathcal{PR}_i^r$ is equivalent to the least class of function derivations containing the initial functions and closed under full composition and the following rule:

• r-safe recursion: given g and h, derive f by full recursion, provided that $\rho(f) \leq r$. This generalizes the rule of safe recursion ($r = 1$) in [4].

Rank differs from "tiers" (Leivant [11]) because rank, as defined by ρ, is not a type system. This is not merely a syntactic distinction, as the present system admits more instances of composition and recursion than the stricter system in [11]. In particular, the systems $\mathcal{PR}_1$ and $\mathcal{PR}_2$ contain all the primitive recursive functions and separates at all levels, while the system in [11] collapses to level 2. However, the philosophical underpinning of rank is similar to that of tiers: by stratifying or ramifying the definition of a function, one obtains at the same time a more predicative definition and a more computationally tractable one.

LEMMA 3.1 (ρ-normalization). *If f is a derivation whose conclusion has rank r, then there is a derivation f' defining the same function with the same input ranks, in which every subderivation has rank at most r.*

Proof. Essentially, the rank of a derivation can be reduced only by composing a constant into a recursion position. Such a constant-depth recursion can be unrolled as a sequence of compositions; the compositions do not increase the rank above the rank of the subderivations.

Say that a derivation f' is *equivalent* to another derivation f if the functions defined by f' and f are the same, and $\rho(f',i) = \rho(f,i)$ for $1 \leq i \leq m_f$. We prove the following strengthened statement: any derivation $f^{\vec{a}}$ obtained by substituting zero or more constants $\vec{a}$ for some of the inputs of a derivation f has an equivalent derivation f' in which every subderivation has rank at most $\rho(f^{\vec{a}})$. The proof is by induction on the derivation f.

Technically, $f^{\vec{a}}$ is obtained from f by a full composition in which each g_i is either some a_j or else is the rank 0 identity function, either $P(S(x_i))$ or $p(s_0(x_i))$. This use of the identity function does not affect the rank; therefore we omit discussion of it below.

The strengthened statement is trivial for the initial functions, since their derivations have no subderivations. Next, consider f defined by recursion from g and h. We consider two subcases.

The first subcase is when a constant a is substituted for the recursion variable and, without loss of generality, other constants $\vec{c}$ are substituted for the subsequent non-recursion variables, i.e., $f^{a,\vec{c}}(\vec{y}) = f(a, \vec{c}, \vec{y})$. Using the induction hypothesis on g and h for each constant $1 \leq j \leq a$, one has derivations g' and h'_j equivalent to $g^{\vec{c}}(\vec{y}) = g(\vec{c}, \vec{y})$ and $h^{j,\vec{c}}(\vec{y}, z) = h(j, \vec{c}, \vec{y}, z)$, such that g' and h'_j have all subderivations of rank at most $\rho(g^{\vec{c}})$ and $\rho(h^{j,\vec{c}})$, respectively. If $a \neq 0$, then define $f'(\vec{y})$ by a compositions of h'_j $(1 \leq j \leq a)$ starting on g' (in the case of $\mathcal{PR}_1$, or by $|a|$ compositions in the case of $\mathcal{PR}_2$). But if $a = 0$, define $f'(\vec{y}) = C(0, g'(\vec{y}), h'_1(\vec{y}, g'(\vec{y})))$ (for $\mathcal{PR}_1$, or use conditional c instead of conditional C for $\mathcal{PR}_2$). The latter construction defines the same function as $g^{\vec{c}}$ but provides the same rank as in $f^{a,\vec{c}}$ for inputs $\vec{y}$. In either case, the defined f' is equivalent to $f^{a,\vec{c}}$ and subderivations have the required rank.

The second subcase is when no constant is substituted into the recursion position of f, e.g., $f^{\vec{a}}(x, \vec{y}) = f(x, \vec{a}, \vec{y})$. Here, the induction hypothesis is applied to both g and h with respect to $\vec{a}$ to get suitable functions g' and h'. Then defining f' by recursion on these gives the required equivalent derivation.

Consider a function $f^{\vec{a}}$, obtained from a function $f(\vec{x}) = h(g_1(\vec{x}^1), \ldots, g_m(\vec{x}^m))$ by substituting arbitrary constants $\vec{a}$ for some of the inputs. Let $g_j^{\vec{a}}$ be g_j with some of the inputs replaced by the constants among $\vec{a}$ that are selected by the list $\vec{x}^j$. Observe that by the definition of ρ, $\rho(g_j^{\vec{a}}) \leq \rho(f^{\vec{a}})$. Applying the induction hypothesis on each $g_j^{\vec{a}}$ gives equivalent functions g'_j in which the rank of every subderivation is at most $\rho(f^{\vec{a}})$. Now let $\vec{c}$ be constants equal to the values of the constant expressions, if any, among $g'_1, \ldots, g'_m$. Applying the induction hypothesis to h with constants $\vec{c}$, one obtains an equivalent function h'. Again all subderivations have rank at most $\rho(h, \vec{c}) \leq \rho(f^{\vec{a}})$. Finally, f' is obtained by composing h' with the remaining (non-constant) functions g'_j. This defines f' equivalent to $f^{\vec{a}}$ with all subderivations of rank at most $\rho(f^{\vec{a}})$. $\square$

Of course, the translation defined by this proof could result in a serious blow-up of the derivation size, but this is not a concern in the current work.

4. Consolidated characterizations. The main results are stated here in a consolidated form and they are compared to previous work. Like *deg*, the ranking ρ characterizes $\mathcal{E}^{r+1}$ for $r \geq 2$. However, it also characterizes $\mathcal{E}^2$. Furthermore, ρ gives natural characterizations when recursion on notation is considered.

Among other characterizations, the following theorem gives $\mathcal{PR}_1^2 = \mathcal{PR}_2^2 = \mathcal{E}^3 =$ the Kalmar elementary functions, and $\mathcal{PR}_1^1 = \mathcal{E}^2 =$ FLINSPACE (the linear-space computable functions).

THEOREM 4.1.

1. $\mathcal{PR}_1^r = \mathcal{E}^{r+1}$ *for* $r \geq 1$.
2. $\mathcal{PR}_2^r = \mathcal{E}^{r+1}$ *for* $r \geq 2$.
3. $\mathcal{PR}_2^1 =$ FPTIME *(the polynomial-time computable functions).*
4. $\mathcal{PR}_1 = \mathcal{PR}_2 = \mathcal{D}$ *(the primitive recursive functions).*

Proof. Each part of items 1–3 is restated and proved as one of the theorems in section 6.

Item 4 is an easy corollary. By items 1 and 2 one obtains $\mathcal{PR}_2 = \mathcal{PR}_1$. The equivalence of $\mathcal{PR}_1$ and $\mathcal{D}$ is obtained by straightforward simulations. For example, the projection functions are computed in $\mathcal{PR}_1$ by the full composition $\pi_i(\vec{x}) = C(0, C(x_i, x_i, x_i), 0)$. $\square$

Oitavem [19] has shown that the Kalmar functions are obtained by adding s_0 as an initial function to the definition of $\mathcal{PR}_1^1$. In the current work, extra rank is used instead: the Kalmar functions are shown to equal $\mathcal{PR}_1^2$ (and $\mathcal{PR}_2^2$). The use of extra rank locates the Kalmar functions above the linear-space functions in a hierarchy.

The Kalmar functions were given a resource-free recursive characterization by Leivant [14] using higher-type recurrence. The present results do not use higher-type recursion.

This work extends the polynomial-time characterization of Bellantoni and Cook [4]. That result is listed above, but it is seen from a new perspective. Rather than defining a restricted class (FPTIME) using the rule of 1-safe recursion, one instead considers all recursion-on-notation derivations and then classifies them according to ρ. The set of derivations f such that $\rho(f) = 1$ are derivations of all the FPTIME functions.

A linear-space characterization similar to the current one was first proved by Bellantoni in his thesis [3, p. 49], at the suggestion of S. Cook; related linear-space results were proved independently by Handley [9] and Leivant [11]. The characterization was adapted and reproved by Nguyen [18] in her work on linear-space reasoning.

The current characterizations of $\mathcal{E}^{r+1}$ can also be compared to Leivant's characterization of Grzegorczyk classes using "coerced recurrence"; see [12, section 4]. Coerced recurrence seems to be more powerful than stratified recurrence. To restrict it, Leivant uses an approach similar to Parson's earlier definition ([20, p. 358]): one separately determines whether the step function uses the critical term. Leivant defined a hierarchy by putting the predicative recurrence rules at levels 0 and 1, together with the Parsons-like rules at levels 2 and above (see [12, section 4]). This hierarchy, which is similar to the present one, has desirable properties: it characterizes an interesting class at level 1, and it corresponds to the Grzegorczyk classes at and above the elementary level. Despite these desirable features, the definition by itself does not seem to integrate the lower levels with the higher ones. One can argue that ρ provides a more integrated definition. The characterization using ρ also differs in that it is type free, requires only one recursion scheme rather than two (stratified and coerced), and does not determine whether the step function uses the critical term.

Predicative recursion results for some other space-bounded complexity classes are available (see Oitavem [19]; Bellantoni [2]; Leivant and Marion [13]; Bloch [5]). With respect to parallel complexity classes, current results using unbounded recursion schemes (Bloch [5]; Leivant and Marion [13]; Bellantoni [3, p. 69] and [2]) are either unable to characterize the computationally interesting class NC or else require awkward constructions to do so.[1]

If ρ is taken as determining a fundamental hierarchy, then one discards classes $\mathcal{E}^0$ and $\mathcal{E}^1$ in favor of $\mathcal{PR}_1^0$. Functions in the class $\mathcal{PR}_1^0$ are compositions of the initial functions, i.e., functions computable in constant time, counting one time step for each initial function application. This seems to be quite a bit less than the functions in $\mathcal{E}^0$ or $\mathcal{E}^1$, which are defined by arbitrary amounts of bounded primitive recursion (albeit with a small size bound).

[1]Note added in press: A more satisfactory characterization recently appeared in D. Leivant, *A characterization of NC by tree recurrence*, in Proceedings of the 1998 IEEE Symposium on Foundations of Computer Science, IEEE, 1998, pp. 716–724.

The remainder of the paper is devoted to proving the results. We introduce Grzegorczyk polynomials and prove some simple facts about them. The statement of the key bounding lemma uses Grzegorczyk polynomials as the bounding functions. The required characterizations items 1–3 can be derived based on the bounding lemma.

5. Grzegorczyk polynomials. Bounding terms will be defined using an analogue, at each level of the Grzegorczyk hierarchy, of the successor, addition, and multiplication functions. These functions, indicated by S^r, $+^r$, and $\times^r$ for $r \geq 1$, are defined by

$$\begin{aligned}
S^1(x) &= S(x),\\[1ex]
0 +^r a &= a,\\
x +^r a &= S^r(Px +^r a) \quad \text{for } x \neq 0,\\[1ex]
0 \times^r y &= 0,\\
x \times^r y &= y +^r (Px \times^r y) \quad \text{for } x \neq 0,\\[1ex]
S^{r+1}(x) &= S^r((x +^r x) \times^r (x +^r x)).
\end{aligned}$$

Thus, $x +^r a$ is x applications of S^r starting at a and $x \times^r y$ is x applications of $y+^r$ starting at 0, that is, $x \cdot y$ applications of S^r starting at 0. Equivalently, $x \times^r y = (x \cdot y) +^r 0$. Under these definitions, S^1 is the ordinary successor function, and $+^1$ and $\times^1$ are ordinary addition and multiplication. At the next level, $+^2$ and $\times^2$ are elementary. Observe that all the functions $S^r, +^r, \times^r$ are monotone increasing. They are also monotone with respect to the level r; e.g., if $r' \leq r$, then $x +^{r'} y \leq x +^r y$ for all x and y.

A Grzegorczyk polynomial, or simply a polynomial, is an expression built up from nonnegative integer constants, input variables, and the functions $+^r$ for $r \geq 1$, $\times^r$ for $r \geq 1$, and max. All these functions are definable in $\mathcal{PR}_1$, but in considering Grzegorczyk polynomials we mean to use them as initial functions and then perform compositions.

Definition 5.1. *Define rank for polynomials as follows. For the primitives (with $r \geq 1$)*

$$\begin{aligned}
&\nu(x,1) = 0 \textit{ for variable } x,\\
&\nu(\max,1) = \nu(\max,2) = 0,\\
&\nu(+^r,1) = r \textit{ and } \nu(+^r,2) = r-1,\\
&\nu(\times^r,1) = \nu(\times^r,2) = r.
\end{aligned}$$

If the polynomial $q(\vec{x})$ is given by the expression $q_0(q_1(\vec{x}^1), \ldots, q_k(\vec{x}^k))$ (where each $\vec{x}^k \subseteq \vec{x}$), then define

$$\nu(q,i) = \max\{\, \nu(q_0,k),\, \nu(q_k,j) : \ \vec{x}_i \equiv (\vec{x}^k)_j \}.$$

When z is the ith variable in the list $\vec{x}$, we may write $\nu(q,z)$ for $\nu(q,i)$.

Intuitively, ν is very much like ρ, except that max is considered an initial function.

When we see that the outermost operation of a polynomial is $\times^r$, we know that all variables have rank at least r. But when we see that the outermost operation is $+^r$, we know that variables in the left subterm have rank at least r, and those in the right have at least $r-1$.

Grzegorczyk polynomials are a way of lifting ordinary polynomials into all levels of the Grzegorczyk hierarchy. The lifted functions do not have all the properties of the ordinary functions, such as commutativity of $+$. However, they do have many desirable properties, allowing us to generalize the proofs of [4] to all levels of the Grzegorczyk hierarchy. Some desirable properties of Grzegorczyk polynomials are as follows: (1) a restricted kind of associativity for $+$; (2) the fact that rank r polynomials are definable in $\mathcal{E}^{r+1}$; (3) the fact that $+^{r+1}$ grows faster than any polynomial of rank r; and (4) the ability to "separate" any polynomial q at any level r to get $q \le q' +^{r+1} \max \vec{x}^*$, where $\vec{x}^*$ are the variables in q of rank at most r, and where q' does not refer to $\vec{x}^*$. These facts are proved in the remainder of this section.

Lemma 5.2 (semiassociativity). *$x +^{r'} (y +^r z) \le (x +^{r'} y) +^r z$ for all x, y, and z and all $r' \le r$.*

Proof. The assertion is proved by calculating $x +^{r'} (y +^r z) = (S^{r'})^{(x)}((S^r)^{(y)}(z)) \le (S^r)^{(x)}((S^r)^{(y)}(z)) = (S^r)^{(x+y)}(z) \le (S^r)^{(x+^{r'}y)}(z) = (x +^{r'} y) +^r z$. ∎

Lemma 5.3 (definability). *Every Grzegorczyk polynomial q belongs to $\mathcal{E}^{\nu(q)+1}$.*

Proof. We first show by induction on $r \ge 1$ that $S^r \in \mathcal{E}^r$, and $+^r, \times^r \in \mathcal{E}^{r+1}$. In the base case, one has $S^1 \equiv S \in \mathcal{E}^0$, $+^1 \in \mathcal{E}^1$, and $\times^1 \in \mathcal{E}^2$, as the latter two are ordinary addition and multiplication. For the step case, the induction hypothesis yields $S^r \in \mathcal{E}^r$ and $+^r, \times^r \in \mathcal{E}^{r+1}$. This implies $S^{r+1} \in \mathcal{E}^{r+1}$ by definition of S^{r+1}. It is well known from [23] that any function defined by primitive recursion from functions in $\mathcal{E}^n$, $n \ge 2$, belongs to $\mathcal{E}^{n+1}$. Thus $+^{r+1} \in \mathcal{E}^{r+2}$. Since $x \times^{r+1} y = (x \cdot y) +^{r+1} 0$, we can define $\times^{r+1} \in \mathcal{E}^{r+1}$ using $+^{r+1}$. This concludes the induction showing $S^r \in \mathcal{E}^r$ and $+^r, \times^r \in \mathcal{E}^{r+1}$.

Now consider a Grzegorczyk polynomial q, and let $r = \nu(q)$. We define q in $\mathcal{E}^{r+1}$ by induction on the structure of q. If q is a constant or a variable, then it is easily defined in $\mathcal{E}^{r+1}$. If q is defined by $q_1 +^s q_2$ or $q_1 \times^s q_2$ for $s \le r$, or by $\max(q_1, q_2)$, then we are done using the induction hypothesis and using the closure of $\mathcal{E}^{r+1}$ under composition (note $\max \in \mathcal{E}^1$). If q is $q_1 +^s q_2$ or $q_1 \times^s q_2$ for $s \ge r+1$, then either q does not contain any variables or q is of the form $c +^{r+1} q_2$ for a constant c. For the case of q having the form $c +^{r+1} q_2$ for a constant c, we can define q in $\mathcal{E}^{r+1}$ using c compositions of S^{r+1} onto the definition of q_2, using $q_2 \in \mathcal{E}^{r+1}$ by the induction hypothesis. ∎

Lemma 5.4 (domination). *For every Grzegorczyk polynomial $q(\vec{x})$ there is a constant c_q satisfying $q(\vec{x}) \le c_q +^{\nu(q)+1} \max \vec{x}$ for all $\vec{x}$.*

Proof. The proof is by induction on the structure of q. If q is a constant, then the result is immediate using q as the required constant c_q; if q is a variable, then it is immediate using $c_q = 0$. Otherwise, $q(\vec{x}) = q_1(\vec{x}^1) \circ q_2(\vec{x}^2)$, where $\circ$ is either $+^s$, $\times^s$, or max. Let $r = \nu(q)$.

Suppose that $\circ$ is $+^s$ or $\times^s$ with $s \le r$, or $\circ$ is max. In these cases, observe that for every y, $y \circ y \le S^r((y +^r y) \times^r (y +^r y)) = S^{r+1}(y)$. Applying the induction hypothesis to obtain c_{q_1} and c_{q_2}, define $c = \max(c_{q_1}, c_{q_2})$ so that

$$\begin{aligned} q(\vec{x}) &\le (c +^{r+1} \max \vec{x}) \circ (c +^{r+1} \max \vec{x}) \\ &\le S^{r+1}(c +^{r+1} \max \vec{x}) \\ &= (c+1) +^{r+1} \max \vec{x}. \end{aligned}$$

If $\circ$ is $+^s$ with $s \ge r+2$, or is $\times^s$ with $s \ge r+1$, then both $\vec{x}^1$ and $\vec{x}^2$ are empty (since otherwise these variables would have rank greater than r, by the definition of ν). In these cases q is a constant expression and the result is easy.

If $\circ$ is $+^s$ with $s = r+1$, then $\vec{x}^1$ is empty, again by definition of ν. In this case, q_1 is a constant expression and $q(\vec{x})$ is equivalent to $c +^{r+1} q_2(\vec{x}^2)$ for some constant c. Applying the induction hypothesis on q_2, one obtains $q(\vec{x}) \le c +^{r+1} (c_{q_2} +^{r+1} \max \vec{x}^2) \le (c +^{r+1} c_{q_2}) +^{r+1} \max \vec{x}$. ◻

LEMMA 5.5 (separation). *Let q be a polynomial over $\vec{x}$, let r be a nonnegative integer, and define $\vec{x}' = \langle x_i : \nu(q, x_i) \ge r+1\rangle$ and $\vec{x}^* = \langle x_i : \nu(q, x_i) \le r\rangle$. Then there is a polynomial q' over $\vec{x}'$ such that*

$$q(\vec{x}) \le q'(\vec{x}') +^{r+1} \max \vec{x}^*, \tag{1}$$

$$\nu(q', z) = \nu(q, z) \text{ for } z \in \vec{x}'. \tag{2}$$

Proof. The proof is by structural induction on q. For q a constant or variable, note $\vec{x}' \equiv \langle\rangle$ and $\vec{x}^* \equiv \vec{x}$ because all the inputs to q are rank 0. So if q is a constant, defining $q' = c$ will do. Otherwise, q is a variable and we can define $q' = 0$; the statements (1) and (2) follow directly.

For the step case of the induction, one has $q(\vec{x}) = q_1(\vec{x}^1) \circ q_2(\vec{x}^2)$, where $\circ$ is either $+^s$ or $\times^s$ or max, and where $\vec{x}^1 \cup \vec{x}^2 \subseteq \vec{x}$. Consider cases on $\circ$.

Suppose $\circ$ is $\times^s$ for some $s \ge r+1$. Then $\vec{x}' = \vec{x}$ and $\vec{x}^* = \langle\rangle$ because for each $z \in \vec{x}$ one has $\nu(q_1 \times^s q_2, z) \ge s \ge r+1$. Defining $q'(\vec{x}') = q(\vec{x})$, the properties (1) and (2) follow trivially.

For a more difficult case, suppose $\circ$ is $+^s$ for some $s \ge r+1$. If $s > r+1$, then again $\vec{x}' = \vec{x}$ and $\vec{x}^* = \langle\rangle$ because for $z \in \vec{x}^1$ one has $\nu(q_1 +^s q_2, z) \ge s > r+1$ and for $z \in \vec{x}^2$ one has $\nu(q_1 +^s q_2, z) \ge s-1 \ge r+1$. Otherwise, $s = r+1$. Then $\vec{x}^1 \subseteq \vec{x}'$ because all the variables in the subexpression $q_1(\vec{x}^1)$ obtain rank at least $r+1$ due to their appearance on the left side of $+^{r+1}$. Applying the induction hypothesis to $q_2(\vec{x}^2)$, one obtains $q_2'((\vec{x}^2)')$, where $(\vec{x}^2)'$ are the rank at least $r+1$ variables in q_2. Defining $q'(\vec{x}') = q_1(\vec{x}^1) +^{r+1} q_2'((\vec{x}^2)')$ one has

$$\begin{aligned} q(\vec{x}) &\le q_1(\vec{x}^1) +^{r+1} (q_2'((\vec{x}^2)') +^{r+1} \max(\vec{x}^2)^*) \\ &\le (q_1(\vec{x}^1) +^{r+1} q_2'((\vec{x}^2)')) +^{r+1} \max \vec{x}^* \end{aligned}$$

as required for statement (1) of the lemma. Statement (2) for q' follows by statement (2) of the induction hypothesis on q_2 together with the definition of ν.

If $\circ$ is $+^s$ for $s \le r$, then we apply the induction hypothesis to both q_1 and q_2 with r, to obtain $q_1(\vec{x}^1) \le q_1'((\vec{x}^1)') +^{r+1} \max(\vec{x}^1)^*$ and $q_2(\vec{x}^2) \le q_2'((\vec{x}^2)') +^{r+1} \max(\vec{x}^2)^*$. Using the domination lemma on the polynomial $y +^s z$ and the fact that $r+1 \ge s+1$, let c be such that $c +^{r+1} \max(y, z) \ge y +^s z$ for all y, z. Then

$$\begin{aligned} q(\vec{x}) &\le (q_1'((\vec{x}^1)') +^{r+1} \max(\vec{x}^1)^*) +^s (q_2'((\vec{x}^2)') +^{r+1} \max(\vec{x}^2)^*) \\ &\le c +^{r+1} \max(q_1'((\vec{x}^1)') +^{r+1} \max(\vec{x}^1)^*, q_2'((\vec{x}^2)') +^{r+1} \max(\vec{x}^2)^*) \\ &\le c +^{r+1} (\max(q_1((\vec{x}^1)'), q_2((\vec{x}^2)')) +^{r+1} \max((\vec{x}^1)^* \cup (\vec{x}^2)^*)) \\ &\le (c +^{r+1} \max(q_1((\vec{x}^1)'), q_2((\vec{x}^2)'))) +^{r+1} \max((\vec{x}^1)^* \cup (\vec{x}^2)^*). \end{aligned}$$

Observing that rank $r+1$ or greater variables in q_1 or q_2 are also rank $r+1$ or greater in q, and that rank at most r variables in q_1 and q_2 are also rank at most r in q, one has $(\vec{x}^1)' \cup (\vec{x}^2)' \subseteq \vec{x}'$ and $(\vec{x}^1)^* \cup (\vec{x}^2)^* \subseteq \vec{x}^*$. Defining then $q'(\vec{x}') = c +^{r+1} \max(q_1'((\vec{x}^1)'), q_2'((\vec{x}^2)'))$, we have demonstrated property (1) and must verify property (2). By the induction hypothesis, every variable of rank at least $r+1$ in q_1

or q_2 has the same rank in q'_1 or q'_2, respectively. In q the subterms are combined with $+^s$, which does not change the rank of any of these variables, since these variables all have rank at least $r+1 > s$. In q' the subterms are combined with max, which also does not change the rank of any variable in the subterms. Property (2) follows.

For $\times^s$ with $s \le r$, one proceeds in the same way as for the case of $+^s$ with $s \le r$. A similar proof also works for max. ◻

6. Bounding lemma. The bounding lemma proved in this section is central to all the results, as it shows how the recursive layering indicated by ρ corresponds to bounded function growth rate. The latter in turn corresponds to bounds on the space or time of a computation.

One can just as well prove the bounding lemma at the same time for $\mathcal{PR}_2^r$ as for $\mathcal{PR}_1^r$. The proofs differ in the size measure used: the relevant measures are $\|x\|_1 \equiv x$ for $\mathcal{PR}_1$ and $\|x\|_2 \equiv |x|$ for $\mathcal{PR}_2$. In the following, one uses the fact that $S\|Px\|_1 = \|x\|_1$, or correspondingly $S\|px\|_2 = \|x\|_2$, for $x \neq 0$. We write $\|\cdot\|$ to mean either $\|\cdot\|_1$ or $\|\cdot\|_2$, depending on the class under discussion. The vector notation $\|\vec{x}\|$ means $\langle \|x_1\|, \dots, \|x_m\| \rangle$, where m is the number of elements in $\vec{x}$.

LEMMA 6.1 (bounding). *For every $f \in \mathcal{PR}_i$ $(i \in \{1,2\})$ there is a Grzegorczyk polynomial q_f over m_f variables such that $\|f(\vec{x})\|_i \le q_f(\|\vec{x}\|_i)$ and for $1 \le i \le m_f$, $\rho(f,i) = \nu(q_f,i)$.*

Proof. If f is one of the initial functions, then one assigns $q_f(\vec{x}) = 1 +^1 \max(\vec{x})$, and the required properties follow directly.

If f is defined by full composition, say $f(\vec{x}) = h(g_1(\vec{x}^1), \dots, g_n(\vec{x}^n))$, then one correspondingly defines q_f so that $q_f(\|\vec{x}\|) = q_h(q_{g_1}(\|\vec{x}^1\|), \dots, q_{g_n}(\|\vec{x}^n\|))$ using the polynomials obtained by the induction hypothesis. The statement of the lemma follows using monotonicity of polynomials to achieve the bounding expression and using the definitions of ρ and ν to achieve the equality of the ranks.

The last case is when f is defined by full recursion, say in $\mathcal{PR}_2$,

$$\begin{aligned} f(0,\vec{y}) &= g(\vec{y}), \\ f(x,\vec{y}) &= h(x,\vec{y}, f(px,\vec{y})) \quad \text{for } x \neq 0. \end{aligned}$$

The induction hypothesis gives $q_g(\vec{v})$ and $q_h(u,\vec{v},w)$ with input ranks equal to the input ranks of g and h. Letting $r = \rho(h, m_h)$, one applies the separation lemma to obtain q'_h such that

$$q_h(u,\vec{v},w) \le q'_h((u\vec{v})') +^{r+1} \max((u\vec{v})^*, w),$$

where $(u\vec{v})'$ is a sublist corresponding to input positions of h (or q_h) having rank at least $r+1$, and $(u\vec{v})^*, w$ is a sublist corresponding to the input positions of h (or q_h) having rank at most r. The separation lemma also gives equality between the ranks of the inputs that are in both q_h and q'_h. Now define q_f by

$$q_f(u,\vec{v}) = (u \times^{r+1} q'_h((u\vec{v})')) +^{r+1} \max((u\vec{v})^*, q_g(\vec{v})).$$

To prove the bound (1) for f, one uses induction on $\|x\|$, where the base case follows from $\|f(0,\vec{y})\| \le q_g(\|\vec{y}\|) \le q_f(\|0,\vec{y}\|)$. For the step case,

$$\begin{aligned} \|f(x,\vec{y})\| &\le q_h(\|x,\vec{y}, f(px,\vec{y})\|) \\ &\le q'_h(\|x,\vec{y}\|') +^{r+1} \max(\|x,\vec{y}\|^*, q_f(\|px,\vec{y}\|)) \\ &\le q'_h(\|x,\vec{y}\|') +^{r+1} \big((\|px\| \times^{r+1} q'_h(\|px,\vec{y}\|')) +^{r+1} \max(\|x,\vec{y}\|^*, q_g(\|\vec{y}\|))\big) \\ &\le \big(\|x\| \times^{r+1} q'_h(\|x,\vec{y}\|')\big) +^{r+1} \max(\|x,\vec{y}\|^*, q_g(\|\vec{y}\|)) \\ &\le q_f(\|x,\vec{y}\|). \end{aligned}$$

All that remains is to show that the ranks of variables in q_f are the same as in f.

First consider u. If $u \in (u\vec{v})'$, then $\nu(q'_h, 1) = \nu(q_h, 1) \geq r+1$ by the definition of $(u\vec{v})'$ and (2) of the separation lemma. Using the induction hypothesis on h, this gives $\rho(h,1) = \nu(q_h, 1) \geq r+1$. In this case, applying ν to q_f gives $\nu(q_f, 1) = \nu(q'_h, 1)$; also, applying ρ to f gives $\rho(f,1) = \rho(h,1)$. Thus $\rho(f,1) = \nu(q_f,1)$ in this case. The other case concerning the induction variable is when $u \in (u\vec{v})^*$, that is, $\rho(h,1) = \nu(q_h,1) \leq r$. In this case, $\rho(q_f, 1) = r+1$ due to the presence of u on the left side of $+^{r+1}$ in the definition of q_f. Correspondingly, $\rho(f,1) = \max(\rho(h,1), r+1) = r+1$ by the definition of ρ. This finishes the proof of $\rho(f,1) = \nu(q_f,1)$.

Now consider one of the parameter positions, $2 \leq i \leq m_f = m_h - 1$, where an input v_{i-1} appears in q_f. If $v_{i-1} \in (u\vec{v})'$, then $\nu(q_h, i) \geq r+1$. Using the equality of ranks in q_h with ranks in q'_h, one has $\nu(q_f, i) = \max(\nu(q_h, i), \nu(q_g, i-1))$ by the definition of ν on q_f. By the induction hypothesis, $\nu(q_h, i) = \rho(h,i)$ and $\nu(q_g, i-1) = \rho(g, i-1)$, leading to $\nu(q_f, i) = \max(\rho(h,i), \rho(g,i-1)) = \rho(f,i)$ by the definition of $\rho(f,i)$ with $\rho(h,i) \geq r+1 > \rho(h, m_h)$. The remaining case is when v_{i-1} does not appear in $(u\vec{v})'$, that is, $\nu(q_h, i) \leq r$. By the induction hypothesis on h, one has $\rho(h,i) = \nu(q_h,i) \leq r$, leading to $\rho(f,i) = \max(\rho(g,i-1), r)$. By the induction hypothesis on g this value is $\max(\nu(q_g, i-1), r)$. The definition of ν on q_f, using the fact that $v_{i-1} \notin (u\vec{v})'$, gives $\max(\nu(q_g, i-1), r) = \nu(q_f, i)$. This finishes the proof that $\rho(f,i) = \nu(q_f,i)$ for $2 \leq i \leq m_f$. $\square$

7. Proofs of characterization theorems. In this section we prove the characterizations which were summarized in section 4. The Schwichtenberg–Müller theorem is generalized. One direction uses the bounding lemma proved above, while the other direction is obtained by generalizing the simulation lemmas of [4] and [3] to all levels of the Grzegorczyk hierarchy. Finally, concerning $\mathcal{PR}_2^1$, we rely on the results of [4]. It is of some interest that the proofs for $r \geq 2$ do not refer to any computation model, unlike the Schwichtenberg and Müller proofs.

LEMMA 7.1 ($\mathcal{E}$-bounding). *For $r \geq 0$, every $f \in \mathcal{E}^{r+2}$ has a monotone increasing bound $b_f \in \mathcal{PR}_1^{r+1}$.*

Proof. Since S, $+^1$, and $\times^1$ are defined in $\mathcal{PR}_1^1$, we already have that A_0, A_1, and A_2 are in $\mathcal{PR}_1^1$. In fact, under these derivations, $\rho(A_2, 1) = \rho(A_2, 2) = 1$. Now note that for $r \geq 0$,

$$A_{r+3}(x, 0) = 1,$$
$$A_{r+3}(x,y) = A_{r+2}(x, A_{r+3}(x, Py)) \quad \text{for } y \neq 0.$$

It follows by induction on r that A_{r+3} is defined in $\mathcal{PR}_1$ with $\rho(A_{r+3}, 1) = r+1$ and $\rho(A_{r+3}, 2) = r+2$. Next note that max is definable in $\mathcal{PR}_1^1$ by $\max(x,y) = C(Sy \dot{-} x, x, y)$, where cutoff subtraction is defined by $x \dot{-} 0 = x$ and $x \dot{-} y = P(x \dot{-} Py)$ for $y \neq 0$. By a theorem in [23, p. 87], every $f \in \mathcal{E}^{r+2}$ satisfies $f(\vec{x}) \leq A_{r+3}(\max(2, \vec{x}), c_f)$ for some constant c_f. Since A_{r+3} is defined in $\mathcal{PR}_1$ with $\rho(A_{r+3}, 1) = r+1$, the composition $A_{r+3}(\max(2,\vec{x}), c_f)$ gives a bounding function for $f(\vec{x})$ in $\mathcal{PR}_1^{r+1}$. $\square$

LEMMA 7.2 (generalized simulation). *For every $f \in \mathcal{E}^{r+2}$ there is a derivation $f^* \in \mathcal{PR}_1^1$ and a monotone increasing function $w_f \in \mathcal{PR}_1^{r+1}$ such that $f^*(w, \vec{x}) = f(\vec{x})$ for all $w \geq w_f(\vec{x})$. In fact, $\rho(f^*, 1) \leq 1$ and for $2 \leq i \leq m_f$, $\rho(f^*, i) = 0$.*

Proof. The proof is by induction on the definition of $f \in \mathcal{E}^{r+2}$. If f is one of the initial functions $0, S, \Pi_i^n$, then $f^*(w, \vec{x}) = f(\vec{x})$ and $w_f(\vec{x}) = 0$ will do.

If f is the initial function A_{r+2}, then observe that f is definable in $\mathcal{E}^{r+2}$ using A_{r+2} itself to bound the recursions. Therefore, to obtain the statement of the lemma for A_{r+2}, one can apply the method below for compositions and bounded recursions to the definition of A_{r+2}.

Suppose that f is defined by a composition in $\mathcal{E}^{r+2}$, say $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$. The induction hypothesis provides suitable simulations $h^*, g_1^*, \ldots, g_m^*$ with monotone increasing functions $w_h, w_{g_1}, \ldots, w_{g_m} \in \mathcal{PR}_1^{r+1}$. By the $\mathcal{E}$-bounding lemma, there is a function $b \in \mathcal{PR}_1^{r+1}$ which is a monotone increasing bound on all $g_1, \ldots, g_m$. Therefore one defines $f^*(w, \vec{x}) = h^*(w, g_1^*(w, \vec{x}), \ldots, g_m^*(w, \vec{x}))$ and $w_f(\vec{x}) = w_h(b(\vec{x}), \ldots, b(\vec{x})) + \sum_j w_{g_j}(\vec{x})$.

Consider f defined by a bounded recursion in $\mathcal{E}^{r+2}$, such as $f(0, \vec{y}) = g(\vec{y})$, $f(x + 1, \vec{y}) = h(x, \vec{y}, f(x, \vec{y}))$, and $f(x, \vec{y}) \le k(x, \vec{y})$; where $g, h, k \in \mathcal{E}^{r+2}$. First note that the characteristic function $x \le y$ can be defined by $C(Sy \dot{-} x, 1, 0)$ with $\rho(\le, 1) = 1$ and $\rho(\le, 2) = 0$. Using the induction hypothesis, let

$$\begin{aligned} \hat{f}(0, w, x, \vec{y}) &= g^*(w, \vec{y}), \\ \hat{f}(v, w, x, \vec{y}) &= \text{if } v \le x, \text{ then } h^*(w, Pv, \vec{y}, \hat{f}(Pv, w, x, \vec{y})) \text{ else } \hat{f}(Pv, w, x, \vec{y}) \ (v \ne 0). \end{aligned}$$

Also define $w_f(x, \vec{y}) = w_h(x, \vec{y}, b(x, \vec{y})) + w_g(\vec{y}) + x$, where $b \in \mathcal{PR}_1^{r+1}$ is a monotone increasing bound on k obtained from the $\mathcal{E}$-bounding lemma. Given the ranks of the input position for $\le$ and for h' and g', we have $\rho(\hat{f}, 1) = \rho(\hat{f}, 2) = 1$ and $\rho(\hat{f}, i+2) = 0$ for $1 \le i \le m_f$. Moreover, if $w \ge w_f(x, \vec{y})$, then a subsidiary induction on v shows that $\hat{f}(v, w, x, \vec{y}) = f(v, \vec{y})$ for $v \le x$ and $\hat{f}(v, w, x, \vec{y}) = f(x, \vec{y})$ for $v \ge x$. Thus, the lemma is finished by defining $f^*(w, x, \vec{y}) = \hat{f}(w, w, x, \vec{y})$. $\square$

Finally, we can prove the main results.

THEOREM 7.3 (characterization). *For* $n \ge 1$, $\mathcal{E}^{r+1} = \mathcal{PR}_1^r$.

Proof. One direction follows from the generalized simulation lemma: given $f \in \mathcal{E}^{r+1}$ with $r \ge 1$, we obtain f^* such that $f(\vec{x}) = f^*(w_f(\vec{x}), \vec{x})$, where $f^* \in \mathcal{PR}_1^1$ and $w_f \in \mathcal{PR}_1^r$. Thus $f \in \mathcal{PR}_1^r$ as desired.

In the other direction, we are given a derivation $f \in \mathcal{PR}_1^r$ and need to show that the function computed by f is in $\mathcal{E}^{r+1}$. First we apply the ρ-normalization lemma to f, to ensure that all subderivations are also in $\mathcal{PR}_1^r$. Now we proceed by induction on the structure of f. In the initial cases $0, S, P$, we have $f \in \mathcal{E}^0$ and $C \in \mathcal{E}^1$. In the case of a full composition of h and $\vec{g}$, the ranks of h and $\vec{g}$ are at most r (by ρ-normalization) and therefore this case follows from the induction hypothesis, using composition in $\mathcal{E}^{r+1}$ with the projection functions. Finally, if f is defined by full recursion from g and h, the induction hypothesis gives g and h in $\mathcal{E}^{r+1}$. By the bounding lemma and the definability lemma, f is bounded by a function in $\mathcal{E}^{r+1}$. Thus f is definable by bounded primitive recursion in $\mathcal{E}^{r+1}$. $\square$

THEOREM 7.4 (see Bellantoni and Cook [4]). $\mathcal{PR}_2^1 =$ FPTIME.

Proof. Essentially this was proved in [4], to which the reader should refer. The definition of $\mathcal{PR}_2^1$ corresponds closely to the definition of the class B in [4], with rank 0 input positions being "safe" and rank 1 input positions being "normal." Although there are slight differences in the details of the formulations, such as the use of full composition instead of ordinary composition, these are inessential. $\square$

THEOREM 7.5 (equivalence of higher levels). *For* $r \ge 2$, $\mathcal{PR}_2^r = \mathcal{PR}_1^r$.

Proof. The first part of the lemma is to show $\mathcal{PR}_1^r \subseteq \mathcal{PR}_2^r$, for $r \ge 2$. First define, in $\mathcal{PR}_2^2$, the concatenation function $\oplus$ and the unary conversion function $u(x) = 2^x - 1$:

$$\begin{aligned} x \oplus 0 &= x, \\ x \oplus y &= c(y, s_0(x \oplus py), s_1(x \oplus py)), \\ u(0) &= 0, \\ u(x) &= c(x, u(px) \oplus u(px), s_1(u(px) \oplus u(px))). \end{aligned}$$

Under this definition, $|u(x)| = x$, and the number of steps in a full recursion on notation on $u(x)$ is the same as the number of steps in a full primitive recursion on x. Also, $p(u(x)) = u(Px)$. We write $u(\vec{x})$ for $u(x_1), \dots, u(x_m)$, where m is the number of elements in $\vec{x}$.

To show $\mathcal{PR}_1^r \subseteq \mathcal{PR}_2^r$, one proceeds by induction on the structure of a $\mathcal{PR}_1$ derivation f, defining a corresponding $\mathcal{PR}_2$ derivation f' such that $f'(u(\vec{x})) = u(f(\vec{x}))$ for all $\vec{x}$, and such that $\rho(f', i) = \rho(f, i)$ for all i. Each case of the induction is straightforward: for initial functions $0, S, P, C$ we use $0, s_1, p, c$ (note $u(x) = 2^x - 1$); if f is defined by full primitive recursion on g and h, then f' is defined by full recursion on notation on g' and h' (note $p(u(x)) = u(Px)$); and for full compositions in $\mathcal{PR}_1$ we use full compositions in $\mathcal{PR}_2$. Having finished the structural induction, one can observe that $f(\vec{x}) = |f'(u(\vec{x}))|$ is a computation of f in $\mathcal{PR}_2^{\max(\rho(f),2)}$, because $u \in \mathcal{PR}_2^2$ and $|\cdot| \in \mathcal{PR}_2^1 =$ FPTIME.

The second part is to show $\mathcal{PR}_2^r \subseteq \mathcal{PR}_1^r$, for $r \geq 2$. Using the characterization theorem, $\mathcal{E}^{r+1} \subseteq \mathcal{PR}_1^r$ for $r \geq 2$, so it is enough to show that for $r \geq 2$, $\mathcal{PR}_2^r \subseteq \mathcal{E}^{r+1}$.

Let f be any derivation in $\mathcal{PR}_2$, and let $r = \rho(f)$; we show that the function computed by f is in $\mathcal{E}^{\max(3,r+1)}$. Applying the ρ-normalization lemma, we can assume that all subderivations of f are in $\mathcal{PR}_2^r$. If f is an initial function, i.e., s_0, s_1, p, or c, then the same function is defined in FLINSPACE $= \mathcal{E}^2$. If f is defined by a full composition $h(g_1(\vec{x}^1), \dots, g_m(\vec{x}^m))$, then by ρ-normalization and the induction hypothesis we can compute h and $\vec{g}$ in $\mathcal{E}^{\max(3,r+1)}$, and therefore by composition $f \in \mathcal{E}^{\max(3,r+1)}$. The last case is when f is defined by full recursion on notation, say, $f(0, \vec{y}) = g(\vec{y})$ and $f(x, \vec{y}) = h(x, \vec{y}, f(px, \vec{y}))$ for $x \neq 0$. In this case, a function to extract the high-order n bits of a value x is defined by $e(x, n) = p^{(|x| \dot{-} n)}(x)$ (where "$\dot{-}$" is the cutoff subtraction defined earlier). The e function, as well as the length function $|\cdot|$ and comparison $\leq$, is in FLINSPACE $= \mathcal{E}^2$. The induction hypothesis yields $g, h \in \mathcal{E}^{\max(3,r+1)}$. Now define a function $\hat{f}$ by $\hat{f}(0, x, \vec{y}) = g(\vec{y})$ and $\hat{f}(n, x, \vec{y}) =$ "if $n \leq |x|$, then $h(e(x, n), \vec{y}, \hat{f}(Pn, x, \vec{y}))$ else $\hat{f}(Pn, x, \vec{y})$" for $n \neq 0$. An induction on n, for $0 \leq n \leq |x|$, shows that $\hat{f}(n, x, \vec{y}) = f(e(x, n), \vec{y})$. For $n \geq |x|$, one has $\hat{f}(n, x, \vec{y}) = f(x, \vec{y}) = f(e(x, n), \vec{y})$ due to the "else" clause. Thus the definition $f(x, \vec{y}) = \hat{f}(|x|, x, \vec{y})$ will complete the proof, provided that the recursion defining $\hat{f}$ can be carried out in $\mathcal{E}^{\max(3,r+1)}$, that is, provided we have a bounding function for $\hat{f}$ in $\mathcal{E}^{\max(3,r+1)}$. Since $\hat{f}(n, x, \vec{y}) \leq f(e(x, n), \vec{y})$ and $e \in \mathcal{E}^2$, a bound on f is sufficient to give a bound on $\hat{f}$. By the bounding lemma for $\mathcal{PR}_2^r$, one has a Grzegorczyk polynomial q of rank r such that $|f(\vec{x})| \leq q(|\vec{x}|)$. Therefore $f(\vec{x}) \leq 2^{q(|\vec{x}|)}$. By the definability lemma, $q \in \mathcal{E}^{r+1}$. Since exponentiation is in $\mathcal{E}^3$, we have $2^{q(|\vec{x}|)} \in \mathcal{E}^{\max(3,r+1)}$. Therefore, f is bounded by an $\mathcal{E}^{\max(3,r+1)}$ function. □

8. Conclusion. A detailed analysis of the nesting of recursions has yielded benefits at the low levels of the Grzegorczyk hierarchy. Ramified recursion has been reframed in the traditional context of counting the nesting of recursions. Computationally interesting classes are seen to be characterized by this "structural" analysis of function derivations, which is based on a novel way of counting recursions. This integrates older characterizations at and above the elementary level with newer characterizations below the elementary level.

All the results obtained here are for recursions in type level 0, i.e., the value computed by the recursion is an integer. An extension of these results to higher-type recursion would be of interest.

Acknowledgments. Thanks to the University of Toronto Department of Computer Science, the Center for Discrete Mathematics and Theoretical Computer Science) at Rutgers, and the Ludwig-Maximilans-Universität München. K.-H. Niggl gratefully acknowledges a DFG grant.

REFERENCES

[1] S. Bellantoni, *Ranking arithmetic proofs by implicit ramification,* in Proof of Complexity and Feasible Arithmetics, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 39, AMS, Providence, RI, 1998, pp. 37–57.

[2] S. Bellantoni, *Predicative recursion and the polytime hierarchy,* in Feasible Mathematics II, P. Clote and J. B. Remmel eds., Birkhäuser, Boston, 1995, pp. 15–29.

[3] S. Bellantoni, *Predicative Recursion and Computational Complexity,* Tech. report 264/92, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1992.

[4] S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the polytime functions,* Comput. Complexity, 2 (1992), pp. 97–110.

[5] S. Bloch, *Function-algebraic characterizations of log and polylog parallel time,* Comput. Complexity, 4 (1994), pp. 175–205.

[6] P. Clote, *Computation models and function algebras,* in Handbook of Recursion Theory, Ed Griffor, ed., Elsevier, New York, 1996, pp. 1–90.

[7] A. Cobham, *The intrinsic computational difficulty of functions,* 1965 Logic, Methodology and Philos. Sci., Proc. 1964 Internat. Cong., Y. Bar-Hillel, ed., North–Holland, Amsterdam, 1965.

[8] A. Grzegorczyk, *Some classes of recursive functions,* Rozprawy Mat., 4 (1953).

[9] W. G. Handley, *Bellantoni and Cook's Characterization of Polynomial Time Functions,* manuscript, August 1992.

[10] D. Leivant, *Intrinsic theories and computational complexity,* Lecture Notes in Comput. Sci. 960, Springer-Verlag, New York, 1995, pp. 177–194.

[11] D. Leivant, *Ramified recurrence and computational complexity* I: *Word recurrence and polytime,* in Feasible Mathematics II, P. Clote and J. B. Remmel eds., Birkhäuser, Boston, 1995, pp. 321–343.

[12] D. Leivant, *Stratified functional programs and computational complexity,* in Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, New York, 1993, ACM, New York, pp.

[13] D. Leivant, and J.-Y. Marion, *Ramified recurrence and computational complexity* II: *Substitution and poly-space,* in Computer Science Logic, J. Tiuryn and L. Pacholsky, eds., Lecture Notes in Comput. Sci. 933, Springer-Verlag, Berlin, 1995, pp. 486–500.

[14] D. Leivant, *Ramified recurrence and computational complexity* III: *Higher-type recurrence and elementary complexity,* in Annals of Pure and Applied Logic 96, M. Fitting, R. Ramanujam, and K. Georgatos, eds., 1999, p. 209.

[15] H. Müller, *Klassifizierungen der primitiv rekursiven Funktionen*, Ph.D. thesis, Universität Münster, 1974.

[16] K.-H. Niggl, *Towards the computational complexity of $\mathcal{PR}^{\omega}$-terms,* Ann. Pure Appl. Logic, 75 (1995), pp. 153–178.

[17] K.-H. Niggl, *The μ-Measure as a Tool for Classifying Computational Complexity,* http: www.eiche.theoinf.tu-ilmenau.de/ niggl; to appear in Archive for Mathematical Logic.

[18] A. Nguyen, *A Formal System For Linear-Space Reasoning*, Tech. report 330/96, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1993.

[19] I. Oitavem, *New recursive characterizations of the elementary functions and the functions computable in polynomial space,* Rev. Mat. Univ. Complut Madrid, 10 (1997), pp. 109–125.

[20] C. Parsons, *Hierarchies of primitive recursive functions,* Z. Math. Logik Grundlagen Math., 14 (1968), pp. 357–376.

[21] G. Plotkin, *LCF considered as a programming language,* Theoret. Comput. Sci., 5 (1977), pp. 223–255.

[22] R. W. Ritchie, *Classes of predictably computable functions,* Trans. Amer. Math. Soc., 106 (1963), pp. 139–173.

[23] H. Schwichtenberg, *Rekursionszahlen und die Grzegorczyk-Hierarchie,* Arch. Math. Logik, Grundlagen forsch., 12 (1969), pp. 85–96.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 416–432

COMMUNICATION-EFFICIENT PARALLEL SORTING[*]

MICHAEL T. GOODRICH[†]

Abstract. We study the problem of sorting n numbers on a p-processor bulk-synchronous parallel (BSP) computer, which is a parallel multicomputer that allows for general processor-to-processor communication rounds provided each processor sends and receives at most h items in any round. We provide parallel sorting methods that use internal computation time that is $O(\frac{n \log n}{p})$ and a number of communication rounds that is $O(\frac{\log n}{\log(h+1)})$ for $h = \Theta(n/p)$. The internal computation bound is optimal for any comparison-based sorting algorithm. Moreover, the number of communication rounds is bounded by a constant for the (practical) situations when $p \leq n^{1-1/c}$ for a constant $c \geq 1$. In fact, we show that our bound on the number of communication rounds is asymptotically optimal for the full range of values for p, for we show that just computing the "or" of n bits distributed evenly to the first $O(n/h)$ of an arbitrary number of processors in a BSP computer requires $\Omega(\log n / \log(h+1))$ communication rounds.

Key words. parallel algorithms, parallel sorting, parallel processing

AMS subject classifications. 68Q15, 68Q25, 68Q30

PII. S0097539795294141

1. Introduction.

Most of the research on parallel algorithm design in the 1970s and 1980s was focused on fine-grain massively parallel models of computation (e.g., see Akl [4], Bitton et al. [7], JáJá [26], Karp and Ramachandran [28], Leighton [32], and Reif [43]), where the ratio of memory to processors is fairly small (typically $O(1)$), and this focus was independent of whether the model of computation was a parallel random-access machine (PRAM) or a network model, such as a mesh-of-processors model. However, as more and more parallel computer systems are being built, researchers are realizing that processor-to-processor communication is a prime bottleneck in parallel computing (e.g., see Aggarwal, Chandra, and Snir [2], Bilardi and Preparata [6], Culler et al. [13], Kruskal, Rudolph, and Snir [30], Mansour, Nisan, and Vishkin [34], Mehlhorn and Vishkin [35], Papadimitriou and Yannakakis [38], and Valiant [47, 46]). The real potential of parallel computation, therefore, will most likely be realized only for coarse to medium-grain parallel systems, where the ratio of memory to processors is unbounded, for such systems allow an algorithm designer to balance communication latency with internal computation time. Indeed, this realization has given rise to several new computation models for parallel algorithm design, which all use what Valiant [46] calls "bulk synchronous" processing. In such a model an input of size n is distributed evenly across a p-processor parallel computer. In a single computation *round* (which Valiant calls a *superstep*) each processor may send and receive h messages and then perform an internal computation on its internal memory cells using the messages it has just received. The amount of additional "internal time" is allowed to be greater than h. To avoid any conflicts that might be caused by asynchronies in the network (whose topology is left undefined) the messages sent out in a round t by some processor cannot depend upon any messages that a processor

[*]Received by the editors November 1, 1995; accepted for publication (in revised form) July 8, 1998; published electronically October 5, 1999. This paper appeared in a preliminary form in the 28th *ACM Symposium on Theory of Computing*. This research was supported by NSF grants CCR-9300079 and CCR-9625289 and ARO grant DAAH04-96-1-0013.

http://www.siam.org/journals/sicomp/29-2/29414.html

[†]Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218 (goodrich@cs.jhu.edu).

receives in round t (but, of course, they may depend upon messages received in round $t-1$). We refer to this model of computation as the bulk-synchronous parallel (BSP) model.

1.1. The BSP computer. As with the PRAM family of computer models,[1] the BSP model is distinguished by the broadcast and combining abilities of the network connecting the various processors. In the weakest version, which is the only version Valiant [46] considers, the network may not duplicate or combine messages but instead may only realize h-relations between the processors. We call this the exclusive-read exclusive-write (EREW) BSP model, noting that it is essentially the same as a model Valiant elsewhere [47] calls the XPRAM and one that Gibbons [21] calls the EREW phase-PRAM. It is also the communication structure assumed by the LogP model [13, 29], which is the same as the BSP model except that the LogP model does not explicitly require bulk-synchronous processing.

It is also natural to allow for a slightly more powerful bulk-synchronous model, which we call the weak–concurrent-read exclusive-write (CREW) BSP model. In this model we assume processors are numbered 1, 2, . . ., p and that messages can be duplicated by the network so long as the destinations for any message are a contiguous set of processors $\{i, i+1, \ldots, j\}$. This is essentially the same as a model Dehne et al. [16] and Dehne, Fabri, and Rau–Chaplin [17] refer to as the coarse-grain multicomputer. In designing an algorithm for this model one must take care to ensure that, even with message duplication, the number of messages received by a processor in a single round is at most h. Nevertheless, as we demonstrate in this paper, this limited broadcast capability can sometimes be employed to yield weak-CREW BSP algorithms that are conceptually simpler than their EREW BSP counterparts.

We can also define more powerful instances of the BSP model, such as a CREW BSP model, which allows for arbitrary broadcasts, or even a concurrent-read concurrent-write (CRCW) BSP model, which also allows for messages to the same location to be combined (using some rule). These models correspond to models Gibbons [21] calls the CREW phase-PRAM and CRCW phase-PRAM. In fact, Nash et al. [36] consider even more-powerful bulk-synchronous models in which messages can be combined using more-powerful combining functions (such as "add" or "bitwise-and"). We personally feel that anything more powerful than a weak-CREW BSP computer is probably not a realistic parallel model, given the possible asynchronies a network may possess; hence, we restrict our attention in this paper to the EREW and weak-CREW versions of the BSP model.

The running time of a BSP algorithm is characterized by two parameters: T_C, the number of communication rounds, and T_I, the internal computation time, that is, the sum, taken over all rounds of the algorithm, of the maximum amount of internal computation time taken by any processor in that round. A prime goal in designing a BSP algorithm is to minimize both of these parameters. Alternatively, by introducing additional characterizing parameters of the BSP model, we can combine T_I and T_C into a single running time parameter, which we call the *combined running time*. Specifically, if we let L denote the latency of the network—that is, the worst-case time needed to send one processor-to-processor message—and we let g denote the time "gap" between consecutive messages received by a processor in a communication round, then we can characterize the total running time of a BSP computation as

[1]Indeed, a PRAM with the same number of processors and memory cells is a BSP model with $h = 1$, as is a *module parallel computer* (MPC) [35], which is also known as a *distributed-memory machine* (DMM) [27], for any memory size.

$O(T_I + (L + gh)T_C)$. Incidentally, this is also the running time of implementing a BSP computation in the analogous[2] LogP model [13, 29].

The goal of this paper is to further the study of bulk-synchronous parallel algorithms by addressing the fundamental problem of sorting n elements distributed evenly across a p-processor BSP computer.

1.2. Previous work on parallel sorting. Let us, then, briefly review a small sample of the work previously done for parallel sorting. Batcher [5] in 1968 gave what is considered to be the first parallel sorting scheme, showing that in a fine-grained parallel sorting network one can sort in $O(\log^2 n)$ time using $O(n)$ processors. Since this early work there has been much effort directed at fine-grain parallel sorting algorithms (e.g., see Akl [4], Bitton et al. [7], JáJá [26], Karp and Ramachandran [28], and Reif [43]). Nevertheless, it was not until 1983 that it was shown, by Ajtai, Komlós, and Szemerédi [3], that n elements can be sorted in $O(\log n)$ time with an $O(n \log n)$-size network (see also Paterson [39]). In 1985 Leighton [31] extended this result to show that one can produce an $O(n)$-node bounded-degree network capable of sorting in $O(\log n)$ steps, based upon an algorithm he called "columnsort." In 1988 Cole [11] gave simple methods for optimal sorting in the CREW and EREW PRAM models in $O(\log n)$ time using $O(n)$ processors, based upon an elegant "cascade mergesort" paradigm using arrays, and this result was recently extended to the Parallel Pointer Machine by Goodrich and Kosaraju [22]. Thus, one can sort optimally in these fine-grained models.

These previous methods are not optimal, however, when implemented in bulk-synchronous models. Nevertheless, Leighton's columnsort method [31] can be used to design a bulk-synchronous parallel sorting algorithm that uses a constant number of communication rounds, provided $p^3 \leq n$. Indeed, there is a host of published algorithms for achieving such a result when the ratio of input size to number of processors is as large as this. For example, a randomized strategy, called sample sort, achieves this result with high probability [8, 19, 20, 24, 25, 33, 42, 44], as do deterministic strategies based upon regular sampling [18, 37, 45, 48]. These methods based upon sampling do not seem to scale nicely for smaller n/p ratios, however. If columnsort is implemented in a recursive fashion, then it yields an EREW BSP algorithm that uses $T_C = O([\log n/\log(n/p)]^\delta)$ communication rounds and internal computation time that is $O(T_C(n/p)\log(n/p))$, where $\delta = 2/(\log 3 - 1)$, which is approximately 3.419. Using an algorithm they call "cubesort," Cypher and Sanz [14] show how to improve the T_C term in these bounds to be $O((25)^{(\log^* n - \log^*(n/p))}[\log n/\log(n/p)]^2)$, and Plaxton [40] shows how cubesort can be modified to achieve $T_C = O([\log n/\log(n/p)]^2)$. Indeed, Plaxton [41] can modify the "sharesort" method of Cypher and Plaxton [15] to achieve $T_C = O((\log n/\log(n/p))\log^2(\log n/\log(n/p)))$. Finally, Chvátal [10] describes an approach of Ajtai, Komlós, Paterson, and Szemerédi for adapting the sorting network of Ajtai, Komlós, and Szemerédi [3] to achieve a depth of $O(\log n/\log(n/p))$ where the basic unit in the network is a "black box" that can sort $\lceil n/p \rceil$ elements. An effective method for constructing such a network is not included in Chvátal's report, however, for the method he describes is a nonuniform procedure based upon the probabilistic method. In addition, the constant factor in the running time appears to be fairly large. Incidentally, these latter methods [10, 15, 14, 31, 40] are actually defined for more-restrictive BSP models where the data elements cannot be duplicated and each

[2]There is also an o parameter in the LogP model, but it would be redundant with L and g in this bound.

internal computation must be a sorting of the internal-memory elements.

The only previous sorting algorithms we are aware of that were designed with the BSP model in mind are recent methods of Adler, Byers, and Karp [1] and Gerbessiotis and Valiant [20]. The method of Adler, Byers, and Karp runs in a combined time that is $O(\frac{ng\log n}{p} + pg + gL)$, provided $p \leq n^{1-\delta}$ for some constant $0 < \delta < 1$. They do not define their algorithm for larger values of p, but they do give a slightly better implementation of their method in the LogP model so as to achieve a running time of $O(\frac{ng\log n}{p} + pg + L)$ for p similarly bounded. Gerbessiotis and Valiant give several randomized methods,[3] the best of which runs with a combined time of $O(\frac{n\log n}{p} + gp^{\epsilon} + gn/p + L)$, with high probability, for any constant $0 < \epsilon < 1$, provided $p \leq n^{1-\delta}$, where δ is a small constant depending upon ϵ.

1.3. Our results. Given a set S of n items distributed evenly across p processors in a weak-CREW BSP computer we show how S can be sorted in $O(\log n/\log(h+1))$ communication rounds and $O((n\log n)/p)$ internal computation time, for $h = \Theta(n/p)$. The method is fairly simple and the constant factors in the running time are fairly small. Moreover, we also show how to extend our result to the EREW BSP model while achieving the same asymptotic bounds on the number of communication rounds and internal computation time. Our bounds on internal computation time are optimal for any comparison-based parallel algorithm. In addition, we achieve a deterministic combined running time that is $O(\frac{n\log n}{p} + (L + gn/p)(\log n/\log(n/p)))$, which is valid for all values of p and improves the best bounds of Adler, Byers, and Karp [1] and Gerbessiotis and Valiant [20] even when $p \leq n^{1-\delta}$ for some constant $0 < \delta < 1$, in which case our method sorts in a constant number of communication rounds. In fact, if $p^3 \leq n$, then our method essentially amounts to a sample sort (with regular sampling). If $p = \Theta(n)$, then our method amounts to a pipelined parallel mergesort, achieving the same asymptotic performance as the fine-grained algorithms of Cole [11] and Goodrich and Kosaraju [22]. Thus, our method provides a sorting method that is fully scalable over all values of p while achieving an optimal internal computation time over this entire range.

Indeed, we show that our bounds on the number of communication rounds needed to sort n elements on a BSP computer are also worst-case optimal for this entire range of values of p. We establish this by showing that simply computing the "or" of n bits distributed evenly across $\Theta(n/h)$ processors requires $\Omega(\log n/\log(h+1))$ number of communication rounds, where each processor can send and receive h messages in a CREW BSP computer. This lower bound holds even if the number of additional processors and the number of additional memory cells per processor are unbounded. Since this lower bound is independent of the total number of processors and amount of memory in the multicomputer, it joins lower bounds of Mansour, Nisan, and Vishkin [34] and Adler, Byers, and Karp [1] in giving further evidence that the prime bottleneck in parallel computing is communication, not the number of processors nor the memory size.

2. A weak-CREW BSP sorting algorithm.

Let S be a set of n items distributed evenly in a p-processor weak-CREW BSP computer. We sort the elements of S using a d-way parallel mergesort, pipelined in a way analogous to the binary parallel mergesort procedures of Cole [11] and Goodrich and Kosaraju [22].

[3]They also give an alternate BSP sorting method that has a combined running time of $O([(n/p)\log^{a+1} p + L\log^2 p + g\log^{a+2} p + g(n/p)\log p]/\log\log p)$, with high probability, provided $p < n/\log^{a+1} p$.

Specifically, we choose $d = \max\{\lceil\sqrt{n/p}\rceil, 2\}$ and let T be a d-way rooted, complete, balanced tree such that each leaf is associated with a subset $S_i \subseteq S$ of size at least $\lfloor n/p \rfloor$ and at most $\lceil n/p \rceil$. For each node v in T define $U(v)$ to be the sorted list of elements stored at descendants of v in T, where we define v to be a descendant of itself if it is a leaf. Note that if $\{w_1, w_2, \ldots, w_d\}$ denote the children of a node v in T, then $U(v) = U(w_1) \cup U(w_2) \cup \cdots \cup U(w_d)$. Our goal, then, is to construct $U(root(T))$. We may assume, without loss of generality, that the elements are distinct, for otherwise we can break ties using the original positions of the elements of S.

We perform this construction in a bottom-up pipelined way. In particular, we perform a series of *stages*, where in stage t we construct a list $U_t(v) \subseteq U(v)$ for each node v that we identify as being *active*. A node is *full* in stage t if $U_t(v) = U(v)$, and a node is *active* if $U_t(v) \neq \emptyset$ and v was not full in stage $t-3$. Likewise, we say that a list A stored at a node v in T is *full* if $A = U(v)$. Initially, each leaf of T is full and active, whereas each internal node is initially inactive.

We say that a list B is a *k-sample* of a list A if B consists of every kth element of A. For each active node v in T we define a sample $L_t(v)$ as follows:

- If v is not full, then $L_t(v)$ is a d^2-sample of $U_t(v)$.
- If v first became full in stage t, then we define $L_t(v)$ to be a d^2-sample of $U_t(v) = U(v)$; we define $L_{t+1}(v)$ to be a d-sample of $U_t(v)$; and we define $L_{t+2}(v) = U(v)$ (i.e., $L_{t+2}(v)$ is full).

We then define

$$U_t(v) = L_{t-1}(w_1) \cup L_{t-1}(w_2) \cup \cdots \cup L_{t-1}(w_d),$$

where, again, $\{w_1, w_2, \ldots, w_d\}$ denote the children of node v in T. Note that by our definition of $L_t(v)$, if a node v becomes full in stage t, then v's parent becomes full in stage $t+3$. Thus, assuming we can implement each stage with a constant number of communication rounds using the p processors, we will be able to sort the elements of S, by constructing $U(root(T))$, in just $O(\log_d n) = O\left(\frac{\log n}{\log(h+1)}\right)$ communication rounds, for $h = \Theta(n/p)$. Before we give the details for implementing each stage in our algorithm, however, we establish the following bounds.

LEMMA 2.1. *If at most k elements of $U_t(v)$ are in an interval $[a, b]$, then at most $dk + 2d^2$ elements of $U_{t+1}(v)$ are in $[a, b]$.*

Proof. Our proof is by induction on t. Suppose there are k elements of $U_t(v)$ in an interval $[a, b]$. Since the claim is clearly true if v is full (which is the base case), let us suppose further that v is not full. Then $U_t(v) = L_{t-1}(w_1) \cup L_{t-1}(w_2) \cup \cdots L_{t-1}(w_d)$, where $\{w_1, w_2, \ldots, w_d\}$ denote the children of v. Let j_i denote the number of elements of $L_{t-1}(w_i)$ in the interval $[a, b]$ and let k_i denote the number of elements of $U_{t-1}(w_i)$ in the interval $[a, b]$. Likewise, let j'_i denote the number of elements of $L_t(w_i)$ in the interval $[a, b]$ and let k'_i denote the number of elements of $U_t(w_i)$ in the interval $[a, b]$. Finally, let k' denote the number of elements of $U_{t+1}(v)$ in the interval $[a, b]$. Then

$$k = \sum_{i=1}^{d} j_i$$

and

$$\left\lfloor \frac{k_i}{d^2} \right\rfloor \leq j_i \leq \left\lceil \frac{k_i}{d^2} \right\rceil$$

by our definition of $L_t(w_i)$, with similar relationships holding, respectively, for k' and the k'_i values. Therefore,

$$\begin{aligned}
k' &= \sum_{i=1}^{d} j'_i \\
&\le \sum_{i=1}^{d} \left\lceil \frac{k'_i}{d^2} \right\rceil \\
&\le \sum_{i=1}^{d} \left\lceil \frac{dk_i + 2d^2}{d^2} \right\rceil \qquad \text{(by our induction hypothesis)} \\
&= \sum_{i=1}^{d} \left(\left\lceil \frac{k_i}{d} \right\rceil + 2 \right) \\
&\le 2d + \sum_{i=1}^{d} \left\lceil \frac{d^2(j_i+1)}{d} \right\rceil \\
&= 2d + \sum_{i=1}^{d} d(j_i+1) \\
&= 2d + d^2 + d\sum_{i=1}^{d} j_i \\
&= 2d + d^2 + dk \\
&\le dk + 2d^2,
\end{aligned}$$

provided $d \ge 2$, which will always be the case for our algorithm. ∎

Intuitively, this lemma says that $U_{t+1}(v)$ will not be wildly different from $U_t(v)$. Similarly, we have the following corollary that relates $L_{t+1}(v)$ and $L_t(v)$.

COROLLARY 2.2. *If at most k elements of $L_t(v)$ are in an interval $[a, b]$, then at most $d(k + 1) + 2$ elements of $L_{t+1}(v)$ are in $[a, b]$.*

Proof. Suppose there are k elements of $L_t(v)$ in an interval $[a, b]$. The corollary is immediately true if v is full, so let us suppose that v is not full. Then there are at most $d^2(k + 1)$ elements of $U_t(v)$ in $[a, b]$; hence, by Lemma 2.1, there are at most $d^3(k + 1) + 2d^2$ elements of $U_{t+1}(v)$ in $[a, b]$. Thus, there are at most $d(k + 1) + 2$ elements of $L_{t+1}(v)$ in $[a, b]$. ∎

Having given this important lemma and its corollary, let us now turn to the details of implementing each stage in our pipelined procedure using just a constant number of communication rounds.

2.1. Implementing each stage using a constant number of communication rounds.

We say that a list A is *ranked* [11, 22] into a list B if, for each element $a \in A$, we know the rank of a's predecessor in B (based upon the ordering of elements in $A \cup B$). If A is ranked in B and B is ranked in A, then A and B are *cross ranked.* The generic situation at the end of any stage t is that we have the following conditions satisfied at each node v in T.

Induction invariants.

1. $L_t(v)$ is ranked into $L_{t-1}(v)$.
2. If v is not full, then $L_{t-1}(w_i)$ is ranked into $U_t(v)$, for each child w_i of v in T.

3. $L_t(v)$ is ranked into $U_t(v)$.

We maintain copies of the lists $L_{t-1}(v)$, $L_t(v)$, $U_{t-1}(v)$, and $U_t(v)$ for each active node v in T, and we do not maintain any other lists during the computation. As we shall show, this will allow us to implement the entire computation efficiently using just p processors. In order to implement each stage in our computation using just $O(1)$ communication rounds we also maintain the following important load-balancing invariant at each node v in T.

Load-balancing invariant.

- If a list A is not full, then A is partitioned into contiguous subarrays of size d each, with each subarray stored on a different processor.
- If a list A is full, then A is partitioned into contiguous subarrays of size d^2 each, with each subarray stored on a different processor.

We assume that the names of the nodes of v in T and the four lists stored at each node v are defined so that given an index, i, into one of these lists, A, one can determine the processor holding $A[i]$ as a local computation (not needing a communication step). For example, we could number the nodes in T using the level-order labeling (e.g., see Goodrich and Tamassia [23]) and number each contiguous subarray of an array A consecutively. Since the arrays stored at the same level in T are all within one element of being the same size, such a numbering scheme would allow us to compute in $O(1)$ time the processor number holding any subarray $A[i..j]$ of an array A simply by knowing the index for $A[i..j]$ and the level-order number of the node holding $A[i..j]$.

Given that the induction and load-balancing invariants are satisfied for each node v in T, we can construct $U_{t+1}(v)$ at each active node, with the above invariants satisfied for it, as follows.

Computation for stage $t + 1$.

1. For each element a in $L_t(w_i)$, let $b(a)$ and $c(a)$, respectively, be the predecessor and successor of a in $L_{t-1}(w_i)$. We can determine $b(a)$ and $c(a)$ in $O(1)$ communication rounds, for each such a, since $L_t(w_i)$ is ranked in $L_{t-1}(w_i)$ by induction invariant 1. In fact, if $L_t(w_i) = U(w_i)$, then this is essentially a local computation. Moreover, by our load-balancing invariant and Corollary 2.2, even in the general case, each processor (storing a portion of some $L_{t-1}(w_i)$) will receive (and then send) at most $d(d+1)+2 = \Theta(h)$ messages to implement this step.
2. Determine the location (rank) of $b(a)$ and $c(a)$ in $U_t(v)$. This can also be easily implemented with a $O(1)$ communication rounds, as in the previous step.
3. Broadcast a (and its rank in $L_t(w_i)$) to all processors holding elements of $U_t(v)$ between $b(a)$ and $c(a)$. Note that this step may require usage of the weak-CREW capability, since there may be many processors holding such elements and, if so, these processors will be consecutively numbered. By our load-balancing invariant and Lemma 2.1 we can guarantee that each processor will receive at most $3d^2 + d = \Theta(h)$ messages to implement this step (each processor receives at least one element from each child of v plus as many elements as fall in its interval of $U_t(v)$); hence, this step can be implemented in $O(1)$ communication rounds.
4. Each processor assigned to a contiguous portion $[e, f)$ of $U_t(v)$ receives elements sent in the previous round and merges them via a simple d-way merge-sort procedure to form a sublist of $U_{t+1}(v)$ of size $O(d^2) = O(h)$. It is im-

portant to observe that the processor for $[e, f)$ receives at least one element from each child of v so as to include all the elements that may intersect the interval $[e, f)$, even if none actually fall inside $[e, f)$. This allows us to accurately compute the rank of each element in $U_{t+1}(v)$ locally; hence, it gives us $U_t(v)$ cross ranked with $U_{t+1}(v)$. Moreover, this step can be accomplished in $O(1)$ communication rounds and $O(d^2 \log d) = O((n/p)\log(n/p))$ internal computation time.

5. For each element a in $U_{t+1}(v)$ send a message to the processor holding $a \in L_t(w_i)$ informing that copy of a of its rank in $U_{t+1}(v)$. This step can easily be accomplished in $O(1)$ communication rounds and gives us induction invariant 2.
6. Determine the sample $L_{t+1}(v)$ and rank it into $U_{t+1}(v)$, giving us induction invariant 3. Also, use the cross ranking of $U_{t+1}(v)$ and $U_t(v)$ to rank $L_{t+1}(v)$ into $L_t(v)$, giving us induction invariant 1. This step can easily be accomplished in $O(1)$ communication rounds.
7. Finally, partition the four lists stored at each node v so as to satisfy the load-balancing invariant. That is, we partition each list in T into the appropriately sized sublists and ship sublists to processors according to our numbering scheme. Assuming the total size of all the nonfull lists in T is $O(n/d)$, this can easily be performed in $O(1)$ communication rounds using $p = \Theta(n/d^2)$ processors.

Therefore, given the above assumption regarding the total size of all the lists, in a constant number of communication rounds and an internal computation time that is $O((n/p)\log(n/p))$ we can build the set $U_{t+1}(v)$ and establish the induction and load-balancing invariants so as to repeat this procedure in stage $t + 2$.

Let us therefore analyze the total size of all the lists stored at nodes in T. Clearly, the size of all the full lists in T is $O(n)$. Moreover, each such list contributes at most $1/d$ of its elements to the next higher level in T, and from then on up T each list on a level l contributes at most $1/d^2$ of its elements to lists on the next higher level in T. Thus, the total size of all nonfull $U_{t-1}(v)$ or $U_t(v)$ lists forms a geometric series that sums to be $O(n/d)$, which is what we require. In addition, any sample $L_t(v)$ or $L_{t-1}(v)$ that is not full can contain at most $1/d$ of the elements of $U(v)$; hence, the total space needed for all these lists is also $O(n/d)$. This establishes the following.

THEOREM 2.3. *Given a set S of n items stored $O(n/p)$ per processor on a p-processor weak-CREW BSP computer, one can sort S in $O(\log n/\log(h+1))$ communication steps and $O((n \log n)/p)$ internal computation time, where $h = \Theta(n/p)$.*

In achieving this result we exploited the broadcast capability of the weak-CREW BSP model (in step 3). In the next section we show how to match the asymptotic performance of Theorem 2.3 without using such a capability.

3. An EREW BSP sorting algorithm.

Suppose we are now given a set S of n items, which are distributed evenly across the p processors of an EREW BSP computer. Our goal is to sort S in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ internal computation time without using any broadcasts, for $h = \Theta(n/p)$. We achieve this result using a cascading method similar to one used by Cole [11], which itself is similar to the general fractional cascading technique of Chazelle and Guibas [9].

Let T be a complete rooted d-way tree with each of its leaves associated with a sublist $S_i \subset S$ of size at most $\lceil n/p \rceil$, where $d = \max\{\lceil (n/p)^{1/7} \rceil, 2\}$ (the reason for this choice will become apparent in the analysis). Our method proceeds in a series of stages, as in the weak-CREW BSP algorithm, with us constructing the set $U_t(v)$ in each stage, as before:

$$U_t(v) = \bigcup_{i=1}^{d} L_{t-1}(w_i),$$

where each $L_t(v)$ list is defined to be a sample of $U_t(v)$ as in our weak-CREW algorithm.

In order to perform this construction so as to avoid broadcasts, however, we will accomplish this by constructing a larger, augmented list, $A_t(v)$, such that $U_t(v) \subseteq A_t(v)$. We also define a list $D_t(v)$ to be a d^2-sample of $A_t(v)$. For each active node v, with parent u and children $w_1, w_2, \ldots, w_d$, we then define

$$A_t(v) = D_{t-1}(u) \cup \bigcup_{i=1}^{d} L_{t-1}(w_i),$$

i.e., $A_t(v) = D_{t-1}(u) \cup U_t(v)$. Intuitively, the D_t lists communicate information "down" the tree T in a way that allows us to avoid broadcasts. Indeed, once a copy of an element begins to traverse down the tree, it will never again traverse up (since the D lists are sent only to children).

Still, although we are assuming, without loss of generality, that the elements of S are distinct, this definition may create duplicate entries of an element in the same list, with some traversing down and at most one traversing up. We resolve any ambiguities this may create by breaking comparison ties based upon an upward-traversing element always being greater than any downward-traversing element, and any comparison between downward-traversing elements is resolved based upon the level in T where the elements first began traversing down (where level numbers increase as one traverses down T).

The goal of each stage t in the computation, then, is to construct $A_t(v)$ and $U_t(v)$, together with their respective samples $D_t(v)$ and $L_t(v)$. In order to prove that each stage of our algorithm can indeed be performed in a constant number of communication rounds on an EREW BSP computer we must establish the following bounds.

LEMMA 3.1. *If at most k elements of $A_t(v)$ are in an interval $[a, b]$, then at most $(d+1)k + 2(d+1)^2$ elements of $A_{t+1}(v)$ are in $[a, b]$.*

Proof. The proof is essentially the same as that for Lemma 2.1 except that the expansion coefficient is now $d + 1$, instead of d, for each nonfull active node now receives elements from all $d + 1$ of its neighbors in T, not just its children. ∎

This immediately implies the following.

COROLLARY 3.2. *If at most k elements of $D_t(v)$ are in an interval $[a, b]$, then at most $(d+1)(k+1) + 3$ elements of $D_{t+1}(v)$ are in $[a, b]$.*

Proof. Suppose there are k elements of $D_t(v)$ in an interval $[a, b]$. Then there are at most $d^2(k + 1)$ elements of $A_t(v)$ in $[a, b]$; hence, by Lemma 3.1, there are at most $(d + 1)d^2(k + 1) + 2(d + 1)^2$ elements of $A_{t+1}(v)$ in $[a, b]$. Thus, there are at most $(d + 1)(k + 1) + 3$ elements of $D_{t+1}(v)$ in $[a, b]$. ∎

In addition, we can also show the following.

LEMMA 3.3. *For any two consecutive elements b and c in $A_t(v)$ let b' and c', respectively, be the predecessor of b and the successor of c in $A_t(u)$, where u is the parent of v in T. There are at most $(d+1)(d^2+1) + 2(d+1)^2 + 2$ elements of $A_t(u)$ in the interval $[b', c']$.*

Proof. By construction, $A_t(v)$ contains $D_{t-1}(u)$ as a subset, and $D_{t-1}(u)$ is a d^2-sample of $A_{t-1}(u)$. Thus, there can be at most d^2+1 elements of $A_{t-1}(u)$ in the interval $[b,c]$. By Lemma 3.1, then, there will be at most $(d+1)(d^2+1)+2(d+1)^2$ elements of $A_t(u)$ in the interval $[b,c]$; hence, there will be at most $(d+1)(d^2+1)+2(d+1)^2+2$ elements of $A_t(u)$ in the interval $[b',c']$. □

Finally, we have the following.

LEMMA 3.4. *For any two consecutive elements b and c in $D_{t-1}(u)$ there are at most $(d+1)^2(d^4+5)$ elements of $A_t(v)$ in the interval $[b,c]$, where u is the parent of v in T.*

Proof. By construction, $D_{t-1}(u)$ is a d^2-sample of $A_{t-1}(u)$, which in turn contains $L_{t-2}(v)$ as a subset. Thus, there are at most d^2+1 elements of $L_{t-2}(v)$ in the interval $[b,c]$. This implies that there are at most d^4+1 elements of $A_{t-2}(v)$ in the interval $[b,c]$. The lemma follows then by two applications of Lemma 3.1. □

As will become apparent in our algorithm description, these bounds are all crucial for establishing that our algorithm runs in the EREW BSP model using a constant number of communication rounds per stage. In order to perform the computation for stage $t+1$ using a constant number of communication rounds we assume that we maintain the following induction invariants for each active node v in T.

Induction invariants.
1. $A_t(v)$ is ranked into $U_t(v)$.
2. $A_t(v)$ and $D_{t-1}(u)$ are cross ranked, where u is the parent of v.
3. $A_{t-1}(v)$ is ranked into $A_t(v)$.
4. $D_t(v)$ is ranked in $D_{t-1}(v)$.

We also maintain a load-balancing invariant, similar to the one we used in our weak-CREW BSP algorithm, except that we now define a list A stored at a node v to be *full* if $A \supseteq U(v)$.

Load-balancing invariant.
- If a list A is not full, then A is partitioned into contiguous subarrays of size d^6 each, with each subarray stored on a different processor.
- If a list A is full, then A is partitioned into contiguous subarrays of size d^7 each, with each subarray stored on a different processor.

Given that the induction and load-balancing invariants hold after the completion of stage t, our method for performing stage $t+1$ is as follows.

Computation for stage $t+1$. For each child w_i of v we perform the following computation:
1. For each element a in $A_t(w_i)$ use the ranking of $A_t(w_i)$ in $U_t(w_i)$ to determine if a is also in $L_t(w_i)$ (together with its rank in $L_t(w_i)$ if so). No communication is necessary for this step, given induction invariant 1.
2. For each such element a in $L_t(w_i)$ use the ranking of $A_t(w_i)$ in $D_{t-1}(v)$ to determine the ranks of the predecessor, $b(a)$, of a and successor, $c(a)$, of a in $D_{t-1}(v)$. No communication is necessary for this step either, given induction invariant 2.
3. For each a in $L_t(w_i)$, use the ranks of the processor(s) for $b(a)$ and $c(a)$ in $D_{t-1}(v)$ to determine the respective ranks of $b(a)$ $c(a)$ in $A_{t-1}(v)$. No communication is necessary for this step.
4. For each a in $L_t(w_i)$, request that the processor(s) for $b(a)$ and $c(a)$ in $A_{t-1}(v)$ send(s) the processor for a the name of predecessor, $b'(a)$, of $b(a)$ and the name of successor, $c'(a)$, of $b(a)$ in $A_t(v)$, using invariant 3. By Lemma 3.1 and

our load-balancing invariant, each processor will receive and send at most $(d+1)d^6 + 2(d+1)^2 = \Theta(h)$ messages to implement this step.

5. Send a (together with its rank in $L_t(w_i)$) to the processor(s) assigned to elements of $A_t(v)$ between $b'(a)$ and $c'(a)$ to be merged with all other elements of $A_{t+1}(v)$ that fall in this range. As with the previous step, by Lemma 3.1 and our load-balancing invariant each processor will receive at most $(d+1)d^6 + 2(d+1)^2 = \Theta(h)$ messages to implement this step. More importantly, by Lemma 3.3, each processor will send an element a to at most $\lceil((d+1)(d^2+1) + 2(d+1)^2 + 2)/d^6\rceil + 1 = O(1)$ other processors. Thus, no broadcasting is needed in order to implement this step.

At the parent u of v we assume a similar (but simpler) computation is being performed:

1. For each element a in $D_t(u)$, determine the ranks of $b(a)$ and $c(a)$, the respective predecessor and successor of a in $D_{t-1}(u)$. No communication is necessary for this step, given induction invariant 4.
2. Request that the processor(s) for the copies of $b(a)$ and $c(a)$ in $D_{t-1}(u)$ return the ranks of $b(a)$ and $c(a)$ in $A_t(v)$, which are available because of induction invariant 2. By Corollary 3.2 and our load-balancing invariant, this step can be implemented with each processor receiving and sending at most $(d+1)(d^6+1) + 3 = \Theta(h)$ messages.
3. Send a (together with its rank in $D_t(u)$) to the processor(s) assigned to elements of $A_t(v)$ between $b(a)$ and $c(a)$ to be merged with all other elements of $A_{t+1}(v)$ that fall in this range. By Lemma 3.4 each processor will send an element a to at most $\lceil(d+1)^2(d^4+5)/d^6\rceil + 1 = O(1)$ other processors; hence, no broadcasting is needed. Moreover, each processor will send d such copies of a, which, by our load-balancing invariant, implies that a processor will send $O(h)$ messages. Likewise, each processor will receive at most $O(h)$ messages.

Finally, at node v we perform the following computation:

1. For each interval $[e, f)$ of elements of $A_t(v)$ assigned to a single processor, merge all the elements coming from the parent u and children $w_1, w_2, \ldots, w_d$ to form $A_{t+1}(v)$. Such a processor will receive at least one element from each node adjacent to v, plus as many elements of $A_{t+1}(v)$ as fall in $[e, f)$, for a total of at most $d + 1 + (d+1)d^6 + 2(d+1)^2 = \Theta(h)$. This mergesort computation amounts to a $(d+1)$-way mergesort and can easily be implemented in $O(d^7 \log(d+1)) = O((n/p)\log(n/p))$ internal steps.
2. Likewise, for each interval $[e, f)$ of elements of $A_t(v)$ assigned to a single processor, merge all the elements coming just from v's children $w_1, w_2, \ldots, w_d$ to form $U_{t+1}(v)$ (and $A_{t+1}(v)$ ranked in $U_{t+1}(v)$, which gives us induction invariant 1).
3. Use the rank information derived from the previous two steps to rank $A_{t+1}(v)$ in $D_t(u)$, giving us half of induction invariant 2. Also, rank $A_t(v)$ in $A_{t+1}(v)$, giving us invariant 3 and by an additional calculation a ranking of $D_{t+1}(v)$ in $D_t(v)$, which is invariant 4. Finally, send a message to each element a in $D_t(u)$ informing it of its rank in $A_{t+1}(v)$ so as to complete the other half of invariant 2. To implement this step requires that each processor send at most h messages and each processor receive at most $d^6(d) = O(h)$ messages.
4. Finally, repartition the lists at each node v so as to satisfy the load-balancing invariant. Assuming that the total size of all nonfull lists is $O(n/d)$ and the size of all full lists is $O(n)$, this step can easily be implemented in $O(1)$ communication rounds.

Let us, therefore, analyze the space requirements of this algorithm. The total size of all the $U(v)$ lists on the full level clearly is $O(n)$. Each such list causes at most $\lceil |U(v)|/d \rceil$ elements to be sent to v's parent, u. Now the inclusion of these elements in u causes at most $(d+1)\lceil |U(v)|/d^3 \rceil$ elements to be sent to nodes at distance 1 from u (including v itself). However, once an element starts traversing down the tree T it never is sent up again. We can repeat this argument to establish that the existence of $U(v)$ causes at most $(d+1)^2 \lceil |U(v)|/d^5 \rceil$ elements to be sent to nodes at distance 2 from u, and so on. Thus, the number of all of these elements that originate from u sum to be a geometric series that is $O(n/d)$. Therefore, the total size of all the nonfull lists is $O(n/d)$. Likewise, the total size of all the lists (and hence the lists on the full level) is $O(n)$. This gives us the following theorem.

THEOREM 3.5. *Given a set S of n items stored $O(n/p)$ per processor on a p-processor EREW BSP computer, one can sort S in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ internal computation time, for $h = \Theta(n/p)$.*

This immediately implies the following.

COROLLARY 3.6. *Given a set S of n items stored $O(n/p)$ per processor, one can sort S on an EREW BSP computer with a combined running time that is $O(\frac{n \log n}{p} + (L + gn/p)(\log n/\log(n/p)))$.*

This bound also applies to the LogP model.

4. A lower bound for BSP computations.

In this section we show that our upper bounds on the number of communication rounds needed to sort n numbers on a p-processor BSP computer are optimal. Specifically, we show that $\Omega(\log n/\log(h+1))$ communication steps are needed to compute the "or" of n bits using an arbitrary number of processors in a CREW BSP computer, where h is the number of messages that can be sent and received by a single processor in a single communication round.

Let us begin by formalizing the framework for proving our lower bound. Assume we have a set S of n Boolean values $x_1, x_2, \ldots, x_n$ initially placed in memory locations $m_1, m_2, \ldots, m_n$ with memory cells $m_{(i-1)h+1}, \ldots, m_{ih}$ stored in the local memory of processor p_i, for $i \in \{1, 2, \ldots, \lceil n/h \rceil\}$. This, of course, implies that we have at least $\lceil n/h \rceil$ processors, but for the sake of the lower bound we allow for an arbitrary number of processors. Moreover, we place no upper bound on the amount of additional memory cells that each processor may store internally. The goal of the computation is that after some T steps the "or" of the values in S should be stored in memory location m_1.

Our lower bound proof will be an adaptation of a lower bound proof of Cook, Dwork, and Reischuk [12] for computing the "or" of n bits on a CREW PRAM. In performing this adaptation there are several difficulties that must be overcome. The primary difficulty comes from the fact that each processor in a BSP computer can send h messages in each communication round, rather than just a single value, because this complicates arguments that bound the amount of information processors can communicate by *not* sending messages.

Each processor p_i is assumed initially to be in a starting state, q_1^i, taken from a possibly unbounded set of states. At the beginning of a round t processor p_i is assumed to be in some state q_t^i. A round begins with each processor sending up to h messages, some of which may be (arbitrary) partial broadcasts, and simultaneously receiving up to h messages from other processors. Without loss of generality, each message may be assumed to be the contents of one of the memory cells associated with the sending processor, since we place no constraints on the amount of information that may be stored in a memory cell nor on the number of memory cells that a processor may contain. A processor then enters a new state q^i_{t+1} that depends upon its previous state q^i_t and the values of the messages it has received. A round completes with a processor possibly writing new values to some of its internal memory cells based upon its new state q^i_{t+1}.

Before analyzing the most general situation, let us first prove a lower bound for the *oblivious* case, where the determination of whether a processor p_i will send a message to processor p_j in round t depends upon only the value of p_i and t and not on the input. Of course, the contents of such a message could depend upon the input. For input string $I = (x_1, x_2, \ldots, x_n)$ of Boolean values, let $I(k)$ denote the input string $(x_1, x_2, \ldots, \bar{x}_k, \ldots, x_n)$, where $\bar{x}_k$ denotes the complement of Boolean value x_k. I is a *critical input* for function $f(I)$ if $f(I) \neq f(I(k))$ for all $k \in \{1, 2, \ldots, n\}$. (Note that $I = (0, 0, \ldots, 0)$ is a critical input for the "or" function.) Say that input index k *affects* [12] processor p_i in round t with input I if the state of p_i on input I after round t differs from the state of processor p_i on input $I(k)$ after round t. Likewise, say that input index k affects memory cell m_i in round t with input I if the contents of m_i on input I after round t differs from the contents of m_i on input $I(k)$ after round t.

THEOREM 4.1. *If $f : \{0,1\}^n \to \{0,1\}$ has a critical input, then any oblivious CREW BSP computer that computes f requires $\Omega(\log n / \log(h+1))$ communication rounds.*

Proof. Let $K(p_i, t, I)$ (respectively, $L(m_i, t, I)$) be the set of input indices that affect processor p_i (respectively, memory cell m_i) in round t with input I. Further, let K_t and L_t satisfy the following recurrence equations:

$$K_0 = 0, \tag{4.1}$$

$$L_0 = 1, \tag{4.2}$$

$$K_{t+1} = K_t + hL_t, \tag{4.3}$$

$$L_{t+1} = K_{t+1} + L_t = K_t + (h+1)L_t. \tag{4.4}$$

Note that it suffices to prove that $|K(p_i, t, I)| \leq K_t$ and $|L(p_i, t, I)| \leq L_t$, for K_t and L_t are both at most $[2(h+1)]^t$, and if I is a critical input for f, then every one of the input indices must affect memory cell m_1. That is, if $m_1 = f(I)$, then $|L(m_1, T, I)| = n$, which implies that T is $\Omega(\log n / \log(h+1))$.

We establish the bounds on $|K(p_i, t, I)|$ and $|L(p_i, t, I)|$ by induction on t. First, note that $K(p_i, t, I)$ is empty in round $t = 0$, and $L(m_i, 0, I) = \{i\}$ if $i \in \{1, 2, \ldots, n\}$ and otherwise $L(m_i, 0, I)$ is empty. In round $t+1$ each processor p_i receives the contents of up to h memory locations; hence, $K(p_i, t+1, I) \subseteq K(p_i, t, I) \cup \bigcup_{j \in \mathcal{I}} L(m_j, t, I)$ for some index set $\mathcal{I}$ with $|\mathcal{I}| \leq h$. Thus, $|K(p_i, t+1, I)| \leq K_{t+1}$ by the induction hypothesis. After a processor p_i receives the values of these memory locations it may, at its option, write to any of its internal memory cells based upon its new state. Therefore, for any memory cell m_j internal to processor p_i, $L(m_j, t+1, I) \subseteq K(p_i, t+1, I) \cup L(m_j, t, I)$; hence, by the induction hypothesis and (4.3), $|L(m_j, t+1, I)| \leq L_{t+1}$. $\square$

The main difficulty in generalizing this result to nonoblivious computations is that in the non oblivious case a processor p_i can receive information from a processor p_j by p_j *not* sending a message to p_i. Still, as we show in the next theorem, this ability cannot alter the asymptotic performance of a CREW BSP computer by more than a constant factor for computing the value of a function with a critical input.

THEOREM 4.2. *If $f : \{0,1\}^n \to \{0,1\}$ has a critical input, then any CREW BSP computer that computes f requires $\Omega(\log n / \log(h+1))$ communication rounds.*

Proof. Let $K(p_i, t, I)$ and $L(m_i, t, I)$ be as in the proof of Theorem 4.1. But now let K_t and L_t be defined by the following recurrence relations:

$$K_0 = 0, \tag{4.5}$$

$$L_0 = 1, \tag{4.6}$$

$$K_{t+1} = (6h+1)K_t + hL_t, \tag{4.7}$$

$$L_{t+1} = K_{t+1} + L_t = (6h+1)K_t + (h+1)L_t. \tag{4.8}$$

As in the previous proof, it suffices to show that $|K(p_i, t, I)| \le K_t$ and $|L(m_i, t, I)| \le L_t$, for K_t and L_t are both at most $[7(h+1)]^t$.

We establish these bounds on $|K(p_i, t, I)|$ and $|L(p_i, t, I)|$ by induction on t. First, note that $K(p_i, t, I)$ is empty in round $t = 0$, and $L(m_i, 0, I) = \{i\}$ if $i \in \{1, 2, \ldots, n\}$ and otherwise $L(m_i, 0, I)$ is empty. At the beginning of round t a processor p_i receives the contents of at most h memory locations, and it also receives information by noting that some processors did not send p_i a message. Still, after it incorporates this information into its new state q^i_{t+1}, it optionally writes to its local memory, as in the previous proof. Thus, if we can establish (4.7), then (4.8) immediately follows.

Say that input index k *possibly causes* a processor p_j to send a message to processor p_i in round t with I if p_j sends a message to processor p_i in round t on input $I(k)$. Using this notion we bound $K(p_i, t+1, I)$ as a subset of

$$K(p_i, t, I) \cup \bigcup_{j \in \mathcal{I}} L(m_j, t, I) \cup Y(p_i, t, I),$$

for some index set $\mathcal{I}$ with $|\mathcal{I}| \le h$, where $Y(p_i, t, I)$ denotes the set of all indices k that possibly cause some processor p_j to send a message to p_i with I. Thus, we must bound $r = |Y(p_i, t, I)|$. So, let $Y = Y(p_i, t, I) = \{k_1, k_2, \ldots, k_r\}$ be the set of indices that possibly cause a processor to send a message to p_i with I. Further, for each k_j, let $p(k_j)$ denote a specific processor that would send p_i a message in round t on $I(k_j)$, and let P denote the set of all such processors, i.e., define $P = \{p(k_j)\colon k_j \in Y\}$. Note that if $r \le hK_t$, then we have established (4.7), so for the remainder of this proof let us assume that $r > hK_t$. We will show that if this is the case, then $r \le 6hK_t$.

As done by Cook, Dwork, and Reischuk [12], we employ a combinatorial graph argument to derive a bound on $r = |Y|$. Consider a bipartite graph G whose two node sets are Y and P. Let there be an edge between k_j in Y and $p(k_l)$ in P if k_j affects $p(k_l)$ in round t with $I(k_l)$. Let e denote the number of edges in G. The degree of any node $p(k_j)$ is at most $|K(p(k_j), t, I(k_j))|$, which, by our induction hypothesis, is bounded by K_t. Thus, $e \le |P|K_t \le rK_t$.

We can also derive a lower bound on e. To this end let us define a second graph H, which is defined on the elements in Y. Say that two indices k_j and k_l in Y are adjacent in H if there is, in G, an edge from k_j to $p(k_l)$ or an edge from k_l to $p(k_j)$. That is, for each edge (k_j, k_l) in H there is at least one corresponding edge in G, indicating that either k_j affects $p(k_l)$ with $I(k_l)$ or k_l affects $p(k_j)$ with $I(k_j)$. Say that a subset $Y' \subseteq Y$ is *processor-disjoint* if, for any k_j and $k_{j'}$ in Y', $p(k_j) \ne p(k_{j'})$.

Claim A. If $Y' \subseteq Y$ is a processor-disjoint independent set in H, then $|Y'| \le h$.

Proof of Claim A. Let $Y' = \{k_{j_1}, k_{j_2}, \ldots, k_{j_m}\}$. Suppose, for the sake of a contradiction, that $m > h$. Since Y' is an independent set in H, then no index k_j in Y' affects the processor associated with any other index k_l in Y' (i.e., $p(k_l)$). Thus, since Y' is processor-disjoint, on input $I(k_{j_1})(k_{j_2}) \cdots (k_{j_m})$ there would be more than h different messages sent to processor p_i. However, this would contradict the communication bandwidth assumptions of the BSP model. $\square$

Claim B. There are at least $r/2$ nodes in H with degree at least $r/2h - K_t$.

Proof of Claim B. Suppose, for the sake of contradiction, that there are at least $r/2$ nodes in H with degree less than $r/2h - K_t$. To reach a contradiction we will construct a processor-disjoint independent set in H of size more than h. We begin by placing the nodes of H into equivalence classes such that the nodes in the same class are all associated with the same processor, i.e., k_j and k_l are in the same class if and only if $p(k_j) = p(k_l)$. Note that there are at most K_t nodes in any class. Now consider a simple greedy algorithm for constructing a processor-disjoint independent Y' set in H:

1. Go to any node v in H of degree less than $r/2h - K_t$ and place v into Y'.
2. Having placed v into Y', delete from H all the nodes in the same class as v as well as all nodes in H that are adjacent to v.
3. If there are nodes left in H, repeat the above two steps.

This, of course, removes less than $r/2h - K_t + K_t = r/2h$ nodes in each iteration. Thus, we can repeat this process for strictly more than

$$\frac{r/2}{r/2h} = h$$

iterations. By construction, there are no edges in H between any of the nodes in Y', and each node in Y' is in a different equivalence class. However, this implies a processor-disjoint independent set $Y' \subseteq Y$ in H such that $|Y'| > h$, which contradicts Claim A. $\square$

Claim B implies that there at least $(r/2)(r/2h - K_t)$ edges in H; hence there are at least this many edges in G. Therefore, $(r/2)(r/2h - K_t) \leq rK_t$. Noting that this implies $r \leq 6hK_t$ completes the proof of the theorem. $\square$

5. Conclusion. We have studied the power and limitations of parallel computing with particular attention paid to the importance of communication in parallel algorithm design, which is an issue gaining prominence in both theory and practice. We gave BSP algorithms for sorting that are optimal in terms of both the internal computation times and the number of communication rounds. Admittedly, the algorithm we derived for the EREW BSP model is considerably more complicated than that we derived for the weak-CREW BSP model. Thus, it is not actually clear which one would be more efficient in practice. A weak-CREW BSP computer would be easier to program but would require the switching hardware in the network to be more sophisticated than what would be required by our EREW BSP algorithm.

Acknowledgments. We would like to thank Bob Cypher, Faith Fich, S. Rao Kosaraju, Greg Plaxton, Mikael Sundström, and Les Valiant for several helpful conversations and e-mail exchanges regarding topics related to this paper.

REFERENCES

[1] M. Adler, J. W. Byers, and R. M. Karp, *Parallel sorting with limited bandwidth*, in Proceedings, Seventh ACM Symposium on Parallel Algorithms and Architectures, 1995, pp. 129–136.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, *Communication complexity of PRAMs*, Theoret. Comput. Sci., 71 (1990), pp. 3–28.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, *Sorting in* $c \log n$ *parallel steps*, Combinatorica, 3 (1983), pp. 1–19.

[4] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, New York, 1985.

[5] K. E. Batcher, *Sorting networks and their applications*, in Proceedings, 1968 Spring Joint Computer Conference, Reston, VA, 1968, AFIPS Press, pp. 307–314.

[6] G. Bilardi and F. P. Preparata, *Lower bounds to processor-time tradeoffs under bounded-speed message propagation*, in Proceedings, Fourth International Workshop on Algorithms and Data Structures (WADS), LNCS 955, Springer-Verlag, Berlin, New York, 1995, pp. 1–12.

[7] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon, *A taxonomy of parallel sorting*, ACM Comput. Surveys, 16 (1984), pp. 287–318.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, *A comparison of sorting algorithms for the connection machine CM-*2, in Proceedings, Third ACM Symposium on Parallel Algorithms and Architectures, 1991, pp. 3–16.

[9] B. Chazelle and L. J. Guibas, *Fractional cascading:* I. *A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[10] V. Chvátal, *Lecture notes on the new AKS sorting network*, Report DCS-TR-294, Computer Science Department, Rutgers University, New Brunswick, NJ, 1992; also available online from ftp://athos.rutgers.edu/pub/technical-reports/dcs-tr-294.ps.Z.

[11] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[12] S. A. Cook, C. Dwork, and R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[13] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a realistic model of parallel computation*, in Proceedings, Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993, pp. 1–12.

[14] R. Cypher and J. L. C. Sanz, *Cubesort: A parallel algorithm for sorting N data items with S-sorters*, J. Algorithms, 13 (1992), pp. 211–234.

[15] R. E. Cypher and C. G. Plaxton, *Deterministic sorting in nearly logarithmic time on the hypercube and related computers*, J. Comput. System Sci., 47 (1993), pp. 501–548.

[16] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar, *A randomized parallel 3D convex hull algorithm for course grained multicomputers*, in Proceedings, Seventh ACM Symposium on Parallel Algorithms and Architectures, 1995, pp. 27–33.

[17] F. Dehne, A. Fabri, and A. Rau-Chaplin, *Scalable parallel geometric algorithms for coarse grained multicomputers*, in Proceedings, Ninth Annual ACM Symposium Comput. Geom., 1993, pp. 298–307.

[18] R. S. Francis and L. J. H. Pannan, *A parallel partition for enhanced parallel quicksort*, Parallel Comput. 18 (1992), pp. 543–550.

[19] W. D. Frazer and A. C. McKellar, *Samplesort: A sampling approach to minimal storage tree sorting*, J. ACM, 17 (1970), pp. 496–507.

[20] A. V. Gerbessiotis and L. G. Valiant, *Direct bulk-synchronous parallel algorithms*, J. Parallel Distrib. Comput., 22 (1994), pp. 251–267.

[21] P. B. Gibbons, *A more practical PRAM model*, in Proceedings, First ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 158–168.

[22] M. T. Goodrich and S. R. Kosaraju, *Sorting on a parallel pointer machine with applications to set expression evaluation*, J. ACM, 43 (1996), pp. 333–356.

[23] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley, New York, 1998.

[24] W. L. Hightower, J. F. Prins, and J. H. Reif, *Implementation of randomized sorting on large parallel machines*, in Proceedings, Fourth ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 158–167.

[25] J. S. Huang and Y. C. Chow, *Parallel sorting and data partitioning by sampling*, in Proceedings, IEEE Seventh International Computer Software and Applications Conference, 1983, pp. 627–631.

[26] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[27] R. Karp, M. Luby, and F. Meyer auf der Heide, *Efficient pram simulation on distributed machines*, in Proceedings, 24th ACM Symposium on Theory of Computing, 1992, pp. 318–326.

[28] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared memory machines*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier/The MIT Press, Amsterdam, 1990, pp. 869–941.

[29] R. M. Karp, A. Sahay, E. Santos, and K. E. Schauser, *Optimal broadcast and summation in the LogP model*, in Proceedings, Fifth ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 142–153.

[30] C. Kruskal, L. Rudolph, and M. Snir, *A complexity theory of efficient parallel algorithms*, Theoret. Comput. Sci., 71 (1990), pp. 95–132.

[31] F. T. Leighton, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput., C-34 (1985), pp. 344–354.

[32] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[33] H. Li and K. C. Sevcik, *Parallel sorting by overpartitioning*, in Proceedings, Sixth ACM Symposium on Parallel Algorithms and Architectures, 1994, pp. 46–56.

[34] Y. Mansour, N. Nisan, and U. Vishkin, *Trade-offs between communication throughput and parallel time*, in Proceedings, 26th ACM Symposium on Theory of Computing (STOC), 1994, pp. 372–381.

[35] K. Mehlhorn and U. Vishkin, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories*, Acta Informatica, 9 (1984), pp. 29–59.

[36] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy, *Parallel algorithm design on the WPRAM model*, Technical Report 94.24, School of Computer Science, University of Leeds, Leeds, UK, 1994.

[37] D. Nassimi and S. Sahni, *Parallel permutation and sorting algorithms and a new generalized connection network*, J. ACM, 29 (1982), pp. 642–667.

[38] C. Papadimitriou and M. Yannakakis, *Towards an architecture-independent analysis of parallel algorithms*, Proceedings, 20th ACM Symposium on Theory of Composition, (STOC), 1988, pp. 510–513.

[39] M. Paterson, *Improved sorting networks with $o(\log n)$ depth*, Algorithmica, 5 (1990), pp. 75–92.

[40] C. G. Plaxton, *Efficient Computation on Sparse Interconnection Networks*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1989.

[41] C. G. Plaxton, *private communication.*

[42] M. J. Quinn, *Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge*, BIT, 29 (1989), pp. 239–250.

[43] J. H. Reif, *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, CA, 1993.

[44] J. H. Reif and L. G. Valiant, *A logarithmic time sort for linear size networks*, J. ACM, 34 (1987), pp. 60–76.

[45] H. Shi and J. Schaeffer, *Parallel sorting by regular sampling*, J. Parallel Distrib. Comput., 14 (1992), pp. 362–372.

[46] L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, 33 (1990), pp. 103–111.

[47] L. G. Valiant, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier/The MIT Press, Amsterdam, 1990, pp. 943–972.

[48] Y. Won and S. Sahni, *A balanced bin sort for hypercube multicomputers*, J. Supercomput., 2 (1988), pp. 435–448.

SIAM J. Comput.
Vol. 29, No. 2, pp. 433–458

PROXIMITY IN ARRANGEMENTS OF ALGEBRAIC SETS[*]

J. H. RIEGER[†]

Abstract. Let X be an arrangement of n algebraic sets X_i in d-space, where the X_i are either parametrized or zero-sets of dimension $0 \leq m_i \leq d-1$. We study a number of decompositions of d-space into connected regions in which the distance-squared function to X has certain invariances. Each region is contained in a single connected component of the complement of the bifurcation set $\mathcal{B}$ of the family of distance-squared functions or of certain subsets of $\mathcal{B}$. The decompositions can be used in the following proximity problems: given some point, find the k nearest sets X_i in the arrangement, find the nearest point in X, or (assuming that X is compact) find the farthest point in X and hence the smallest enclosing $(d-1)$-sphere. We give bounds on the complexity of the decompositions in terms of n, d, and the degrees and dimensions of the algebraic sets X_i.

Key words. k nearest points, Voronoi regions, bifurcation sets, critical points, symbolic computation

AMS subject classifications. 68U05, 68Q40, 51M20, 53A07, 57R45

PII. S0097539796300945

1. Introduction.

Let X be the union of n algebraic sets X_i of dimension $0 \leq m_i \leq d-1$ in d-space which are defined either by parametrizations or, more generally, as zero-sets. The dimension d of the ambient space is assumed to be arbitrary but fixed. Given a point $p \in \mathbb{R}^d$ with rational or, more generally, with algebraic number coordinates and a set of defining polynomials of X with rational coefficients, we would like to do the following:

1. find the k nearest sets X_i;
2. find the nearest point in X;
3. and, provided that X is compact, find the farthest point in X (and hence the smallest sphere with center p enclosing X).

For all of these proximity problems it is convenient to decompose d-space into certain connected regions, depending on X, in which the distance-squared function to X has certain invariances. A number of such decompositions are possible. Some decompositions have many invariants but also many regions, and it is of interest to bound the number of regions in terms of n, d, and the degrees and dimensions of the algebraic sets X_i. For example, the coarsest decomposition considered below consists of the first-order Voronoi regions, and the finest consists of the regions in the complement of the bifurcation set of the family of all distance-squared functions on X. However, all the decompositions studied here have the property that the proximity problems above can be solved in $O(\log n) \cdot P$ time (discarding the preprocessing time for constructing the decomposition), where P is a polynomial in the degrees and coefficient sizes of both the defining polynomials of X and the minimal polynomials of the algebraic number coordinates of p (from section 4 on we shall often concentrate on the combinatorial complexity, where the degrees and coefficient sizes of these polynomials are assumed to be bounded by some constant independent of n). Decompositions of

[*]Received by the editors March 20, 1996; accepted for publication (in revised form) May 24, 1998; published electronically October 5, 1999. Most of this work was done at the Max-Planck-Institut für Informatik in Saarbrücken, Germany, and has appeared in preliminary form as internal report MPI-96-1-003.

http://www.siam.org/journals/sicomp/29-2/30094.html

[†]Institut für Algebra and Geometrie, FB Mathematik und Informatik, Martin-Luther-Universität Halle, 06099 Halle (Saale), Germany (rieger@mathematik.uni-halle.de).

d-space into regions made of points having certain proximity properties with respect to some collection of submanifolds of $\mathbb{R}^d$ have been studied both in computational geometry and in singularity theory, but there hasn't been much interaction between these fields.

Most of the works in computational geometry consider either the classical, first-order, Voronoi diagram of sets of isolated points or extensions to arrangements of linear subspaces of $\mathbb{R}^d$. The relation between higher-order Voronoi diagrams in $\mathbb{R}^d$ and arrangements in $\mathbb{R}^{d+1}$ is investigated by Edelsbrunner and Seidel [12]. A few works also consider Voronoi diagrams of arrangements of curved objects. First-order Voronoi diagrams of disjoint convex semialgebraic sites in d-space are studied in the book of Sharir and Agarwal [22]. Alt and Schwarzkopf [1] study first-order Voronoi diagrams of parametrized (semialgebraic) curve-segments and points in the plane. These authors are also interested in the local geometry of Voronoi edges: for example, they point out that end-points of self-Voronoi-edges (in the singularity theory literature known as symmetry sets) correspond to centers of osculating circles at curvature extrema of a planar curve and also to a cusp singularity of the evolute (or focal set). The local geometry of such symmetry sets and of evolutes has been studied in great detail in a number of singularity theory works.

One of the main topics of singularity theory is the classification of stable and unstable singularities of functions and maps, and of the bifurcation sets in the parameter space of families of functions and maps. The bifurcation set of a family of functions $F : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d \times \mathbb{R}$, $(p, x) \mapsto (p, f(p, x))$ consists of all points p in parameter space $\mathbb{R}^d$ for which the function $x \mapsto f(p, x)$ has an unstable (degenerate) singularity. The family of distance-squared functions from any point $p \in \mathbb{R}^d$ to a parametrized m-dimensional surface X in d-space is a particular example of such a family, and the bifurcation set of this family is precisely the union of the evolute and the symmetry set of X. Porteous [16, 17] has used the classification of families of functions by Thom (from the early 1960s) to study the relation between the geometry of evolutes and the curvature of surfaces. Bruce, Giblin, and Gibson [6] have classified the singularities of symmetry sets of planar curves and of surfaces and space-curves in $\mathbb{R}^3$; see also the recent paper by Bruce [5]. Symbolic algorithms for computing bifurcation sets of families of projection maps have been studied by Rieger [19, 20] and these algorithms can also be used, with some minor modifications, to compute other bifurcation sets, such as evolutes and symmetry sets.

1.1. Assumptions and some notation. Let $X := \cup X_i \subset \mathbb{R}^d$ be a collection of n closed algebraic sets X_i and set $m_i := \dim X_i$ $(0 \le m_i \le d-1)$ and $m := \sup m_i$. For parametrized algebraic m_i-surfaces $x \mapsto X_i(x)$ we denote the maximal degree of the d component functions of $X_i(x)$ by δ_i, and set $\delta := \sup \delta_i$ $(1 \le i \le n)$. For the more general case of zero-sets $X_i = h_i^{-1}(0)$, where $h_i := (h_i^1, \dots, h_i^{d-m_i}) : \mathbb{R}^d \to \mathbb{R}^{d-m_i}$, we assume that X_i, or rather its complexification, is a complete intersection (i.e., its codimension is equal to the number of defining equations) and we set $\Delta_i := \Pi_j \deg h_i^j = \deg X_i$ and $\Delta := \sup \Delta_i$ (geometrically, $\deg X_i$ is the number of real and complex intersection points, including those "at infinity," of X_i and a "generic" linear subspace of $\mathbb{R}^d$ of dimension $d - m_i$).

The following notation will be used in this paper: $Z(I)$ denotes the zero-set of an ideal I, $I(Z)$ the ideal of polynomials vanishing on Z, $I : J$ the ideal quotient, and cl$Z$ denotes the closure of the set Z. Also, $\langle g_1, \dots, g_s \rangle$ denotes the ideal generated by the g_i, $1 \le i \le s$. The components of a vector $x = (x^1, \dots, x^d)$ are denoted by superscripts, so that subscripts can be used to enumerate elements of sets; and $(x^d)^3$

denotes the third power of the dth component.

Next, we need some notation from singularity theory. For most parts of this paper (sections 2 to 5), it is enough to remember the following notations: a nondegenerate critical point of a function, where the matrix of second derivatives has maximal rank, is of type A_1 (or of Morse type), a pair of A_1 points having the same critical values is denoted by A_1^2, and the least degenerate of the degenerate critical points is of type A_2. The bifurcation set $\mathcal{B} \subset \mathbb{R}^d$ of a family of distance-squared functions is a generally singular hypersurface whose regular components correspond to (codimension 1) singularities of type A_2 or A_1^2 of the distance-squared function. However, in section 6 we need to count the strata of the singular locus of $\mathcal{B}$ corresponding to more degenerate types of singularities (of codimension ≥ 2). The following notation for these singularities is more or less standard (for more details on the classification of functions up to $\mathcal{K}$- and $\mathcal{R}$-equivalence see chapter 8 of Dimca's book [10]). A function $f : \mathbb{R}^d \to \mathbb{R}$, $x := (x^1, \ldots, x^d) \mapsto f(x)$ has an A_k-singularity at $x = 0$ if there exists a smooth coordinate change $h : \mathbb{R}^d \to \mathbb{R}^d$, defined in the neighborhood of $x = 0$, such that $f \circ h(x) = c + (x^1)^{k+1} + \sum_{i=2}^d \epsilon_i (x^i)^2$, where c is some constant and $\epsilon_i = \pm 1$; see [2]. In other words, A_k denotes an equivalence class of function-germs (we consider pm;u f and h near $x = 0$), and the formula above describes a particular representative of this class. The equivalence classes A_k are orbits of the Mather groups $\mathcal{K}$ and $\mathcal{R}$. We shall abuse the notation $A_{\geq k}$ slightly: it will denote *all* classes of singularities in the closure of the A_k orbit, not just the orbits represented by $c + (x_1)^{\geq k+1} + \sum \epsilon_i (x^i)^2$. Finally, the function f has an $A_{\geq 1}^r$-singularity at a set of points $\{x_1, \ldots, x_r\}$, if it has an $A_{\geq 1}$-singularity at each $x_i \in \mathbb{R}^d$ and the r critical values $f(x_i)$ coincide.

Throughout this paper, $d^k f(p, x)[v_1, \ldots, v_l]$ will always denote the kth differential of a function f with respect to the variables $x \in \mathbb{R}^m$ and not with respect to the parameter $p \in \mathbb{R}^d$, and this k-linear form $d^k f$ should be multiplied with the vectors $v_i \in \mathbb{R}^m$, $1 \leq i \leq l$. Occasionally, we shall omit the parameters and variables (p, x).

Given a hypersurface $M \subset \mathbb{R}^d$, we denote the arrangement cut-out by M by $\mathcal{A}(M)$. We denote by $|\mathcal{A}(M)|$ the size of this arrangement, that is, the number of i-cells, $0 \leq i \leq d$, in $\mathcal{A}(M)$. Note that the connected regions of $\mathbb{R}^d \setminus M$ are the d-cells in $\mathcal{A}(M)$.

The Voronoi diagram of order k of a set $S := \{X_1, \ldots, X_n\}$ of algebraic sets $X_i \subset \mathbb{R}^d$ is defined as follows. Set $\mu_p(X_i) := \inf_{q \in X_i} \|q - p\|^2$ and let $\tilde{S} \subset S$ be a subset with k elements, $1 \leq k \leq n - 1$. Then

$$V_k(\tilde{S}) := \{p \in \mathbb{R}^d : \mu_p(X_i) < \mu_p(X_j), \text{for all } (X_i, X_j) \in \tilde{S} \times (S \setminus \tilde{S})\}$$

is the kth-order Voronoi cell of $\tilde{S}$ (which, in general, is not connected). The kth-order Voronoi surface V_k of S is the union of the boundaries of such Voronoi cells, i.e., $V_k := \cup_{\tilde{S} \subset S} \partial V_k(\tilde{S})$, and the kth-order Voronoi diagram is the arrangement $\mathcal{A}(V_k)$.

1.2. Contents of following sections. In section 2 we study the bifurcation set $\mathcal{B}$ of the family of distance-squared functions on an arrangement of m_i-surfaces X_i in $\mathbb{R}^d$ which are parametrized by polynomial maps. In particular, we give bounds for the number of regions in the complement of $\mathcal{B}$ (and, in fact, for $|\mathcal{A}(\mathcal{B})|$) and describe certain invariants which characterize these regions. We also obtain a result on the local topology of $\mathcal{B}$ that yields a priori information on how the semialgebraic components of $\mathcal{B}$ are glued together. This result is valid both for parametrized surfaces X_i and for zero-sets and does not assume that the family of distance-squared functions is versal (this is a common assumption in singularity theory works on this subject that does not necessarily hold for "almost all" algebraic surfaces X_i of some bounded degree).

In section 3 we consider the more general case of arrangements of algebraic zero-sets X_i (note that most zero-sets do not have a global parametrization given as the image of some polynomial map). For zero-sets we exploit the geometric characterization of the singularities of the distance-squared function in terms of the contact order (or intersection multiplicity) of X_i with certain $(d-m_i)$-spheres, where $m_i = \dim X_i$. This avoids the problem of finding local parametrizations of the X_i given by analytic maps (working with polynomials is much more convenient). The more classical case of contact between hypersurfaces X_i and osculating circles is treated in subsection 3.1; the more complicated case of contact between algebraic sets X_i of codimension $d - m_i \geq 2$ and $(d - m_i)$-spheres is studied in subsection 3.2.

In section 4 we describe an algorithm for determining the regions in the complement of the bifurcation set $\mathcal{B}$. This algorithm is similar, in its overall structure, to the algorithms in [19, 20] and uses standard techniques from computational algebra. We also describe solutions to the proximity problems 1 to 3 stated at the beginning of this introduction, which are based on the decomposition of d-space into regions in the complement of $\mathcal{B}$ or of certain subsets of $\mathcal{B}$.

In section 5 we present a few examples of these decompositions for curves and points in the plane, which have been computed with the methods described in section 4.

Finally, in section 6, we compare the combinatorial complexities of the arrangements $\mathcal{A}(\mathcal{B})$ and of the kth-order Voronoi diagrams $\mathcal{A}(V_k)$. Note that the boundaries V_k of the Voronoi regions of order k are subsets of $\mathcal{B}$, and the bounds in sections 2 and 3 are therefore upper bounds for the complexity of the kth-order Voronoi diagram of arrangements of algebraic sets in terms of n, d, and the degrees and dimensions of these sets. A comparison of the combinatorial complexities of the bifurcation set $\mathcal{B}$ and of the Voronoi boundaries V_k for the more special arrangements studied in [1] and [22] shows that there is a considerable gap, which can be partially understood by studying the combinatorial complexity of certain intermediate sets $V_k \subset R_k \subset \mathcal{B}$. Even so, for general arrangements of algebraic sets, an asymptotically tight bound (at least in terms of combinatorial complexity) for the number of regions of $\mathbb{R}^d \setminus V_k$ remains a widely open problem (for n intersecting hypersurfaces in $\mathbb{R}^d$ we show, for example, that $|\mathcal{A}(V_1)| \sim O(n^{d+1})$ and $|\mathcal{A}(V_1)| \sim \Omega(n^d)$).

2. The complement of the bifurcation set $\mathcal{B}$ of a family of distance-squared functions.

In this section the algebraic m_i-surfaces X_i of the arrangement are parametrized by polynomial maps $x \mapsto X_i(x)$, where $x = (x^1, \ldots, x^{m_i}) \in \mathbb{R}^{m_i}$. The necessary modifications in the (more general) case of zero-sets will be described in section 3. The family of distance-squared functions on X_i is defined by

$$F_i : \mathbb{R}^d \times \mathbb{R}^{m_i} \to \mathbb{R}^d \times \mathbb{R}, \quad (p, x) \mapsto (p, f_i(p, x) := \|X_i(x) - p\|^2).$$

Recall that an element of this d-parameter family of functions in m_i variables is a Morse function if its critical points are nondegenerate (i.e., the corresponding matrix of second derivatives has maximal rank) and have distinct critical values. The bifurcation set $\mathcal{B}_i \subset \mathbb{R}^d$ of the family F_i is the set of "bad" parameters p for which $x \mapsto f_i(p, x)$ fails to be a Morse function. The set $\mathcal{B}_i$ is the union of the local bifurcation set

$$\mathcal{E}_i := \{p \in \mathbb{R}^d : \exists x : df_i(p, x) = 0, \text{rank } d^2 f_i(p, x) < m_i\},$$

consisting of $A_{\geq 2}$-singularities, and the level bifurcation set

$$\mathcal{S}_i := \mathrm{cl}\{p \in \mathbb{R}^d : \exists x \neq \bar{x} : df_i(p,x) = df_i(p,\bar{x}) = 0, f_i(p,x) = f_i(p,\bar{x})\},$$

consisting of $A_{\geq 1}^{\geq 2}$-singularities. (The notation $\mathcal{E}_i$ and $\mathcal{S}_i$ indicates that, from a classical differential geometry point of view, the local and level bifurcation sets are evolutes and symmetry sets, respectively; see section 3. Also, recall that in all functions depending on the parameters $p \in \mathbb{R}^d$, the differentials are with respect to the remaining variables.)

The bifurcation set $\mathcal{B}$ of the arrangement associated to $X = \cup X_i$ is the union of the bifurcation sets $\mathcal{B}_i$ of the X_i and the following intersurface level bifurcation sets:

$$\mathcal{S}_{i,j} := \{p \in \mathbb{R}^d : \exists x, \bar{x} : df_i(p,x) = df_j(p,\bar{x}) = 0, f_i(p,x) = f_j(p,\bar{x})\},$$

that is,

$$\mathcal{B} := \bigcup_{1\leq i\leq n} \mathcal{E}_i \cup \bigcup_{1\leq i\leq n} \mathcal{S}_i \cup \bigcup_{1\leq i<j\leq n} \mathcal{S}_{i,j}.$$

This definition of $\mathcal{B}$ assumes that $1 \leq \dim X_i \leq d-1$, but it can be extended easily to include isolated points $X_i = \{q_i\}$. For a point q_i the sets $\mathcal{E}_i$ and $\mathcal{S}_i$ are defined to be empty, for a point pair q_i, q_j the set $\mathcal{S}_{i,j}$ is defined to be the hyperplane perpendicular to $q_j - q_i$ through $(q_i + q_j)/2$, and for surface-point pairs X_i ($\dim X_i \geq 1$), $X_j = \{q_j\}$ we define

$$\mathcal{S}_{i,j} := \{p \in \mathbb{R}^d : \exists x : df_i(p,x) = 0, f_i(p,x) = \|q_j - p\|^2\}.$$

The definitions of the local bifurcation sets $\mathcal{E}_i$ and of the intersurface level bifurcation sets $\mathcal{S}_{i,j}$ are fairly straightforward from a computational point of view. The definition of the intrasurface level bifurcation sets $\mathcal{S}_i$ is less straightforward: the inequalities $x \neq \bar{x}$, together with the defining equations appearing in the definition, yield semialgebraic sets $\mathcal{S}'_i \subset \mathbb{R}^d \times \mathbb{R}^{2m_i}$ which are not closed. It is, however, possible to close up the sets $\mathcal{S}'_i$ by adding a set of boundary points $\partial\mathcal{S}'_i$ on the diagonal $\{x = \bar{x}\} \subset \mathbb{R}^{2m_i}$ (see below). Furthermore, the closed sets $\tilde{\mathcal{S}}_i := \mathcal{S}'_i \cup \partial\mathcal{S}'_i$ can be defined by polynomial equations (inequations are not required), which is a big advantage from a computational algebra point of view.

Let $\bar{\mathcal{S}}_i \subset \mathbb{R}^d \times \mathbb{R}^{2m_i}$ denote the set defined by the defining equations of $\mathcal{S}'_i$ (omitting the inequalities $x \neq \bar{x}$); this is a closed set which coincides with $\mathcal{S}'_i$ away from the diagonal, but has too high dimension on the diagonal. Then the vanishing ideal of the closure of $\mathcal{S}'_i$ is given by the ideal quotient

$$I(\bar{\mathcal{S}}_i) : (I(\bar{\mathcal{S}}_i) + \langle x^1 - \bar{x}^1, \ldots, x^{m_i} - \bar{x}^{m_i}\rangle),$$

whose generators can be determined using Gröbner basis methods (see, for example, [3, Chapter 6.2]). For zero-sets X_i of codimension ≥ 2 we actually close up the sets $\mathcal{S}'_i$ in this way (see section 3.2). However, for arrangements of parametrized sets (and of zero-sets of codimension 1; see section 3.1) the defining equations of the closure of $\mathcal{S}'_i$ can be constructed in a more direct way that avoids the expensive computation of Gröbner bases and also yields some useful information about the topology of the

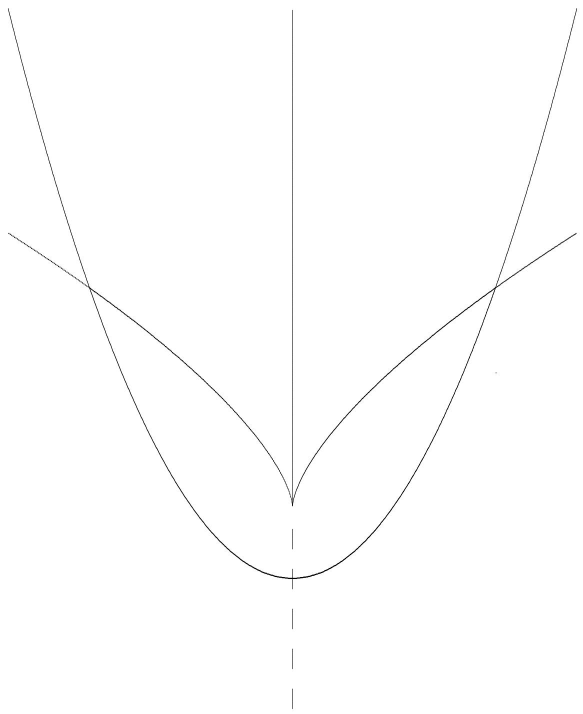


FIG. 1. *The bifurcation set $\mathcal{B}_i$ of a single parabola X_i: the local bifurcation set $\mathcal{E}_i$ is the cusp-shaped curve and the level bifurcation set $\mathcal{S}_i$ is the solid half-line whose boundary $\partial\mathcal{S}_i$ in the union $\hat{\mathcal{S}}_i$ of the solid and dashed vertical line coincides with the cusp point. The distance-squared function to X_i from points of $\mathcal{B}_i$ has the following singularities: A_3 at the cusp, A_2 for all other points of $\mathcal{E}_i$, and A_1^2 for all other points of $\mathcal{S}_i$.*

bifurcation set $\mathcal{B}_i$. We first give an outline of this construction and its topological consequences; more detailed statements follow in Proposition 2.1 and its proof.

To find the defining equations of the closure of $\mathcal{S}'_i$, we first "blow up" the diagonal $\{x = \bar{x}\} \subset \mathbb{R}^{2m_i}$ by a change of coordinates β that replaces the pair of points $(x, \bar{x})$ by $(x, x + \lambda \cdot \omega)$, where $\lambda \in \mathbb{R}$ and $\omega \in \mathbb{P}^{m_i - 1}$ (i.e., we represent the point $\bar{x}$ by moving it some distance λ along a ray through x and direction ω). The map β is an isomorphism for $\lambda \neq 0$ "blowing down" the hyperplane $\{\lambda = 0\}$ to the diagonal $\{x = \bar{x}\}$, which is a linear subspace of $\mathbb{R}^{2m_i}$ of codimension m_i. Next, we choose a certain set of generators of $I(\bar{\mathcal{S}}_i)$ and divide them by suitable powers of λ. The modified generators define a closed algebraic set $\tilde{\mathcal{S}}_i$ that coincides with the sets $\bar{\mathcal{S}}_i$ and $\mathcal{S}'_i$ in the complement of the diagonal $\{\lambda = 0\}$. The set of boundary points of $\mathcal{S}'_i$ on the diagonal is given by $\partial\mathcal{S}'_i = \tilde{\mathcal{S}}_i \cap \{\lambda = 0\}$. Using the defining equations of the sets $\tilde{\mathcal{S}}_i$, $\mathcal{S}'_i$, and $\bar{\mathcal{S}}_i$ and excluding a Zariski closed subset of X_i in the space of all algebraic sets (where m_i, δ_i, and d are fixed) one easily checks the following properties. The set $\mathcal{S}'_i$ has dimension $d-1$, and "almost all" of its points $(p, x, \bar{x})$ correspond to pairs of A_1-singularities of the distance-squared function having the same critical values $f_i(p, x) = f_i(p, \bar{x})$. The set $\partial\mathcal{S}'_i$ has dimension $d-2$ and almost all of its points correspond to A_3-singularities. (By contrast, most points of $\bar{\mathcal{S}}_i \cap \{\lambda = 0\}$ merely correspond to A_1-singularities and form a $d+m_i$-dimensional set.) It therefore follows that the projection of $\partial\mathcal{S}'_i$ into the parameter space $\mathbb{R}^d$ is contained in the local bifurcation set $\mathcal{E}_i$ (which corresponds to

$A_{\geq 2}$-singularities). The projection of $\partial\mathcal{S}_i'$, denoted by $\partial\mathcal{S}_i$, also forms the boundary of the semialgebraic level bifurcation set $\mathcal{S}_i \subset \mathbb{R}^d$ in the smallest real algebraic set $\hat{\mathcal{S}}_i$ containing $\mathcal{S}_i$: moving along a generic path from $\mathcal{S}_i$ to $\hat{\mathcal{S}}_i \setminus \mathcal{S}_i$ we first get a pair of real A_1-singularities having the same critical value, which coalesce in an A_3-singularity as we cross the boundary $\partial\mathcal{S}_i$ and then become complex. Figure 1 illustrates this situation in the simple case of a single parabola $X_i(x) = (x, x^2)$ in the plane. We can now give a more precise description of this construction.

PROPOSITION 2.1.

(i) *For all $(p, x, x) \in \partial\mathcal{S}_i'$, the distance-squared function $x \mapsto f_i(p, x)$ has an $A_{\geq 3}$-singularity at x. This implies that $\pi(\partial\mathcal{S}_i') \subset \mathcal{E}_i$, where $\pi : \mathbb{R}^d \times \mathbb{R}^{2m_i} \to \mathbb{R}^d$ denotes the projection onto the first factor.*

(ii) *The degree of $\partial\mathcal{S}_i'$ is of order $\delta_i^{2m_i+1}$ and that of $\cup\partial\mathcal{S}_i'$ of order $n \cdot \delta^{2m+1}$.*

Proof. (i) The proof of the first part of the proposition follows the construction outlined above. Set $A := (a^1, \ldots, a^{m_i-1}, 1)$, then the map given by

$$\beta : \mathbb{R}^{2m_i} \to \mathbb{R}^{2m_i}, \quad (x, \lambda, A) \mapsto (x, x + \lambda \cdot A)$$

"blows down" the hyperplane $\{\lambda = 0\}$ to the diagonal $\{x = \bar{x}\}$ and has maximal rank for $\lambda \neq 0$. Note that we have replaced the space of directions $\omega \in \mathbb{P}^{m_i-1}$ by the affine chart of vectors A in $\mathbb{R}^{m_i}$ whose last component is equal to 1. To cover all of $\mathbb{P}^{m_i-1}$, m_i such charts are required, but it is easy to check that the arguments below do not depend on the choice of chart. We then claim that the set $\tilde{\mathcal{S}}_i$ can be defined by the following three equations (omitting the inequality $\lambda \neq 0$): by $df_i(p, x) = 0$ (as before), and by

$$U_i(p, x, \lambda, A) := \lambda^{-1}\big(df_i(p, x + \lambda \cdot A) - df_i(p, x)\big) = 0,$$

and by

$$V_i(p, x, \lambda, A) := \lambda^{-3}\Big(f_i(p, x + \lambda \cdot A) - f_i(p, x) - \lambda df_i(p, x)[A] - \frac{\lambda^2}{2}\langle U_i, A\rangle\Big) = 0.$$

It is easy to see that, away from the diagonal $\{\lambda = 0\}$,

$$df_i(p, x) = U_i(p, x, \lambda, A) = V_i(p, x, \lambda, A) = 0$$

and the original system

$$df_i(p, x) = df_i(p, x + \lambda \cdot A) = f_i(p, x) - f_i(p, x + \lambda \cdot A) = 0$$

define the same zero-sets $\mathcal{S}_i' \subset \mathbb{R}^d \times \mathbb{R}^{2m_i} \setminus \{\lambda = 0\}$. Furthermore, the right-hand sides of U_i and V_i are divisible by λ and λ^3 (by Taylor's theorem); hence $df_i = U_i = V_i = 0$ defines a closed algebraic variety $\tilde{\mathcal{S}}_i := \mathcal{S}_i' \cup \partial\mathcal{S}_i' \subset \mathbb{R}^d \times \mathbb{R}^{2m_i}$.

In fact, $\tilde{\mathcal{S}}_i$ is the smallest closed set containing $\mathcal{S}_i'$, and the boundary $\partial\mathcal{S}_i' := \tilde{\mathcal{S}}_i \cap \{\lambda = 0\}$ of $\mathcal{S}_i'$ in $\tilde{\mathcal{S}}_i$ corresponds to $A_{\geq 3}$-singularities of the distance-squared function $x \mapsto f_i(p, x)$. The boundary $\partial\mathcal{S}_i'$ is defined by the following equations:

$$\begin{aligned} df_i(p, x) &= 0, \\ U_i(p, x, 0, A) = d^2 f_i(p, x)[A] &= 0, \\ V_i(p, x, 0, A) = -\frac{1}{12} d^3 f_i(p, x)[A^3] &= 0. \end{aligned}$$

This system "recognizes" an $A_{\geq 3}$-singularity of $x \mapsto f_i(p,x)$ at x—the condition for an $A_{\geq 3}$-singularity is precisely that $df_i = 0$ and $d^2 f_i[v] = 0$, $d^3 f_i[v^3] = 0$ for some nonzero vector v (see, for example, Porteous [18, p. 397]).

(ii) The degree of the variety $\partial \mathcal{S}'_i$ defined by the above system of equations is at most of order $\delta_i^{2m_i+1}$ (by Bezout's theorem), so that the degree of the union of n such varieties is of order $\sum_{i=1}^{n} \delta_i^{2m_i+1} \leq n \cdot \delta^{2m+1}$. ∎

Remarks. 1. Patching together the $\tilde{\mathcal{S}}_i$ in the m_i affine charts in the proof of part i yields a variety $V \subset \mathbb{R}^d \times \mathbb{R}^{m_i+1} \times \mathbb{P}^{m_i-1}$ whose projection π onto $\mathbb{R}^d$ is the intrasurface level bifurcation set $\mathcal{S}_i$. To compute the defining equations of $\mathcal{S}_i$ it is sufficient to use a single "good" affine chart for $\mathbb{P}^{m_i-1}$: for example, $(a^1, \dots, a^{m_i-1}, 1)$ is good if $\dim V > \dim(V \cap \{a^{m_i} = 0\})$. In this case, the missing component of V "at infinity" will be closed up by the projection π.

2. Part i of the proposition also holds locally for germs of C^∞-submanifolds X_i of dimension m_i (note that the proof merely depends on the Taylor expansion of f_i at $(p,x) = (p_0, x_0)$). In particular, it also holds for algebraic zero-sets of dimension m_i (which will be studied in section 3), because these sets have a parametrization which is even analytic.

3. For $m = 1$ and $d = 2$, the set $\cup \partial \mathcal{S}'_i$ consists of isolated points $(p_l, x_l) \in \mathbb{R}^2 \times \mathbb{R}$ (this can be checked by a simple dimensional argument), and there are at most $O(n \cdot \delta^3)$ such endpoints by the proposition above. The projections p_l of these points into the plane are possible endpoints of the level-bifurcation set $\mathcal{S} = \cup \mathcal{S}_i$. However, their projections x_l onto $\mathbb{R}$ correspond to curvature extrema of X_i and each curvature extremum corresponds to one endpoint. But $X = \cup X_i$ has at most $O(n \cdot \delta)$ curvature extrema.

PROPOSITION 2.2. *For all points p in a single connected region of $\mathbb{R}^d \setminus \cup \mathcal{E}_i$ the collection of distance-squared functions $\{x \mapsto f_i(p,x) : 1 \leq i \leq n\}$ has a constant number, c, of critical points, where*

$$n \leq c \leq \sum_{i=1}^{n} (2\delta_i - 1)^{m_i} \sim O(n \cdot \delta^m).$$

Proof. From the definition of the local bifurcation sets $\mathcal{E}_i$ we see that the distance-squared functions $x \mapsto f_i(p,x)$ have isolated critical points (of multiplicity 1) for all $p \in \mathbb{R}^d \setminus \cup \mathcal{E}_i$. Each f_i is nonnegative and has degree $2\delta_i$. Hence, each f_i has at least one local minimum and at most $(2\delta_i - 1)^{m_i}$ critical points; this yields the desired bounds for c. ∎

Remark. For arrangements of hypersurfaces X_i (i.e., $m_i = d-1$) the number of critical points c has the following geometrical interpretation: it is equal to the number of normal lines of $X = \cup X_i$ passing through the point p.

PROPOSITION 2.3. *The number of connected regions of $\mathbb{R}^d \setminus \mathcal{B}$ (and, in fact, the size of $\mathcal{A}(\mathcal{B})$) are at most of order $n^{2d} \cdot \delta^{(2m+1)d}$. Furthermore, let $p \in \mathbb{R}^d \setminus \mathcal{B}$, and let*

$$\xi_{1,\nu_1}(p), \xi_{2,\nu_2}(p), \dots, \xi_{c,\nu_c}(p)$$

denote the critical points $\xi_{l,\nu_l}(p)$ of the collection of distance-squared functions $x \mapsto f_{\nu_l}(p,x)$, where $\nu_l \in \{1, \dots, n\}$, ordered by increasing distance. That is, $f_{\nu_l}(p, \xi_{l,\nu_l}(p)) < f_{\nu_{l+1}}(p, \xi_{l+1,\nu_{l+1}}(p))$. For all points p in a single connected region of $\mathbb{R}^d \setminus \mathcal{B}$ we have the following: (i) *the numbers $\nu_1, \dots, \nu_c$ are invariant, and* (ii) *the maps $p \mapsto \xi_{l,\nu_l}(p)$ are continuous for $1 \leq l \leq n$.*

Proof. The bifurcation set $\mathcal{B}$ is a semialgebraic subset of a closed real algebraic set $\hat{\mathcal{B}} \subset \mathbb{R}^d$, and the number of connected regions cut out by $\mathcal{B}$ is less than or equal to the number of regions cut out by $\hat{\mathcal{B}}$. The number of connected regions of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ is equal to the $(d-1)$st Betti number of $\hat{\mathcal{B}}$ plus 1 (see below), and the desired upper bound follows at once from a result of Milnor [15] (which says that the sum of the Betti numbers of $\hat{\mathcal{B}}$ is of order $(\deg \hat{\mathcal{B}})^d$) and the bound for the degree of $\hat{\mathcal{B}}$ derived below. (On the other hand, the singular stratification of $\hat{\mathcal{B}}$ has $O((\deg \hat{\mathcal{B}})^d)$ strata and $|\mathcal{A}(\mathcal{B})| \leq |\mathcal{A}(\hat{\mathcal{B}})|$, which yields the bound for $|\mathcal{A}(\mathcal{B})|$.)

The (linear) formula for the number of connected components of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ in terms of the $(d-1)$st Betti number of $\hat{\mathcal{B}}$ follows from standard duality results in algebraic topology: roughly speaking, either from Lefschetz duality (which yields an isomorphism between the zeroth homology group H_0 of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ and the dth cohomology group H^d of the pair $(\mathbb{R}^d, \hat{\mathcal{B}})$) and the isomorphism $H_d(\mathbb{R}^d, \hat{\mathcal{B}}) \cong H_{d-1}(\hat{\mathcal{B}})$ (coming from the standard exact homology sequence of the pair $(\mathbb{R}^d, \hat{\mathcal{B}})$); or, more directly, one can use Alexander duality to get an isomorphism between H_0 of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ and H^{d-1} of $\hat{\mathcal{B}}$. The more precise argument (included in parentheses below) is a bit more complicated, due to the possible noncompactness of $\hat{\mathcal{B}}$ and the appearance of reduced homology groups and might be skipped. (Let $S^d = \mathbb{R}^d \cup \{\infty\}$ and $\hat{\mathcal{B}}^c = \hat{\mathcal{B}} \cup \{\infty\}$ denote 1-point compactifications; then the Alexander duality yields the following isomorphism of reduced (co)homology groups $\tilde{H}_0(S^d \setminus \hat{\mathcal{B}}^c) \cong \tilde{H}^{d-1}(\hat{\mathcal{B}}^c)$. (See [11, Section 8.15, Chapter VIII]). The set $\hat{\mathcal{B}}$ is a closed real algebraic set; hence $\tilde{H}_0(S^d \setminus \hat{\mathcal{B}}^c) \cong \tilde{H}_0(\mathbb{R}^d \setminus \hat{\mathcal{B}})$ and $b_i(\hat{\mathcal{B}}^c) = b_i(\hat{\mathcal{B}})$, for all $i > 0$. Finally, note that the rank of the zeroth homology group is one plus that of the reduced one.)

We claim that the degree of $\hat{\mathcal{B}}$ is of order $n^2\delta^{2m+1}$. The set $\hat{\mathcal{B}}$ is the union of $\binom{n}{2}$ (real algebraic) sets $\hat{\mathcal{S}}_{i,j}$, n sets $\hat{\mathcal{S}}_i$, and n sets $\hat{\mathcal{E}}_i$. The orders of the degrees of the $\hat{\mathcal{S}}_i$ and the $\hat{\mathcal{E}}_i$ are lower than those of the $\hat{\mathcal{S}}_{i,j}$; hence it suffices to estimate the degree of $\hat{\mathcal{S}}_{i,j}$. So let $\tilde{\mathcal{S}}_{i,j} \subset \mathbb{R}^d \times \mathbb{R}^{m_i+m_j}$ be the real algebraic set defined by the defining equations of $\mathcal{S}_{i,j}$ (omitting the existential quantifier). The restriction of the projection $\pi : \mathbb{R}^d \times \mathbb{R}^{m_i+m_j} \to \mathbb{R}^d$ to $\tilde{\mathcal{S}}_{i,j}$ yields the semialgebraic set $\mathcal{S}_{i,j}$. Complexifying the defining equations of $\tilde{\mathcal{S}}_{i,j}$ and taking the real part of the projection π onto $\mathbb{C}^d$ of the resulting zero-set yields a closed real algebraic set $\hat{\mathcal{S}}_{i,j} \subset \mathbb{R}^d$ which contains the semialgebraic set $\mathcal{S}_{i,j}$. Suppose that $\operatorname{codim}\hat{\mathcal{S}}_{i,j} = 1$ (otherwise the complement of $\hat{\mathcal{S}}_{i,j}$ is connected, and we are finished), and let $\mathcal{L} \subset \mathbb{R}^d$ be any line. Now there are two cases: (1) the set $A := \pi^{-1}(\mathcal{L}) \cap \tilde{\mathcal{S}}_{i,j}$ consists of isolated points (the "generic case") and (2) $\dim A = e > 0$. Let $\bar{\pi} : \mathbb{R}^d \times \mathbb{R}^{m_i+m_j} \to \mathbb{R}^{m_i+m_j}$ denote the projection onto the second factor, let $\bar{\mathcal{L}} \subset \mathbb{R}^{m_i+m_j}$ be any linear subspace of codimension e such that $\bar{\pi}^{-1}(\bar{\mathcal{L}})$ is not contained in A and set $\bar{A} := A \cap \bar{\pi}^{-1}(\bar{\mathcal{L}})$. The sets A in case 1 and $\bar{A}$ in case 2 are discrete point sets, and the restriction of π to these point sets onto the set of intersection points $\hat{\mathcal{S}}_{i,j} \cap \mathcal{L}$ is surjective. The degree of $\hat{\mathcal{S}}_{i,j}$ is therefore bounded by the number of points of A (or $\bar{A}$ in case 2). Inspecting the defining equations of these sets, we get from Bezout's theorem that

$$\deg \hat{\mathcal{S}}_{i,j} \leq (2\delta_i - 1)^{m_i} \cdot (2\delta_j - 1)^{m_j} \cdot 2\max(\delta_i, \delta_j)$$

(counting both real and complex roots with their multiplicities).

For the proof of the second part of the proposition, consider the following real algebraic set:

$$\Sigma_{F_i} := \{(p, x) : df_i(p, x) = 0\} \subset \mathbb{R}^d \times \mathbb{R}^{m_i}.$$

The set Σ_{F_i} is the critical set of the family F_i of all distance-squared functions f_i on X_i. The fibers $\pi^{-1}(p) \cap \Sigma_{F_i}$ of the projection $\pi : \mathbb{R}^d \times \mathbb{R}^{m_i} \to \mathbb{R}^d$ correspond to the critical points of f_i from p. The restriction of π to Σ_{F_i} is a covering map whose branch-locus is the (preimage of the) evolute $\mathcal{E}_i$ and which is finite-to-one off the branch-locus. The number of points in each fiber $\pi^{-1}(p) \cap \Sigma_{F_i}$ is therefore finite and constant for all points p in a connected region of $\mathbb{R}^d \setminus \mathcal{E}_i$. The same is true for the total number c of critical points of a collection $\{f_i\}_{1 \le i \le n}$ of distance-squared functions on $X := \cup X_i$ for all p in a single connected region of $\mathbb{R}^d \setminus \cup\mathcal{E}_i$. Furthermore, the indices $\nu_1, \dots, \nu_c$ are invariant within a connected region of $\mathbb{R}^d \setminus (\cup\mathcal{E}_i) \cup (\cup\mathcal{S}_{i,j})$, because c is constant and permutations of indices can only occur along the intersurface level bifurcation sets $\mathcal{S}_{i,j}$. Finally, let U be any connected region of $\mathbb{R}^d \setminus \cup\mathcal{E}_i$ and consider the union of the n bundles $\cup_{i=1}^n \pi^{-1}(U) \cap \Sigma_{F_i}$. This is a semialgebraic set consisting of c disjoint components of dimension d, and these components are the graphs of continuous maps $h_j : U \to \mathbb{R}^{m_i}$, $1 \le j \le c$ (these facts are established by arguments that are quite similar to the proof of the first main structure theorem in [4, Chapter 2.2]; in fact, most stratification schemes of semialgebraic sets seem to be based on some version of this theorem). The composition of the h_j with the projection $\bar{\pi} : \mathbb{R}^d \times \mathbb{R}^{m_i} \to \mathbb{R}^{m_i}$ is a continuous map, which implies that the c critical points of the collection of distance-squared functions $x \mapsto f_i(p,x)$, $1 \le i \le n$, vary continuously with $p \in U$. The continuity of the maps $p \mapsto \xi_{l,\nu_l}(p)$ for all p within a single connected region of $\mathbb{R}^d \setminus \mathcal{B}$ then follows from the results above and the fact that the permutation of the critical points of a single function $x \mapsto f_i(p,x)$ can only occur on $\mathcal{S}_i$. □

3. Contact of X with spheres and the definition of $\mathcal{B}$ for zero-sets X.

The fibers of the distance-squared function from a point $p \in \mathbb{R}^d$ are $(d-1)$-spheres of varying radius r, given by $\{x \in \mathbb{R}^d : \|x - p\|^2 - r^2 = 0\}$. The conditions for an A_k-singularity of the distance-squared function, which appear in the definition of the bifurcation set $\mathcal{B}$, can be reformulated in more geometric terms involving the contact between a family of such spheres and a collection $X = \cup X_i$ of algebraic sets. Using these more geometric conditions, we can easily define, and compute, the bifurcation set $\mathcal{B}$ in the case of algebraic sets X_i given as zero-sets of polynomials $h_i^j \in \mathbb{Q}[x] = \mathbb{Q}[x^1, \dots, x^d]$, $1 \le j \le d - m_i$.

We first consider the special case of algebraic hypersurfaces (codimension 1) where the local and level bifurcation sets of the distance-squared function are the well-known evolutes and symmetry sets of classical differential geometry (section 3.1). In section 3.2 we consider the more general case of arrangements of algebraic sets X_i of codimension $1 \le d - m_i \le d - 1$ which are complete intersections (i.e., are defined by $d - m_i$ polynomials). Note that the case of points X_i (of codimension d) can be handled as in section 2.

3.1. Arrangements of hypersurfaces, evolutes, and symmetry sets.

First, recall that a hypersurface X_i in d-space has $d-1$ (not necessarily distinct) principal curvatures κ_j and directions d_j which are the eigenvalues and eigendirections of the Weingarten map. (The Weingarten map $W_p : T_pX_i \to T_pX_i$, $v \mapsto -\nabla_v N$ measures the rate of change of the normal direction N along a direction v in the tangent space of X_i at p.) A $(d-1)$-sphere is a curvature sphere at $x \in X_i$ if its center lies on the normal line through x and its radius r is the inverse of one of the principal curvatures of X_i at x. The unique great circle in this curvature sphere whose tangent line at x is oriented along the principal direction associated to $1/r$ is an osculating circle. The evolute (or focal surface) $\mathcal{E}_i$ of X_i is the locus of centers of such osculating circles and of the curvature spheres containing them (for each surface patch of X_i there are generically $d-1$ sheets of the evolute, one for each principal curvature).

The distance-squared function from $p \in \mathbb{R}^d$ to X_i has an A_k-singularity ($k \geq 1$) at $x \in \mathbb{R}^d$ if and only if there exists a circle with center p having $(k+1)$-point contact with X_i at x. The order of contact is ≥ 2 if p lies on the normal line to X_i at x and ≥ 3 if, in addition, the circle is an osculating circle. The local bifurcation set $\mathcal{E}_i$ consists of points p for which the distance-squared function to X_i has an $A_{\geq 2}$-singularity; such points are centers of osculating circles (and of curvature spheres). The local bifurcation set $\mathcal{E}_i$ is therefore the evolute of X_i. The relation between singularities of the distance-squared function, normal singularities of submanifolds (i.e., singularities of the exponential map of the normal bundle), and the possible types of contact between these submanifolds and spheres was first studied by Porteous; see [16] and [17].

The intra- and intersurface level bifurcation sets $\mathcal{S}_i$ and $\mathcal{S}_{i,j}$ are loci of centers of bitangent spheres touching $X = \cup X_i$ in two distinct points (the spheres can shrink to a point as their centers tend to the self-intersection locus of X). If both points of tangency lie on a single surface X_i then the center belongs to $\mathcal{S}_i$, otherwise it belongs to $\mathcal{S}_{i,j}$. Clearly, the distance-squared function from a center of a bitangent sphere has the two points of tangency as its critical points, and the corresponding critical values are given by the square of the radius of the bitangent sphere. The locus of centers of bitangent spheres of a hypersurface is known as symmetry set in the differential geometry literature, and the singularities of such symmetry sets of plane curves and of surfaces in 3-space have been classified by Bruce, Giblin, and Gibson [6]. (In the pattern recognition literature, the symmetry set of a plane curve is also known under the names skeleton, medial axis, and symmetric axis transform.)

Using these geometrical descriptions of the local bifurcation sets $\mathcal{E}_i$ and of the level bifurcation sets $\mathcal{S}_i$ and $\mathcal{S}_{i,j}$, we can now define the bifurcation set of the distance-squared functions for arrangements of algebraic hypersurfaces given as zero-sets $X_i = h_i^{-1}(0)$. Below, $V \| W$ denotes the condition that the pair of vectors V, W in $\mathbb{R}^d$ is parallel (obviously, this condition involves the vanishing of $d-1$ functions involving the components of the vectors), and $S(p,x,r) := \|x-p\|^2 - r^2$ defines a $(d-1)$-sphere with center p and radius r. The fact that (at least) one of the principal curvatures of X_i at x is equal to $1/r$ is equivalent to the vanishing of the following two equations:

$$Q_i(x,u) := \det \begin{pmatrix} (d^2 h_i(x) - u \cdot I) & dh_i(x) \\ (dh_i(x))^t & 0 \end{pmatrix}$$

(where I denotes the $d \times d$ identity matrix) and

$$R_i(x,u,r) := u^2 r^2 - \|dh_i(x)\|^2.$$

(The condition $Q_i = R_i = 0$ can be deduced easily from the standard formula for the principal curvatures of a hypersurface defined as zero-set; see, e.g., [23, p. 204]. Note that the derivation of this formula [23, pp. 202–204] is for hypersurfaces in 3-space, but the d-dimensional case ($d \geq 2$) is analogous.)

Using this notation, the local bifurcation sets (evolutes) are defined as follows:

$$\begin{aligned} \mathcal{E}_i := \{p \in \mathbb{R}^d : \exists x, u, r : & h_i(x) = S(p,x,r) = Q_i(x,u) = R_i(x,u,r) = 0, \\ & dh_i(x) \| dS(p,x,r)\}. \end{aligned}$$

The level bifurcation sets (symmetry sets) are given by

$$\begin{aligned} \mathcal{S}_i := \mathrm{cl}\{p \in \mathbb{R}^d : \exists x_1 \neq x_2 : & h_i(x_k) = 0, dh_i(x_k) \| (x_k - p), k = 1, 2; \\ & \|x_1 - p\|^2 = \|x_2 - p\|^2\} \end{aligned}$$

and

$$\begin{aligned}\mathcal{S}_{i,j} := \{p \in \mathbb{R}^d : \exists x_i, x_j : & h_k(x_k) = 0, dh_k(x_k) \| (x_k - p), k = i, j; \\ & \|x_i - p\|^2 = \|x_j - p\|^2\}.\end{aligned}$$

The estimates in Propositions 2.1, 2.2, and 2.3 for arrangements of parametrized surfaces, in terms of n and δ, have the following analogues, (i)–(iii) of 3.1, in the case of $(d-1)$-dimensional zero-sets.

PROPOSITION 3.1. *Let $\mathcal{B}$ denote the bifurcation set of the family of distance-squared functions on a collection $X = \cup_{i=1}^n X_i$ of algebraic hypersurfaces of maximal degree Δ. Then the following holds:* (i) *the degree of $\cup \partial \mathcal{S}'_i$ is at most of order $n \cdot \Delta^{2d}$;* (ii) *the number of critical points of the distance-squared function from any point $p \in \mathbb{R}^d \setminus \cup \mathcal{E}_i$ to X is at most of order $n \cdot \Delta^d$; and* (iii) *the number of connected regions of $\mathbb{R}^d \setminus \mathcal{B}$ and the size of $\mathcal{A}(\mathcal{B})$ are at most of order $n^{2d} \cdot \Delta^{2d^2}$.*

Proof. For statement (i), we modify the defining equations of $\mathcal{S}_i$, as in the case of parametrized sets X_i, by blowing up the diagonal $\{x_1 = x_2\} \subset \mathbb{R}^{2d}$ by setting $x_2 := x_1 + \lambda \cdot \omega$, where $\omega \in \mathbb{P}^{d-1}$, and by dividing certain generators of the resulting ideal by suitable powers of λ. Let $\tilde{\mathcal{S}}_i$ be the zero-set of these modified defining equations and let $\mathcal{S}'_i$ denote the semialgebraic set that coincides with $\tilde{\mathcal{S}}_i$ off the diagonal $\{\lambda = 0\}$; then again $\partial \mathcal{S}'_i = \tilde{\mathcal{S}}_i \cap \{\lambda = 0\}$ (see the proof of Proposition 2.1). For statements (ii) and (iii) we simply follow the proofs of Propositions 2.2 and 2.3 using the new definitions of the components of $\mathcal{B}$. □

3.2. Arrangements of algebraic sets of higher codimension. Let $X_i = h_i^{-1}(0)$ be the m_i-dimensional zero-set of a polynomial map $h_i := (h_i^1, \ldots, h_i^{d-m_i}) : \mathbb{R}^d \to \mathbb{R}^{d-m_i}$. The distance-squared function from p to X_i has an A_k-singularity at x if and only if there exists a $(d-m_i)$-sphere with center p having $(k+1)$-point contact with X_i at x. Algebraically, the order of contact (or intersection multiplicity) between X_i and a $(d-m_i)$-sphere, with defining equations $s^1(\xi) = \cdots = s^{m_i}(\xi) = 0$, at x is equal to the dimension of the vector space

$$\mathbb{R}[\xi]/\langle h_i^1(\xi - x), \ldots, h_i^{d-m_i}(\xi - x), s^1(\xi - x), \ldots, s^{m_i}(\xi - x)\rangle.$$

It is easy to see that such a sphere has at least 2-point contact with X_i at x if its center p lies in the normal space $N_x X_i = x + \operatorname{span}\{dh_i^1(x), \ldots, dh_i^{d-m_i}(x)\}$ of X_i at x (this assumes that x is a regular point of X_i, but the algebraic definition of the intersection multiplicity above is also valid for the singular locus of X_i).

We can now define the local bifurcation set $\mathcal{E}_i$ for complete intersections X_i and give an estimate for its degree. The point p lies in the normal space of X_i at x if $x - p \in \operatorname{span}\{dh_i^1(x), \ldots, dh_i^{d-m_i}(x)\}$, which means that all $(d-m_i+1) \times (d-m_i+1)$ minors of $\binom{dh_i(x)}{x-p}$ have to vanish. Note that only m_i of these minors are independent and that each of them has degree $O(\Delta_i)$. Let $\mathcal{M}_i := (\mathcal{M}_i^1, \ldots, \mathcal{M}_i^{m_i}) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^{m_i}$ be a d-parameter family of polynomial maps, depending on the variables x and parameters p, whose component functions are such independent minors. Then $\varphi_i := (h_i, \mathcal{M}_i) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$, $(p, x) \mapsto \varphi_i(p, x)$ is a d-parameter family of polynomial maps from $\mathbb{R}^d$ to $\mathbb{R}^d$. Using the algebraic definition of the intersection multiplicity, one checks that the simple roots in x of φ_i correspond to points of X_i having 2-point contact with $(d-m_i)$-spheres with center p through x. Roots of higher multiplicity correspond to points x in which the order of contact is at least 3-point; hence we define

$$\mathcal{E}_i := \{p \in \mathbb{R}^d : \exists x : \varphi_i(p,x) = \det d\varphi_i(p,x) = 0\}.$$

The product of the degrees of these defining equations of $\mathcal{E}_i$ is at most $O(\Delta_i^{2(m_i+1)})$.

The level bifurcation set of the family of all distance-squared functions to a pair of complete intersections $X_i = h_i^{-1}(0)$ and $X_j = h_j^{-1}(0)$ of dimension m_i and m_j is given by

$$\mathcal{S}_{i,j} := \{p \in \mathbb{R}^d : \exists x, \bar{x} : \varphi_i(p,x) = \varphi_j(p,\bar{x}) = 0, \|x-p\|^2 = \|\bar{x}-p\|^2\}$$

and has degree at most $O(\Delta_i^{m_i+1}\Delta_j^{m_j+1})$. The level bifurcation set of a single set X_i is given by

$$\mathcal{S}_i := \mathrm{cl}\{p \in \mathbb{R}^d : \exists x \neq \bar{x} : \varphi_i(p,x) = \varphi_i(p,\bar{x}) = 0, \|x-p\|^2 = \|\bar{x}-p\|^2\}$$

and has degree at most $O(\Delta_i^{2(m_i+1)})$. Recall that $\mathcal{S}_i$ is the projection of the algebraic set $\tilde{\mathcal{S}}_i := \mathcal{S}_i' \cup \partial\mathcal{S}_i'$. The set $\tilde{\mathcal{S}}_i$ is the closure of the difference of two algebraic sets $U \setminus V$, where U is the zero-set of the defining equations of $\mathcal{S}_i$, omitting the inequations $x \neq \bar{x}$, and where V is defined by the equations of $\mathcal{S}_i$ and by $x = \bar{x}$. Hence, $\tilde{\mathcal{S}}_i = Z(I(U) : I(V))$ is an algebraic set of degree at most $\deg U \sim O(\Delta_i^{2(m_i+1)})$, and its projection $\mathcal{S}_i$ is a semi-algebraic subset of an algebraic set of degree $O(\Delta_i^{2(m_i+1)})$.

Next recall that the boundary $\mathcal{S}_i'$ of $\tilde{\mathcal{S}}_i$ is contained in the diagonal $E := \{x = \bar{x}\} \subset \mathbb{R}^{2d}$. The subspace E is linear which implies that $\deg \tilde{\mathcal{S}}_i \cap E = \deg \tilde{\mathcal{S}}_i$ and that $\mathcal{S}_i' \subset \tilde{\mathcal{S}}_i \cap E$ has degree at most $O(\Delta_i^{2(m_i+1)})$.

Finally, note that the number of critical points of the distance-squared function from some fixed point $p \in \mathbb{R}^d \setminus \mathcal{E}_i$ is finite and bounded above by the degree of the map φ_i, which is $O(\Delta_i^{m_i+1})$. Summing up, we have the following proposition.

PROPOSITION 3.2. *Let $\mathcal{B}$ denote the bifurcation set of the family of distance-squared functions on a collection $X = \cup_{i=1}^n X_i$ of algebraic sets of maximal degree Δ and maximal dimension m. Then the following holds:* (i) *the degree of $\cup\partial\mathcal{S}_i'$ is at most of order $O(\Delta^{2(m+1)})$;* (ii) *the number of critical points of the distance-squared function from any point $p \in \mathbb{R}^d \setminus \cup\mathcal{E}_i$ to X is at most of order $n \cdot \Delta^{m+1}$; and* (iii) *the number of connected regions of $\mathbb{R}^d \setminus \mathcal{B}$ and the size of $\mathcal{A}(\mathcal{B})$ are at most of order $n^{2d} \cdot \Delta^{2(m+1)d}$*

Remarks. 1. Note that the estimates i, ii, and iii yield in the case of hypersurfaces $(m = d-1)$ the same estimates as in Proposition 3.1 (i), (ii), and (iii).

2. The estimate (i) implies for arrangements of plane curves (where $m = 1$) that there are at most $O(n \cdot \Delta^4)$ endpoints of the level bifurcation set. But, again using the fact that these endpoints correspond to curvature extrema of the curves X_i, one checks that actually there are at most $O(n \cdot \Delta)$ such endpoints.

3. It is also interesting to compare these estimates for arrangements of zero-sets with the corresponding bounds in the special case of parametrized m_i-surfaces given in section 2. Not surprisingly, the combinatorial complexities (fixing the degrees Δ or δ) are the same. However, in terms of algebraic complexity, the estimates in Propositions 2.1, 2.2, and 2.3 for arrangements of parametrized surfaces are sharper than the corresponding ones in Proposition 3.2 This can be seen using the following fact.

LEMMA 3.3. *The degree Δ_i of a parametrized m_i-surface X_i given by*

$$x \mapsto X_i(x) := (X_i^1(x), \dots, X_i^d(x)), \quad \delta_i := \sup_j \deg X_i^j$$

is of order $\delta_i^{m_i}$ (which implies, for arrangements of such surfaces, that $\Delta \sim O(\delta^m)$).

Proof. Let $\mathcal{L}$ be a $(d - m_i)$-dimensional linear subspace of $\mathbb{R}^d$ not contained in X_i, and let $\mathcal{L}$ be given as zero-set of some linear map $L = (L_1, \dots, L_{m_i}) : \mathbb{R}^d \to \mathbb{R}^{m_i}$. By Bézout's theorem, $L \circ X_i : \mathbb{R}^{m_i} \to \mathbb{R}^{m_i}$ has at most $\delta_i^{m_i}$ roots (counting multiplicities, complex roots, and roots at infinity), hence $|\mathcal{L} \cap X_i| \sim O(\delta_i^{m_i})$. $\square$

4. Determining the connected regions of $\mathbb{R}^d \setminus \mathcal{B}$, and applications to proximity queries. This section consists of two parts: in subsection 4.1 we sketch the exact symbolic computation of connected regions of $\mathbb{R}^d \setminus \mathcal{B}$ of constant description size for arrangements of algebraic sets defined by polynomials with rational coefficients. And in subsection 4.2 we discuss how this partition of $\mathbb{R}^d$ can be used to efficiently answer proximity queries for points with algebraic number coordinates. In terms of combinatorial complexity (where the degrees and coefficient sizes of the defining polynomials of the algebraic sets are bounded by some constant), computing the partition takes $O(n^{4d-6+\epsilon})$ (for $d \geq 3$) or $O(n^{4+\epsilon})$ (for $d = 2$) expected time (here ϵ is some small positive constant), and answering a proximity query takes $O(\log n)$ time. In dimensions 2 and 3, the time for computing the partition almost matches the number of regions of $\mathbb{R}^d \setminus \mathcal{B}$ (in the worst case), but in higher dimensions the computation time is much larger than the number of regions. In section 6 we shall study certain partitions, cut out by subsets of $\mathcal{B}$, which have a lower combinatorial complexity but can still be used for the same proximity problems.

4.1. Determining the partition. The bifurcation set $\mathcal{B}$ of the family of distance-squared functions between points $p \in \mathbb{R}^d$ and a collection of algebraic sets is a semi-algebraic set which is the projection of a real algebraic set $\tilde{\mathcal{B}} \subset \mathbb{R}^d \times \mathbb{R}^a$, where $a \leq 2m$ (for parametrized m_i-surfaces, $m := \sup m_i$) or $a = 2d$ (for zero-sets). The defining equations of the components $\tilde{\mathcal{E}}_i$, $\tilde{\mathcal{S}}_i$, $\tilde{\mathcal{S}}_{ij}$ of $\tilde{\mathcal{B}}$ are described in sections 2 and 3 and are polynomials with rational coefficients (we assume that the X_i are defined by polynomials over $\mathbb{Q}$). If π denotes the restriction to $\tilde{\mathcal{B}}$ of the obvious projection from $\mathbb{R}^d \times \mathbb{R}^a$ to $\mathbb{R}^d$ and if $\hat{\mathcal{B}} \subset \mathbb{R}^d$ is a closed real algebraic set containing $\mathcal{B}$, then we have the following set-up for the algorithm below (which consists of three steps):

$$\begin{tikzcd}
\tilde{\mathcal{B}} \arrow[d, "\pi"] & \subset \mathbb{R}^d \times \mathbb{R}^a \\
\mathcal{B} & \subset \hat{\mathcal{B}} \subset \mathbb{R}^d
\end{tikzcd}$$

1. Eliminate $x^1, \dots, x^a$ between the defining equations of $\tilde{\mathcal{B}}$. Result: the defining equations of the real algebraic set $\hat{\mathcal{B}} \subset \mathbb{R}^d$.
2. Decompose $\mathbb{R}^d$ into connected regions (of constant combinatorial complexity) such that each such region lies in a single component of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ (and hence of $\mathbb{R}^d \setminus \mathcal{B}$).
3. Optional step: determine the connected regions of $\mathbb{R}^d \setminus \mathcal{B}$ by deleting the "branches" of $\hat{\mathcal{B}} \setminus \mathcal{B}$ from $\hat{\mathcal{B}}$.

In these steps we use known techniques; here we discuss their complexity and give some references.

Step 1. The set $\tilde{\mathcal{B}}$ has $2n + \binom{n}{2}$ components; the combinatorial complexity of the elimination is therefore $O(n^2)$. Next, we consider the algebraic complexity. Recall from section 3.2 that, for zero-sets X_i of codimension ≥ 2, the generators of the ideals $I(\tilde{\mathcal{S}}_i)$ have to be precomputed from certain ideal quotients. In all other cases the defining equations of the components of $\tilde{\mathcal{B}}$ are already known. For parametrized surfaces one has to eliminate m_i (for $\tilde{\mathcal{B}}_l = \tilde{\mathcal{E}}_i$), $2m_i$ (for $\tilde{\mathcal{B}}_l = \tilde{\mathcal{S}}_i$), or $m_i + m_j$ variables (for $\tilde{\mathcal{B}}_l = \tilde{\mathcal{S}}_{i,j}$). During the elimination, which can use either multipolynomial resultants (see, for example, [7]) or Gröbner bases, one can remove repeated factors, because the later steps of the algorithm only require information about the radicals of the elimination ideals $I(\hat{\mathcal{B}}_l) := I(\tilde{\mathcal{B}}_l) \cap \mathbb{Q}[p]$. (Note that the worst-case computation time is $D^{O(v)}$ for the multipolynomial resultant and $D^{2^{O(v)}}$ for Gröbner bases, where $D \leq \delta$ or Δ is the maximal degree of the input polynomials and $v = d + a$ the number of variables.)

Step 2. We mention two algorithms in (i) and (ii) below which can be used to compute a partition of $\mathbb{R}^d$ into connected regions, of constant combinatorial complexity, in the complement of $\hat{\mathcal{B}}$. The first is the classical one and has been used to compute the examples shown in section 5; the second is more efficient. Recall that $\hat{\mathcal{B}}$ is the union of the zero-sets of $N \sim O(n^2)$ polynomials of maximal degree $D \sim O(\delta^{2m+1})$ (for parametrized X_i) or $O(\Delta^{2(m+1)})$ (for zero-sets X_i).

(i) The cylindrical algebraic decomposition of Collins [9] yields at most $(ND)^{2^d}$ d-cells in the complement of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$. The cells are diffeomorphic to open d-cubes (so that the number of lower dimensional cells in their closure is independent of N) and can be determined in $L^3(ND)^{2^d}$ time (where L denotes the maximal coefficient size of the input polynomials).

(ii) Chazelle et al. [8] (see also [22, Theorem 8.23]) describe a stratification which yields d-cells in the complement of $\hat{\mathcal{B}}$ whose closure contains a number of lower dimensional cells which does not depend on N. Assuming that the maximal degree D and the bit lengths of all polynomials arising during the computation are bounded by some constant, this stratification consists of at most $O(N^{2d-3+\epsilon})$ (for $d \geq 3$) or $O(N^{2+\epsilon})$ (for $d = 2$) cells which can be determined deterministically in $O(N^{2d+1})$ time or by a randomized algorithm in $O(N^{2d-3+\epsilon})$ (for $d \geq 3$) or $O(N^{2+\epsilon})$ (for $d = 2$) expected time (here ϵ denotes an arbitrarily small positive constant). Furthermore, given some point $p \in \mathbb{R}^d$, the cell containing p can be determined in $O(\log N)$ time. The drawback of this stratification procedure is that, considering the degree D as a variable, the number of cells and the running time become doubly exponential in d with base D.

Remark. The algorithm of Grigor'ev and Vorobjov [14] can be used to find the representative points of a partition of $\mathbb{R}^d \setminus \hat{\mathcal{B}}$ into connected regions, and this algorithm produces at most $(ND)^{d^2}$ such points in $L^{O(1)}(ND)^{O(d^2)}$ time. But it is not clear whether the number of lower-dimensional cells in the closure of each of these regions is independent of N (also, some extra work would be required to compute the region boundaries).

Step 3. The algorithms in Rieger [19, 20] can be adapted to determine the connected regions in the complement of $\mathcal{B}$. The adapted algorithm is based on a very coarse "stratification" (in which the strata can have singularities) of $\hat{\mathcal{B}}$ consisting of $O(n^2) \cdot P$ "branches" (where P is a polynomial in the degrees of the X_i) such that each "branch" either lies entirely in $\mathcal{B}$ or in $\hat{\mathcal{B}} \setminus \mathcal{B}$. Picking a "good" sample point q in each "branch," one can count the number k of real roots of the specialization $I(\tilde{\mathcal{B}})_{p=q}$

(using results from real algebra): if $k > 0$, then the "branch" belongs to $\mathcal{B}$, otherwise we delete it. The running time of this procedure is quadratic in n and polynomial in the remaining parameters. However, a region in the complement of $\hat{\mathcal{B}}$ could have up to $O(N) = O(n^2)$ "branches" of $\hat{\mathcal{B}}$ in its closure, and in the proximity queries discussed next it is important that the regions have a constant number of cells in their closures.

4.2. Answering proximity queries. We now discuss how the decompositions above can be used to answer proximity queries exactly (i.e., without numerical errors). Let $p \in \mathbb{R}^d$ be a point whose coordinates $(\alpha^1, \dots, \alpha^d)$ are algebraic numbers, represented by minimal polynomials $m^j(t) = 0$ and isolating intervals with rational endpoints. Given p and a set of defining polynomials of X with rational coefficients we would like to do the following:

1. find the k nearest sets X_i;
2. find the nearest point in X;
3. and, provided that X is compact, find the farthest point in X (and hence the smallest sphere with center p enclosing X).

For all three problems we first decompose $\mathbb{R}^d$ into regions which lie in a single connected component in the complement of $\cup\mathcal{S}_{i,j}$ (or $\mathcal{B}$ or $\hat{\mathcal{B}}$—the latter two possibilities yield finer decompositions but with the same "leading term" with respect to the asymptotic complexity in the number of regions). For problem 1 we store for each region the k nearest X_i (for any sample point in the region), for problem 2 the nearest X_i, and for problem 3 the farthest. This completes the preprocessing.

Next, given p, we use the algorithm in [8] to find the region containing p. This takes $O(\log n)$ time, assuming that the degrees and coefficient sizes of the defining polynomials of X and of the minimal polynomials m^j are bounded by some constant. This solves problem 1 already. In the case of problem 2 (respectively, 3) we now know the set X_i in the arrangement that contains the nearest (respectively, farthest) point $q \in X$ from p.

The only remaining task, then, is to determine the coordinates $(\beta^1, \dots, \beta^d)$ of the point $q \in X_i$. The coordinates are algebraic numbers, and we want to compute their minimal polynomials and isolating intervals. We know from sections 2 and 3 that the critical points of the distance-squared function from p to X_i are either the real roots of $df_i^1(p, x) = \cdots = df_i^{m_i}(p, x) = 0$, where $df_i^j \in \mathbb{Q}(\alpha^1, \dots, \alpha^d)[x]$, (for parametrized m_i-surfaces X_i) or of $\varphi_i^1(p, x) = \cdots = \varphi_i^d(p, x) = 0$, where $\varphi_i^j \in \mathbb{Q}(\alpha^1, \dots, \alpha^d)[x]$, (for zero-sets X_i). The real roots $\xi_1, \dots, \xi_s$ of these systems are isolated and have algebraic coordinates whose minimal polynomials and isolating intervals can be determined by computing a diagonal basis of the systems and by isolating the roots of univariate polynomials with algebraic coefficients (using primitive element methods and the modified Uspenski algorithm). For parametrized surfaces X_i we have to determine the root ξ_j, $1 \le j \le s$, for which the algebraic number $f_i(p, \xi_j)$ is minimal (problem 2) or maximal (problem 3); and the result is $q = X_i(\xi_j)$, whose coordinates are algebraic numbers. For zero-sets X_i we have to determine the ξ_j for which $\|\xi_j - p\|^2$ is minimal or maximal; and the result is $q = \xi_j$.

Remarks. 1. The decomposition of d-space into connected regions in the complement of $Y := \cup\mathcal{S}_{i,j}$ is finer than it needs to be for answering the above queries. In section 6 we study certain subsets R_k and $\tilde{R}_k$ of Y which cut out far fewer regions (see

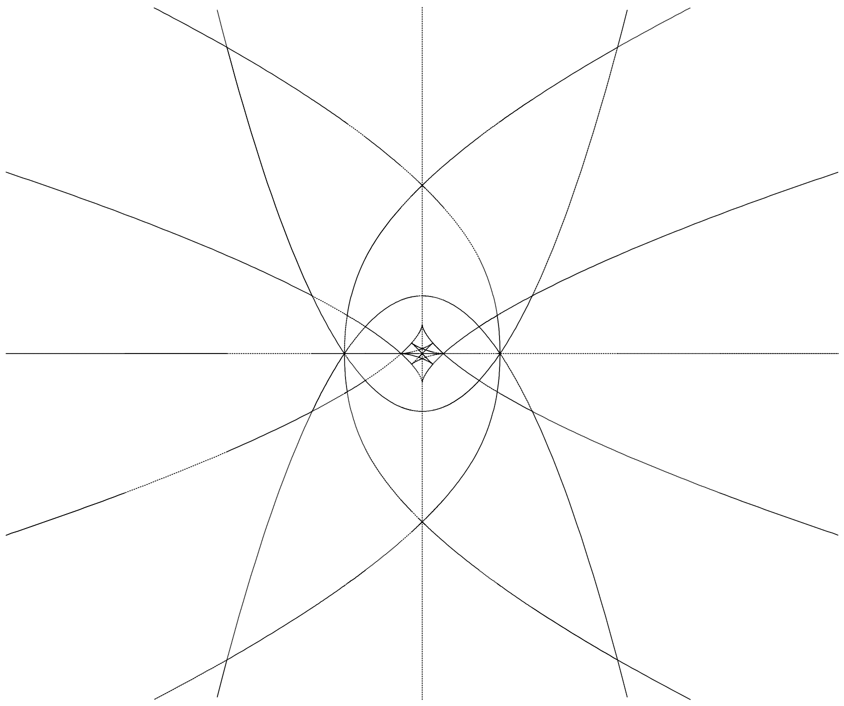


FIG. 2. *The set $\hat{\mathcal{B}}$ for a pair of parabolas.*

Proposition 6.1 for precise statements). Using the above exact symbolic methods, one can replace Y by the following sets: in problem 1 by R_k, in problem 2 by R_1, and in problem 3 by $\tilde{R}_1$.

2. However, in problems 2 and 3, if one replaces the exact symbolic computation of the nearest and farthest point $q \in X$ from p by the numerical curve-tracing procedure sketched below, then the decomposition into regions of $\mathbb{R}^d \setminus Y$ is too coarse, and one must replace Y by the full bifurcation set $\mathcal{B}$! The numerical procedure consists of the following: determine one sample point p' for each region in the complement of $\mathcal{B}$ (or $\hat{\mathcal{B}}$) and the corresponding nearest or farthest point $q' \in X$. After determining the region containing p (as before), one knows that any path in this region joining its sample point p' with p corresponds to a unique path in X joining the corresponding nearest/farthest points q' and q. This follows from the fact that the critical points of the distance-squared function are isolated in the complement of $\mathcal{B}$ and the continuity of the map which assigns to p its nearest/farthest point in X (Proposition 2.3). This would not be the case if we replace $\mathcal{B}$ by Y.

5. Some examples for arrangements in the plane. The first example, in Figure 2, shows a pair of parabolas $X_1(x) = (x, x^2 - 1)$, $X_2(x) = (x, 1 - x^2)$ together with the set $\hat{\mathcal{B}}$ which contains the bifurcation set $\mathcal{B}$. It should be noted that most curves in Figure 2, except for the intercurve level bifurcation set, are already known from Figure 1: the first parabola X_1 and the sets $\hat{\mathcal{E}}_1 = \mathcal{E}_1$ (a cusp-shaped curve) and $\hat{\mathcal{S}}_1$ (a vertical line) are exactly as shown in Figure 1, and turning Figure 1 upside down yields X_2, $\mathcal{E}_2$, and $\hat{\mathcal{S}}_2$ ($\hat{\mathcal{S}}_2$ coincides with $\hat{\mathcal{S}}_1$). The set $\hat{\mathcal{S}}_{1,2}$, which contains the intercurve level bifurcation set $\mathcal{S}_{1,2}$ (but we do not know whether it is equal to it), consists of a horizontal line through the origin and the zero-set Z of an irreducible

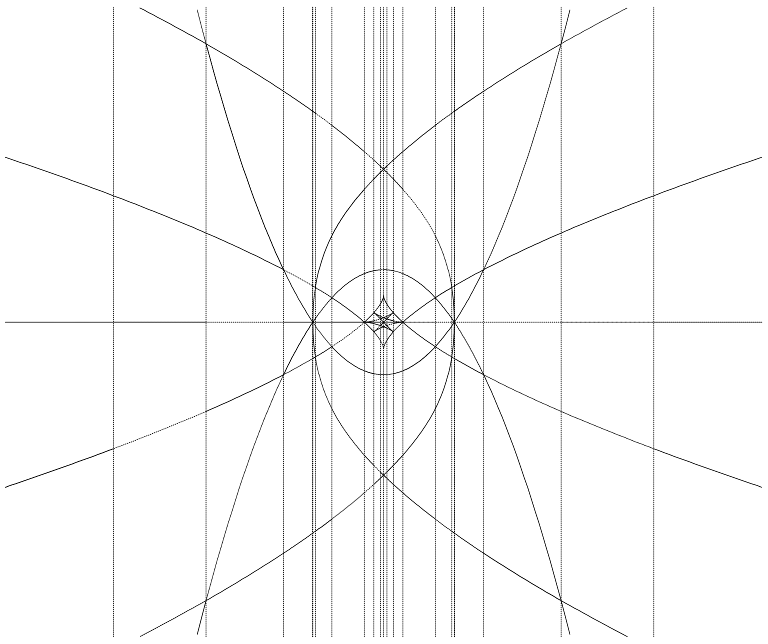


FIG. 3. *Cylindrical algebraic decomposition of $\hat{\mathcal{B}}$ and a pair of parabolas.*

(over $\mathbb{Q}$) degree 12 polynomial. The set Z has three real components: a compact curve with 6 cusps and a pair of nonsingular curves passing through the intersection points of the parabolas. Figure 3 shows a cylindrical algebraic decomposition of the plane into regions in the complement of $\mathbb{R}^2 \setminus \hat{\mathcal{B}} \cup X_1 \cup X_2$, which are arranged in "vertical cylinders." The cylinders are bounded by vertical tangent lines or by vertical lines passing through singular points. The regions within a cylinder $I \times \mathbb{R}$, where I is an interval on the x-axis, are separated by nonintersecting function graphs over I.

The second example, in Figure 4, shows the set $\hat{\mathcal{S}}_{1,2}$ for a parabola $X_1(x) = (x, x^2)$ and a point $X_2 = (1, 2)$. Note that the Voronoi diagram of X_1, X_2 consists of just two regions: the region cut out by $\hat{\mathcal{S}}_{1,2}$ containing the point X_2 and the complement of the closure of this region. The curve $\hat{\mathcal{S}}_{1,2}$ has 2 cusps, which correspond to centers of osculating circles of X_1 which pass through the point X_2. Figure 5 illustrates this fact: the cusps of $\hat{\mathcal{S}}_{1,2}$ lie on the evolute $\mathcal{E}_1 = \hat{\mathcal{E}}_1$ of the parabola X_1 (and hence are centres of osculating circles having A_2-contact with X_1).

6. Regions of $\mathbb{R}^d \setminus \mathcal{B}$ and Voronoi regions.

The estimates in sections 2 and 3 for the size of $\mathcal{A}(\mathcal{B})$ (in terms of n, d, and the degrees and dimensions of the X_i) yield upper bounds for the complexity of the kth-order Voronoi diagram. They also bound the complexity of Voronoi diagrams of collections $X^s := \cup_i X_i^s$ of closed semialgebraic sets $X_i^s \subset \mathbb{R}^d$, each given as unions and intersections of $O(1)$ "elementary" sets of the form $\{x \in \mathbb{R}^d : h(x) \ ?\ 0\}$, where $? \in \{=, <, >, \leq, \geq\}$ (actually, any such X_i^s can be defined by choosing $? \in \{=, <\}$). For each X_i^s we can define a real algebraic set $X_i \supset X_i^s$, using the $O(1)$ polynomials h, such that $\dim X_i = \dim X_i^s$. If $\mathcal{B}^s$ and $\mathcal{B}$ are the bifurcation sets of X^s and $X := \cup_i X_i$, then $\mathcal{B}^s \subset \mathcal{B}$; in fact $\mathcal{B} \setminus \mathcal{B}^s$ consists of the closure of certain $(d-1)$-cells of $\mathcal{B}$. The size of the arrangement $\mathcal{A}(\mathcal{B}^s)$ (and hence of the kth-order Voronoi diagram of X^s) is therefore bounded above by $|\mathcal{A}(\mathcal{B})|$.

A comparison of the bounds for the size of $\mathcal{A}(\mathcal{B})$ and of Voronoi diagram $\mathcal{A}(V_k)$

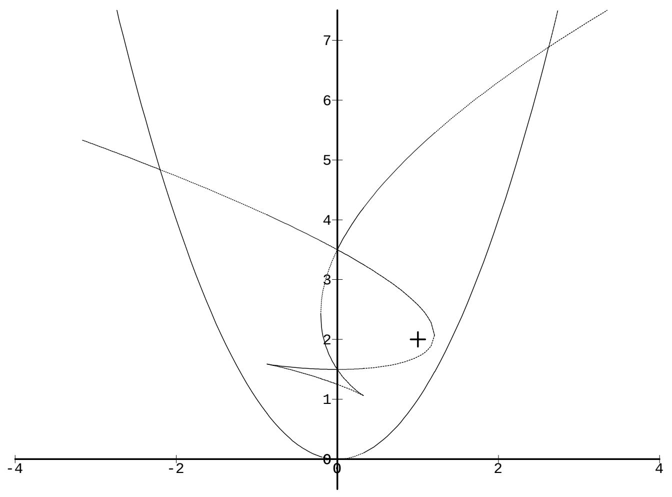


FIG. 4. *The set $\hat{\mathcal{S}}_{1,2}$ for a parabola X_1 and the point $X_2 = (1, 2)$ (marked by a cross).*

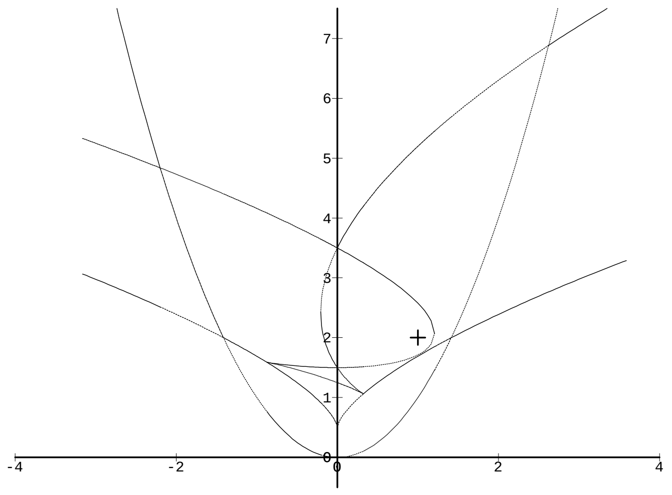


FIG. 5. *The sets $\hat{\mathcal{S}}_{1,2}$ and $\hat{\mathcal{E}}_1$ for a parabola X_1 and the point $X_2 = (1, 2)$.*

reveals, however, a considerable gap—at least in the special cases where something about the complexity of the Voronoi diagram is known. The works on Voronoi diagrams of arrangements of (semi-)algebraic sets study the combinatorial complexity (i.e., assume that the degrees of the algebraic sets in an arrangement are bounded above by some constant). In this case, $|\mathcal{A}(\mathcal{B})| \sim O(n^{2d})$. On the other hand, Sharir and Agarwal show in [22, Appendix 7.1] that the size of the first-order Voronoi diagram of n disjoint convex semialgebraic sets of "constant description size" (i.e., defined by $O(1)$ polynomial (in)equations of bounded degree and coefficient size) in d-space is $O(n^{d+\epsilon})$ (for any $\epsilon > 0$). And Alt and Schwarzkopf [1] show that the first-order Voronoi diagram of n points and disjoint parametrized algebraic curve segments in the plane, which also do not have self-intersections, has $O(n)$ size and can be constructed by a randomized algorithm in $O(n \log n)$ (expected) time. Their algorithm concentrates on the combinatorial aspect of the problem and assumes that the semialgebraic level bifurcation sets $\mathcal{S}_i$ and $\mathcal{S}_{i,j}$ (in our notation) can be determined by some numerical polynomial equation solver. We have seen in previous sections that the bifurcation set can also be determined by exact symbolic methods.

On the other hand, the first-order Voronoi diagram of the $dn \sim \Theta(n)$ intersecting hyperplanes $X_{ij} := \{x^i = j\}$, where $1 \leq i \leq d$ and $1 \leq j \leq n$, has $\Theta(n^d)$ connected d-dimensional regions (recall that $(x^1, \ldots, x^d)$ are coordinates in $\mathbb{R}^d$). Hence we have the lower bound $|\mathcal{A}(V_1)| \sim \Omega(n^d)$.

The goal of the present section, then, is to study the gap in the combinatorial complexities of $\mathcal{A}(\mathcal{B})$ and of $\mathcal{A}(V_k)$. To this end, we shall derive a bound for the combinatorial complexity of certain intermediate sets $V_k \subset R_k \subset \mathcal{B}$, which we are going to define next.

Roughly speaking, the sets R_k are constructed by deleting from the intersurface level bifurcation set $Y := \cup_{1 \leq i < j \leq n} \mathcal{S}_{i,j} \subset \mathcal{B}$ certain "branches" that cannot belong to V_k. In order to describe this construction, we need the following notation. Recall the notation for an $A^r_{\geq 1}$-singularity of the distance squared-function (see section 1.1). For a family of distance-squared functions $f(p, x)$ parametrized by the coordinates of the points $p \in \mathbb{R}^d$, we denote the set of points p for which f has a singularity of type $A^r_{\geq 1}$ by $\mathcal{L}(A^r_{\geq 1})$. Geometrically, it is the locus of centers of r-tangent spheres (i.e., spheres touching the algebraic set X in r distinct points). Now set

$$Y_r := Y \cap \mathcal{L}(A^r_{\geq 1}).$$

Furthermore, let $Y_{r_1,\ldots,r_s}$ denote the locus of common intersection points of s such sets Y_{r_i}. In order to avoid redundancies, let us agree that the indices are nonincreasing, i.e., $r_i \geq r_{i+1}$. It is also convenient to define the "closure" of $Y_{r_1,\ldots,r_s}$ as

$$\bar{Y}_{r_1,\ldots,r_s} := \bigcup \{Y_{a_1,\ldots,a_t} : t \geq s, a_i \geq r_i, 1 \leq i \leq s\}. \qquad (*)$$

Note that the points p of the s-fold self-intersection locus of $Y_{r_1,\ldots,r_s}$ of $Y = \bar{Y}_2$ are centers of s concentric r_i-tangent spheres (where $r_i \geq 2$). Also note that $p \in \bar{Y}_{r_1,\ldots,r_s} \setminus Y_{r_1,\ldots,r_s}$ if and only if there are more than s such spheres or some sphere has more than r_i points of tangency. To save breath, we shall often refer to the "k smallest r_i-tangent spheres" with common center $p \in Y_{r_1,\ldots,r_s}$, $k \leq s$, rather than to the "subset of the set of s simultaneous $A^r_{\geq 1}$-singularities with the k smallest critical values." Finally, let $S_{c_1,\ldots,c_t}$ be the locus of common intersections of t sets X_i of codimension $c_i := d - m_i$, which corresponds to the intersection locus of $\binom{t}{2}$ branches of Y, and define its "closure" $\bar{S}_{c_1,\ldots,c_t}$ as in $(*)$. Each branch consists of centers of spheres tangent to a pair of intersecting sets X_i, X_j whose radius tends to zero as the center approaches $X_i \cap X_j$, such spheres will be called *vanishing spheres*. We are now ready to construct the sets R_k.

First, we decompose the intersurface level bifurcation set Y into certain branches which, for generic arrangements X, will be $(d-1)$-dimensional. Let $B(Y)$ denote the set of connected components (branches) of $Y \setminus \bar{Y}_3 \cup \bar{S}_{1,1}$. Note that all points of such a branch lie either in $V_k \subset Y$ or in $Y \setminus V_k$, because for all these points we have a pair of critical points of the distance-squared function whose critical value is distinct from all other critical values.

Next, we decompose the self-intersection locus of Y into connected components of i-fold intersections, $i = 2, 3, \ldots, s$ and compare the radii of ≥ 2-tangent spheres associated to the i branches of $B(Y)$ passing through an i-fold intersection point. Let us call the set of $X_j \in \{X_1, \ldots, X_n\}$ containing the r points of tangency of an r-tangent sphere the *support set* of this sphere and the smallest sphere among a set of concentric r-tangent spheres with the same support set the *minimal sphere*. We also consider any vanishing sphere to be minimal. If, at any point p of the self-intersection locus, the ≥ 2-tangent sphere associated with some branch of $B(Y)$ does not belong to the k smallest minimal spheres with center p (including the vanishing sphere if $p \in S_{c_1,\ldots,c_t}$) and distinct support sets, then this branch cannot belong to V_k. Deleting all such branches from Y yields the set R_k. To be a bit more precise, let L be the set of "strata" of the "stratification" (the reason for the quotes will be explained in the remark below) of the self-intersection locus of Y into connected components of $Y_{r_1,\ldots,r_s}$ $(s, r_i \geq 2)$, $S_{c_1,\ldots,c_t}$ $(t \geq 2)$, and $S_{c_1,\ldots,c_t} \cap Y_{r_1,\ldots,r_s}$ $(t, r_i \geq 2,\ s \geq 1)$. For any $l \in L$, let l_k denote the set of branches $b \in B(Y)$ passing through l which correspond to the k smallest minimal $\geq$ 2-tangent spheres with center in l, which by definition have distinct support sets (if there are fewer than k minimal spheres with distinct radius, then l_k contains all branches through l that correspond to some minimal sphere). We can now define

$$R_k := \{b \in B(Y) : b \in l_k, \text{for all } l \in L : l \subset \text{cl} b\} \cup \bar{Y}_3 \cup \bar{S}_{1,1}.$$

The principal result of the present section is based on an enumeration of the connected components of the self-intersection locus of Y, on the one hand, and of those components that also belong to R_k, on the other hand. We give an outline of our enumeration technique. Given an arrangement $X = \cup_{i=1}^n X_i$, there are $\Pi_{i=1}^s \binom{n}{r_i}$ sets $\bar{Y}_{r_1,\ldots,r_s}$, and each of them has a constant number of connected components (recall that the maximal degree of the defining equations of X and the ambient dimension are assumed to be fixed). The number of connected components of $Y_{r_1,\ldots,r_s}$ depends on the number of connected components of all the (lower dimensional) sets

$$Y_{a_1,\ldots,a_t} \subset \bar{Y}_{r_1,\ldots,r_s} \setminus Y_{r_1,\ldots,r_s}$$

in its boundary. For "large enough" s and $r_1, \ldots, r_s$ (see more precise statements below), the boundary of $Y_{r_1,\ldots,r_s}$ will be empty, so that $Y_{r_1,\ldots,r_s}$ has as many connected components as $\bar{Y}_{r_1,\ldots,r_s}$. We call a connected component of such a nonempty set $Y_{r_1,\ldots,r_s}$, whose boundary is empty, a *maximal component*, and $Y_{r_1,\ldots,r_s}$ a *maximal set*. (Note that its index set $r_1, \ldots, r_s$ is maximal, with respect to the natural partial order of $\mathbb{N}^s$, among the nonempty sets $Y_{a_1,\ldots,a_s}$. Among these nonempty sets, however, it will be the one with minimal dimension.) Likewise, the combinatorial complexity of the closures of the sets $S_{c_1,\ldots,c_t}$ and $S_{c_1,\ldots,c_t} \cap Y_{r_1,\ldots,r_s}$ is $O(n^t)$ and $O(n^{t+\Sigma r_i})$, respectively, and the complexity of the interiors of these sets will depend on the number of components in their boundary (for Σr_i and Σc_j sufficiently large we get, again, maximal sets with empty boundary). By inductively deleting the lower dimensional boundary components from $Y = \bar{Y}_2$, beginning with the maximal components, whose boundary is empty, we obtain a "stratification" of Y whose "strata" are the connected components of the sets $Y_{r_1,\ldots,r_s}$, $S_{c_1,\ldots,c_t}$ and their intersections. The number of strata obtained in this way is of the order of the number of maximal sets (recall that a given maximal set has $O(1)$ connected components, in terms of combinatorial complexity).

Remarks (refining stratification to get genuine stratification). 1. First, the rough idea. The strata of this stratification of Y can have singularities along the intersection of Y with the local bifurcation set $\mathcal{E} := \cup\mathcal{E}_i$. For example, the intersurface level bifurcation curve Y in Figure 2 has a component C with six cusps contained in $Y \cap \mathcal{E}$ (C is the small compact curve in the center of the figure). So what is going on here? The strata of C are connected components of $Y_{r_1,\ldots,r_s}$ and correspond to points $p \in \mathbb{R}^d$ for which the distance-squared function to X has s simultaneous singularities of type $A^{r_i}_{\geq 1}$. But any r-tuple of singular points with the same critical values belongs to $A^{r}_{\geq 1}$: for example, the regular branches of C in Figure 2, which are of type $A_1^2 = \{A_1, A_1\}$, but also the cusps, which are of type $\{A_2, A_1\}$. Roughly speaking, one can further subdivide the components (i.e., the strata) of Y into submanifolds (i.e., genuine strata) by requiring that its singular points $\Sigma_1, \ldots, \Sigma_{r_i}$, $1 \leq i \leq s$, are of the same local type (for example, we distinguish A_1 from A_2 points). For each original stratum we then obtain $O(1)$ genuine "equisingular" strata (the number of local types on a given stratum depends on the degree of the X_i and the ambient dimension, but not on n). With this understood, we shall no longer make the distinction between a stratification and its genuine refinement.

2. The meaning of equisingular (for readers familiar with singularity theory). We are considering, in general, nonversal d-parameter families of functions. As d increases, the standard groups of equivalences for functions, such as $\mathcal{R}$ and $\mathcal{K}$, will quickly yield an infinite number of equisingular strata (due to the appearance of moduli). Recall that a $\mathcal{K}$-orbit consists, in general, of several $\mathcal{R}$-orbits; we can only expect for the class of quasi-homogeneous functions that the $\mathcal{K}$- and $\mathcal{R}$-orbits coincide (by a result of Saito [21]). The appropriate definition of equisingular stratum, which yields a finite number of smooth strata, therefore involves the union of $\mathcal{K}$-modular strata (minus certain exceptional strata of higher codimension).

We can now state the main result of section 6.

PROPOSITION 6.1. *For any arrangement of parametrized or implicitly defined algebraic sets (whose degrees are bounded by some constant) in d-space consisting of n elements of any positive codimension the following hold:*

(i) $V_k \subset R_k \subset Y \subset \mathcal{B}$, $1 \leq k \leq n-1$.

(ii) *The size of* $\mathcal{A}(R_k)$ *and hence the number of connected regions of* $\mathbb{R}^d \setminus R_k$ *are bounded above by* $O(\min(n^{d+k}, n^{2d}))$.

(iii) *The combinatorial complexity of* $Y_{2,\ldots,2}$ *(d twos) is* $\Pi_{j=1}^{d}\binom{n}{2} \sim O(n^{2d})$ *and represents the "leading term" in the combinatorial complexity of* $\mathcal{A}(\mathcal{B})$.

Proof. Statement (i) simply follows from the definitions of these sets. For the proof of statements (ii) and (iii) it is convenient to distinguish generic and nongeneric arrangements X, which are defined as follows. Let $\mathcal{X}$ be the space of arrangements $X \subset \mathbb{R}^d$ of n zero-sets X_i of codimension c_i of maximal degree Δ (or of n m_i-surfaces X_i parametrized by polynomials ofdegree $\leq \delta$). $\mathcal{X}$ can be identified with some semialgebraic subset of the finite dimensional space of coefficients of $\sum_{i=1}^n c_i$ polynomials in d variables of degree $\leq \Delta$ (or of nd polynomials in $\sum_{i=1}^n m_i$ variables of degree $\leq \delta$). (Note that not all choices of coefficients yield m_i-dimensional real algebraic sets X_i.) Now define W to be the union of the following sets (corresponding to degenerate X for which Y has "excess intersection"):

$$\left\{X \in \mathcal{X} : \exists s \geq 1, \exists r_i \geq 2 : \dim Y_{r_1,\dots,r_s} > d + s - \sum_{i=1}^{s} r_i\right\},$$

$$\left\{X \in \mathcal{X} : \exists t \geq 2, \exists c_i \geq 1 : \dim S_{c_1,\dots,c_t} > d - \sum_{i=1}^{t} c_i\right\},$$

and

$$\left\{X \in \mathcal{X} : \exists s, c_i \geq 1, \exists t, r_j \geq 2 : \dim(S_{c_1,\dots,c_t} \cap Y_{r_1,\dots,r_s}) > d + s - \sum_{i=1}^{t} c_i - \sum_{j=1}^{s} r_j\right\}.$$

(Note that X denotes both a subset of $\mathbb{R}^d$ as well as a point of $\mathcal{X}$, but the meaning of X should be clear from the context.) One shows, using the defining equations of these sets, that W is a Zariski closed subset of $\mathcal{X}$. We shall therefore say that an element X in $\mathcal{X} \setminus W$ is generic and one in W nongeneric.

First, assume that X is generic and consider the following two stratifications of $\mathbb{R}^d$. In the first, take as strata of dimension 0 to $d-1$ the connected components of the sets $Y_{r_1,\dots,r_s}$, $S_{c_1,\dots,c_t}$, $S_{c_1,\dots,c_t} \cap Y_{r_1,\dots,r_s}$ and as d-dimensional strata the connected components in the complement of $Y = Y_2$. In the second stratification, we discard the connected components of $Y_{r_1,\dots,r_s}$ that do not belong to R_k and take the connected components of $\mathbb{R}^d \setminus R_k$ as d-dimensional strata.

For the 0-dimensional maximal sets $Y_{r_1,\dots,r_s}$ we have, by the genericity of X, the relation $\Sigma_{i=1}^s r_i = d + s$. For the 0-dimensional maximal sets $S_{c_1,\dots,c_t}$ and $S_{c_1,\dots,c_t} \cap Y_{r_1,\dots,r_s}$ we have in the worst case of hypersurface arrangements (where all $c_i = 1$) the relations $t = d$ and $t + \Sigma r_i = d + s$. Hence, there are at most $\Pi_{i=1}^s \binom{n}{r_i} \sim O(n^{d+s})$ such maximal sets, and each of them has $O(1)$ connected components. For the first stratification (whose union of strata of dimension less than d is Y), the relation $\Sigma r_i = d + s$, where all $r_i \geq 2$, implies that $s \leq d$. For the second stratification (whose union of strata of dimension $< d$ is R_k) we have, by the definition of R_k, $s \leq \min(k, d)$. Let e_i denote the number of i-dimensional strata. Then, for all $0 \leq i \leq d-1$, $e_i \sim O(n^{2d})$ (for the stratification of Y) and $e_i \sim O(n^{\min(d+k,2d)})$ (for the stratification of R_k).

For statement (ii) of the proposition we now consider the second stratification. We claim that the number e_d of connected regions of $\mathbb{R}^d \setminus R_k$ is also $O(n^{\min(d+k,2d)})$. Taking a 1-point compactification $S^d = \mathbb{R}^d \cup \{\infty\}$ and adding at most $O(n^{\min(d+k,2d)})$ cells to the induced stratification of R_k in the d-sphere, we get a cell complex K whose Euler characteristic is equal to $\chi(S^d) = 1 + (-1)^d$. Note that e_d is bounded above by the number of d-cells of K; this implies (ii).

For statement (iii), note that $Y_{2,\dots,2}$ (with d twos) is the maximal set of the highest combinatorial complexity in the stratification of Y satisfying the relation $\Sigma r_i = d + s$, and its number of connected components is of order n^{2d}.

For nongeneric arrangements $X \in W$, we consider a "linear deformation" X_t, $t \in (-\epsilon, +\epsilon)$, of $X = X_0$ such that X_0 is the only nongeneric element—linear in the sense that $t \mapsto X_t$ defines a line in the space of coefficients which can be identified with $\mathcal{X}$. (Such a deformation can be obtained, for example, by constructing a stratification of the semialgebraic set W and by restricting a line in the normal space of the stratum containing X to some sufficiently small open neighborhood.) Consider the union U of any of the semialgebraic sets $U_t = Y_t$ or $(R_k)_t$ associated to X_t; U is a semialgebraic subset of $\mathbb{R}^d \times (-\epsilon, +\epsilon)$. We claim that the combinatorial complexity of the degenerate arrangement $\mathcal{A}(U_0)$ is of the same order as that of its generic deformation $\mathcal{A}(U_t)$, for small $t \neq 0$, which implies the desired bounds in the degenerate case.

The claim follows from the following argument. First, we want to check that all strata of U_0 lie in the closure of some stratum of $U \setminus U_0$. Given a pair of closed, connected subsets $A, B \subset \mathbb{R}^d$ and any point $q \in A$ there exists a sphere tangent to A at q and to B in some point q'. Let A_t and B_t be subsets of $X_t \subset \mathbb{R}^d \times (-\epsilon, +\epsilon)$ such that A_0, B_0 are connected subsets of $X = X_0$. Our assumptions about the algebraic set X imply that the dimensions of A_t and B_t are constant for t in some open neighborhood I of 0 (in particular the sets remain nonempty over the reals). By the geometric fact above, there exist points $q_t \in A_t$ and $q'_t \in B_t$ supporting bitangent spheres with centers p_t, such that the sets $\{q_t : t \in I\}$, $\{q'_t : t \in I\}$, and $\{p_t : t \in I\}$ are connected and p_0 is any point of U_0. Next, let $\epsilon > 0$ be small enough such that U is transverse to all hyperplanes $t = c$, for any constant $|c| < \epsilon$, except $t = 0$. (U is, in general, a singular semialgebraic set, and transverse means that the hyperplane in question is transverse to all the strata of a suitable stratification of U, e.g., a stratification satisfying the Whitney condition (b). See the book by Goresky and MacPherson [13, Part I, Chapters 1.2–1.8] for a good introduction to stratification theory.) So the numbers of strata of dimension $0 \leq i \leq d$ in $\mathcal{A}(U_t)$ are locally constant for $t \in (-\epsilon, 0)$ and $t \in (0, \epsilon)$; denote these numbers by r_- and r_+, respectively. Hence there are $r_- + r_+$ strata in $\mathcal{A}(U \setminus U_0)$ and therefore at most that many strata of $\mathcal{A}(U_0)$. $\square$

Remarks. 1. For compact arrangements $X = \cup X_i$ we can define regions analogous to the usual kth-order Voronoi regions, such that for all points within each region the farthest k sets in the arrangement do not change. If $\tilde{V}_k$ is the union of the boundaries of these regions and $\tilde{R}_k$ the set analogous to R_k, except that the k smallest minimal spheres are replaced by the k largest maximal spheres, then Proposition 6.1 above holds for $\tilde{R}_k$ and $\tilde{V}_k$ in place of R_k and V_k.

2. One can get a sharper bound for the (expected) size of $\mathcal{A}(V_k)$, where $k \sim O(1)$, in the average case. Set $\mu_p(X_i) := \inf_{q \in X_i} \|q - p\|^2$. The sets $Y_{r_1,\dots,r_s} \cap R_k$ and $(Y_{r_1,\dots,r_s} \cap S_{c_1,\dots,c_t}) \cap R_k$, $s \leq k$, can only belong to V_k if the critical values of the s $A^{r_i}_{\geq 1}$-singularities of the distance-squared function from $p \in Y_{r_1,\dots,r_s}$ are smaller than all but $k - s \sim O(1)$ of the minima $\mu_p(X_j)$ of the $O(n)$ sets X_j that do not belong to the support sets of one of these $A^{r_i}_{\geq 1}$-singularities (for $p \in \cap_{i=1}^t X_i$, we include the intersecting X_i in the support set). Now suppose for the moment that there exists a "good" probability measure on the space $\mathcal{X}$ of arrangements (definition follows below) such that if we pick some $X = \cup X_i \in \mathcal{X}$ "at random," then $\Pr[\mu_p(X_i) < \mu_p(X_j)] = 1/2$ (actually it is enough if this probability is different from 0 and 1). One then checks that the probability that the critical values of all s $A^{r_i}_{\geq 1}$-singularities are smaller than $O(n)$ minima $\mu_p(X_j)$ is $O(n^{-s})$. Looking at the proof of Proposition 6.1 we now see the following: if M sets X are picked independently from some "good" distribution on $\mathcal{X}$ then, as $M \to \infty$, the expected size of $\mathcal{A}(V_k)$ is $O(n^d)$.

The "good" probability measures on $\mathcal{X}$ are defined as follows. Let $\mathcal{X}^{m_i}$ denote the space of real algebraic sets of dimension m_i of some bounded degree (recall that $\mathcal{X}^{m_i}$ can be identified with a semialgebraic subset of the space of coefficients of the defining polynomials of such sets); then $\mathcal{X} := \times_{i=1}^{n} \mathcal{X}^{m_i}$. Let $\mathcal{M}_i$ be a probability measure on $\mathcal{X}^{m_i}$ (for example, the uniform distribution on some compact subset B of $\mathcal{X}^{m_i}$ of bounded coefficients). We say that the collection of probability measures $\mathcal{M}_i$, $1 \leq i \leq n$, is "good" if the following hold. 1. For each pair $(\mathcal{M}_i, \mathcal{M}_j)$, the Lebesgue measure on $\mathcal{X}^{m_i} \times \mathcal{X}^{m_j}$ is *absolutely continuous* with respect to the product measure $\mathcal{M}_i \times \mathcal{M}_j$. 2. For any given $p \in \mathbb{R}^d$ and all ordered pairs i, j the sets $\{(X_i, X_j) : \mu_p(X_i) < \mu_p(X_j)\}$ have non-zero Lebesgue measure in $\mathcal{X}^{m_i} \times \mathcal{X}^{m_j}$. Conditions 1 and 2 imply that $\Pr[\mu_p(X_i) < \mu_p(X_j)] \neq 0, 1$, but seem quite strong. Note, however, that the conditions are trivially satisfied in an important special case: if all sets X_i in an arrangement have the same dimension and are chosen from a single, but arbitrary, distribution, then, by symmetry, $\Pr[\mu_p(X_i) < \mu_p(X_j)] = 1/2$.

Acknowledgments. I am very grateful to the referees for their detailed comments that contained several corrections and many other useful suggestions. The author acknowledges financial support from the Max–Planck Gesellschaft and the research foundation FAPESP, São Paulo, Brazil.

REFERENCES

[1] H. Alt and O. Schwarzkopf, *The Voronoi diagram of curved objects*, in Proceedings of the 11th Annual Sympos. Comput. Geometry, Vancouver, B.C., Canada, ACM, 1995, pp. 89–97.

[2] V.I. Arnol'd, *Normal forms of functions near degenerate critical points, the Weyl groups A_k, D_k, E_k, and Lagrange singularities*, Funct. Anal. Appl., 6 (1972), pp. 254–272.

[3] T. Becker and V. Weispfenning (In cooperation with H. Kredel), *Gröbner Bases. A Computational Approach to Commutative Algebra*, Springer-Verlag, New York, 1993.

[4] R. Benedetti and J.-J. Risler, *Real Algebraic and Semi-Algebraic sets*, Hermann, Paris, 1990.

[5] J.W. Bruce, *Lines, circles, focal and symmetry sets*, Math. Proc. Cambridge Phil. Soc., 118 (1995), pp. 411–436

[6] J.W. Bruce, P.J. Giblin, and C.G. Gibson, *Symmetry sets*, Proc. Roy. Soc. Edinburgh, 101A (1985), pp. 163–186.

[7] J. Canny, *Generalised characteristic polynomials*, J. Symbolic Comput., 9 (1990), pp. 241–250.

[8] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *A singly exponential stratification scheme for real semi-algebraic varieties and its applications*, Theoret. Comput. Sci., 84 (1991), pp. 77–105.

[9] G.E. Collins, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, in Proceedings of the 2nd GI Conf. Automata Theory and Formal Languages, Springer LNCS 33, Springer-Verlag, Berlin, Heidelberg, New York, 1975, pp. 134–183.

[10] A. Dimca, *Topics on Real and Complex Singularities*, Vieweg, Braunschweig, Wiesbaden, 1987.

[11] A. Dold, *Lectures on Algebraic Topology*, Springer-Verlag, New York, 1972.

[12] H. Edelsbrunner and R. Seidel, *Voronoi diagrams and arrangements*, Discrete Comput. Geom., 1 (1986), pp. 25–44.

[13] M. Goresky and R. MacPherson, *Stratified Morse Theory*, Springer-Verlag, Berlin, 1988.

[14] D.Yu. Grigor'ev and N.N. Vorobjov, *Solving systems of polynomial inequalities in subexponential time*, J. Symbolic Comput., 5 (1988), pp. 37–64.

[15] J. Milnor, *On the Betti numbers of real varieties*, Proc. Amer. Math. Soc., 15 (1964), pp. 275–280.

[16] I.R. Porteous, *The normal singularities of a submanifold*, J. Differential Geom., 5 (1971), pp. 543–564.

[17] I.R. Porteous, *The normal singularities of surfaces in $\mathbb{R}^3$*, in Singularities, Part 2, Proc. Sympos. Pure Math. 40, Providence, RI, 1983, pp. 379–393.

[18] I.R. Porteous, *Probing singularities*, in Proc. Sympos. Pure Math., Vol. 40 (1983), Part 2, American Mathematical Society, Providence, RI, pp. 395–406.

[19] J.H. Rieger, *Computing view graphs of algebraic surfaces*, J. Symbolic Comput., 16 (1993), pp. 259–272.

[20] J.H. Rieger, *On the complexity and computation of view graphs of piecewise smooth algebraic surfaces*, Phil. Trans. Roy. Soc. London Ser. A, 354 (1996), pp. 1899–1940.

[21] K. Saito, *Quasihomogene isolierte Singularitäten von Hyperflächen*, Invent. Math., 14 (1971), pp. 123–142.
[22] M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge, New York, 1995.
[23] M. Spivak, *A Comprehensive Introduction to Differential Geometry*, Vol. III, 2nd ed., Publish or Perish Inc., Berkeley, CA, 1979.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 459–473

BETTER BOUNDS FOR ONLINE SCHEDULING[*]

SUSANNE ALBERS[†]

Abstract. We study a classical problem in online scheduling. A sequence of jobs must be scheduled on m identical parallel machines. As each job arrives, its processing time is known. The goal is to minimize the makespan. Bartal et al. [*J. Comput. System Sci.*, 51 (1995), pp. 359–366] gave a deterministic online algorithm that is 1.986-competitive. Karger, Phillips, and Torng [*J. Algorithms*, 20 (1996), pp. 400–430] generalized the algorithm and proved an upper bound of 1.945. The best lower bound currently known on the competitive ratio that can be achieved by deterministic online algorithms is equal to 1.837. In this paper we present an improved deterministic online scheduling algorithm that is 1.923-competitive; for all $m \geq 2$. The algorithm is based on a new scheduling strategy, i.e., it is not a generalization of the approach by Bartal et al. Also, the algorithm has a simple structure. Furthermore, we develop a better lower bound. We prove that, for general m, no deterministic online scheduling algorithm can be better than 1.852-competitive.

Key words. makespan minimization, online algorithm, competitive analysis

AMS subject classifications. 68Q20, 68Q25, 90B35

PII. S0097539797324874

1. Introduction.

We study a classical problem in online scheduling. A sequence of jobs must be scheduled on m identical parallel machines. Whenever a job arrives, its processing time is known in advance, and the job must be scheduled immediately on one of the machines without knowledge of any future jobs. Preemption of jobs is not allowed. The goal is to minimize the *makespan*, i.e., the completion time of the last job that finishes.

Algorithms for this scheduling problem are used in multiprocessor scheduling. Moreover, the problem is important because it is the root of many problem variants where, for instance, preemption is allowed, precedence constraints exist among jobs, or machines run at different speeds. The problem was first investigated by Graham [10]. In fact, Graham also studied the *offline* version of the problem, when all jobs are known in advance. The problem of computing an optimal offline schedule for a given job sequence is NP-hard [9]. Graham gave a fast scheduling heuristic that achieves a good approximation ratio. He developed the well-known *List* algorithm that takes the given jobs one by one and always schedules them on the least loaded machine. Clearly, *List* is also an online algorithm.

Following [13], we call a deterministic online scheduling algorithm A *c-competitive* if, for all job sequences $\sigma = J_1, J_2, \ldots, J_n$,

$$A(\sigma) \leq c \cdot OPT(\sigma),$$

where $A(\sigma)$ is the makespan of the schedule produced by A and $OPT(\sigma)$ is the makespan of an optimal schedule for σ.

[*]Received by the editors July 23, 1997; accepted for publication (in revised form) July 9, 1998; published electronically October 5, 1999. A preliminary version of this paper was presented at the 29th Annual ACM Symposium on Theory of Computing, 1997. This work was supported in part by EU ESPRIT LTR project 20244 (ALCOM-IT).

http://www.siam.org/journals/sicomp/29-2/32487.html

[†]Max-Planck-Institut für Informatik, Im Stadtwald, D–66123 Saarbrücken, Germany (albers@mpi-sb.mpg.de).

Graham's *List* algorithm is $(2 - \frac{1}{m})$-competitive. Galambos and Woeginger [8] presented an algorithm that is $(2 - \frac{1}{m} - \epsilon_m)$-competitive, where $\epsilon_m > 0$, but ϵ_m tends to 0 as m goes to infinity. It was unknown for a long time whether there was an algorithm that achieves a competitive ratio of c, $c < 2$, for general m. Bartal, et al. [3] then gave an algorithm that is 1.986-competitive, for all $m \geq 70$. Karger, Phillips, and Torng [11] generalized the algorithm and proved a competitive ratio of 1.945, for all m. This has been the best upper bound known so far for general m. For the special case $m = 4$, Chen, van Vliet, and Woeginger [6] developed an algorithm that is 1.733-competitive. With respect to lower bounds, Faigle, Kern, and Turan [7] showed that no deterministic online algorithm can have a competitive ratio smaller than $(2 - \frac{1}{m})$ for $m = 2$ and $m = 3$. Thus, for these values of m, *List* is optimal. Faigle, Kern, and Turan [7] also proved that no deterministic online algorithm can be better than 1.707-competitive, for any $m \geq 4$. The best lower bound known so far for general m is due to Bartal, Karloff, and Rabani [4], who showed that no deterministic online algorithm can have a competitive ratio smaller than 1.837, for $m \geq 3454$. For more work on related online scheduling problems see, for instance, [1, 2, 5, 12, 14].

In this paper we present an improved deterministic online algorithm for the scheduling problem defined above. The algorithm is 1.923-competitive, for all $m \geq 2$. Our algorithm is based on a new scheduling strategy, i.e., it is not a generalization of the approach by Bartal et al. [3]. Moreover, the algorithm has a simple structure. At any time, the algorithm maintains a set S_1 of $\lfloor \frac{m}{2} \rfloor$ machines with a low load and a set S_2 of $\lceil \frac{m}{2} \rceil$ machines with a high load. Every job is either scheduled on the least loaded machine in S_1 or on the least loaded machine in S_2. The decision about which of the two machines to choose depends on the ratio of the load on machines in S_1 to the load on machines in S_2. A description of the algorithm is given in section 2. A detailed analysis follows in section 3. We also develop a better lower bound for online scheduling. In section 4 we show that if a deterministic online scheduling algorithm is c-competitive for all $m \geq 80$, then $c \geq 1.852$.

2. The new scheduling algorithm.

For the description of the algorithm we need some definitions. Let the *load* of a machine be the sum of the processing times of the jobs already assigned to it. At any time, the algorithm maintains a list of the machines sorted in nondecreasing order by the current load. Let M_i^t denote the machine with the ith smallest load, $1 \leq i \leq m$, after exactly t jobs have been scheduled. In particular, M_1^t is the machine with the smallest load and M_m^t is the machine with the largest load. We denote by l_i^t the load of machine M_i^t, $1 \leq i \leq m$. Note that the load l_m^t of the most loaded machine is always equal to the current makespan.

As with previous algorithms [3, 11], our new scheduling strategy tries to prevent schedules in which the load on all machines is about the same. If all machines have the same load, with all previous jobs being very small, an adversary can present an additional large job and force a competitive ratio of $(2 - \frac{1}{m})$. This is the worst-case scenario for *List*.

Our new algorithm, called $M2$, always tries to maintain k machines with a low load and $m - k$ machines with a high load, where $k = \lfloor \frac{m}{2} \rfloor$. The goal is to always have a schedule in which the total load L_l on the k lightly loaded machines is at most α times the total load L_h on the $m - k$ heavily loaded machines, for some α to be specified later. A schedule satisfying $L_l \leq \alpha L_h$ is always prepared to handle a large incoming job and can easily maintain a competitive ratio of c, where c is 1.923.

Algorithm $M2$ always schedules a new job J_t with processing time p_t on the least loaded machine as long as $L_l \leq \alpha L_h$ is satisfied after the assignment. Note that during this assignment, the load L_l on the lightly loaded machines does not necessarily increase by p_t because the least loaded machine might become one of the machines M_i^t, $k < i \leq m$. If an assignment of J_t to the least loaded machine results in $L_l > \alpha L_h$, then $M2$ considers scheduling J_t on the machine with the $(k+1)$st smallest load. However, if this assignment increases the makespan and the new makespan exceeds $c \cdot (L_l + L_h)/m$, then J_t is finally scheduled on the least loaded machine, ignoring the violation of $L_l \leq \alpha L_h$. Note that $L_l + L_h$ is the sum of the processing times of all jobs that have arrived so far, and thus $(L_l + L_h)/m$ is a lower bound on the optimum makespan.

Algorithm *M2*. Set $c = 1.923$, $k = \lfloor \frac{m}{2} \rfloor$, and $j = 0.29m$. Set $\alpha = \frac{(c-1)k - j/2}{(c-1)(m-k)}$. Every new job J_t is scheduled as follows. Let L_l be the sum of the loads on machines $M_1^t, \ldots, M_k^t$ if J_t is scheduled on the least loaded machine. Similarly, let L_h be the sum of the loads on machines $M_{k+1}^t, \ldots, M_m^t$ if J_t is scheduled on the least loaded machine. Let λ_m^t be the makespan, i.e., the load of the most loaded machine, if J_t is scheduled on the machine with the $(k+1)$st smallest load. Recall that l_m^{t-1} is the makespan before the assignment of J_t.

Schedule J_t on the least loaded machine if one of the following conditions holds.

(a) $L_l \leq \alpha L_h$.
(b) $\lambda_m^t > l_m^{t-1}$ and $\lambda_m^t > c \cdot \frac{L_l + L_h}{m}$.

Otherwise schedule J_t on the machine with the $(k+1)$st smallest load.

Theorem 1. *Algorithm M2 is 1.923-competitive for all $m \geq 2$.*

Before analyzing the algorithm in the next section, we discuss the choice of α. First observe that $0 < \alpha < 1$ for $m \geq 2$. The inequality $0 < \alpha$ holds because $c - 1 > 1/2$ and $k > j$; thus $(c-1)k - j/2 > 0$. Inequality $\alpha < 1$ holds because $(c-1)k \leq (c-1)(m-k)$ and $j/2 > 0$. In fact, for even m, $\alpha = \frac{(c-1) - j/m}{c-1} \approx 0.686$ and, for odd m, α tends to this value as m goes to infinity. Always setting $\alpha = 0.686$ in the algorithm $M2$ asymptotically results in the same competitive ratio of 1.923. Choosing $\alpha = \frac{(c-1)k - j/2}{(c-1)(m-k)}$ has two advantages. (1) We can prove a competitiveness of 1.923 for even small m. (2) In the analysis we can do symbolic calculations where a fixed $\alpha = 0.686$ would require numeric calculations.

3. Analysis of the algorithm.

We present a detailed proof of Theorem 1. The analysis presented by Graham [10] for the *List* algorithm, combined with the observation that algorithm $M2$ only schedules a job on the machine with the $(k+1)$st smallest load if the resulting makespan does not exceed 1.923 times the optimum makespan, shows that $M2$ is c-competitive, where $c = \max\{(2 - \frac{1}{m}), 1.923\}$, for all $m \geq 2$. This gives the desired bound for small m. For $m \geq 8$, the following analysis applies.

Consider an arbitrary job sequence $\sigma = J_1, J_2, \ldots, J_n$. Let p_t be the processing time of J_t, $1 \leq t \leq n$. We will show that $M2$ schedules every job J_t, $1 \leq t \leq n$, such that

$$M2(\sigma_t) \leq 1.923 \cdot OPT(\sigma_t),$$

where $M2(\sigma_t)$ is the makespan of the schedule produced by $M2$ on the subsequence $\sigma_t = J_1, J_2, \ldots, J_t$ and $OPT(\sigma_t)$ is the makespan of an optimal schedule for σ_t.

3.1. The basic analysis. Suppose that $M2$ has already scheduled the first $t-1$ jobs and that a competitive ratio of $c = 1.923$ was maintained at all times. Let

$$L = \sum_{s=1}^{t} p_s.$$

L is the sum of the loads on all machines after J_t is assigned.

If the makespan does not change during the assignment of J_t, then by the induction hypothesis there is nothing to show. Also, if the makespan changes but is bounded from above by $c\frac{L}{m}$, then we are done because $\frac{L}{m}$ is a lower bound on the optimum makespan for σ_t.

Thus we concentrate on the case where, during the assignment of J_t, the makespan increases and exceeds $c\frac{L}{m}$. Condition (b) in Algorithm $M2$ implies that J_t is scheduled on the least loaded machine. Let $l_1 = l_1^{t-1}$ be the load of the least loaded machine immediately before J_t is assigned.

First we consider the case where $l_1 \le (c-1)\frac{L}{m} = 0.923\frac{L}{m}$. We have

$$M2(\sigma_t) = l_1 + p_t \le (c-1)\tfrac{L}{m} + p_t.$$

If $p_t \le \frac{L}{m}$, then

$$M2(\sigma_t) \le (c-1)\tfrac{L}{m} + \tfrac{L}{m} \le c\tfrac{L}{m} \le c \cdot OPT(\sigma_t).$$

If $p_t = (1+\delta)\frac{L}{m}$, for some positive δ, then

$$\begin{aligned} M2(\sigma_t) &= l_1 + p_t \\ &\le (c-1)\tfrac{L}{m} + (1+\delta)\tfrac{L}{m} \\ &\le c \cdot (1+\delta)\tfrac{L}{m} = c \cdot p_t \\ &\le c \cdot \max_{1 \le s \le t} p_s \le c \cdot OPT(\sigma_t). \end{aligned}$$

Here we use the fact that $\max_{1\le s\le t} p_s$ is also a lower bound on the optimum makespan.

In the remainder of this proof we will study the situation that the load on the least loaded machine is greater than $(c-1)\frac{L}{m}$, i.e., $l_1 = (c-1+\epsilon)\frac{L}{m}$ for some positive ϵ. Since l_1 cannot be greater than $\frac{L}{m}$, we have $0 < \epsilon \le 2 - c = 0.077$. Note that all machines must have a load of at least $(c-1+\epsilon)\frac{L}{m}$. Since J_t is assigned to the least loaded machine and the makespan after the assignment is greater than $c\frac{L}{m}$, we have $p_t > c\frac{L}{m} - l_1 \ge (1-\epsilon)\frac{L}{m} \ge (\frac{1}{2}+\epsilon)\frac{L}{m}$. Our goal is to show that the sequence $\sigma_{t-1} = J_1, J_2, \ldots, J_{t-1}$ contains m jobs, each with a processing time of at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$. Then there are $m+1$ jobs with a processing time of at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$, two of which must be scheduled on the same machine in an optimal schedule. Thus

$$OPT(\sigma_t) \ge (1+2\epsilon)\tfrac{L}{m}.$$

If $p_t \le \frac{L}{m}$, then

$$\begin{aligned} M2(\sigma_t) = l_1 + p_t &\le (c-1+\epsilon)\tfrac{L}{m} + \tfrac{L}{m} \\ &\le c(1+\epsilon)\tfrac{L}{m} \le c \cdot OPT(\sigma_t). \end{aligned}$$

If $p_t = (1+\delta)\frac{L}{m}$ for some positive δ, then

$$\begin{aligned} M2(\sigma_t) &= (c-1+\epsilon)\tfrac{L}{m} + p_t \le (c+\epsilon+\delta)\tfrac{L}{m} \\ &\le c \cdot \max\{(1+2\epsilon),\ (1+\delta)\}\tfrac{L}{m} \\ &\le c \cdot OPT(\sigma_t). \end{aligned}$$

In each case, Theorem 1 is proved. It remains to show that the sequence $\sigma_{t-1} = J_1, J_2, \ldots, J_{t-1}$ does contain m jobs, each with a processing time of at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$.

3.2. Identifying large jobs. We have to analyze jobs in the sequence σ_{t-1}. Let *time* s, $1 \le s \le t$, denote the point of time immediately after J_s is scheduled. (Time 0 is the point of time before any jobs are scheduled.) For any time s, $1 \le s \le t$, let L_s be the total load on the m machines, i.e.,

$$L_s = \sum_{r=1}^{s} p_r.$$

Note that $L_t = L$.

DEFINITION 1. *At any time s, $1 \le s \le t$, the schedule constructed by $M2$ is called* steady *if the total load on the k lightly loaded machines $M_1^s, \ldots, M_k^s$ is at most α times the total load on the $m-k$ heavily loaded machines $M_{k+1}^s, \ldots, M_m^s$.*

In the following, when referring to machines $M_1^s, \ldots, M_m^s$, we will often drop s when the meaning is clear.

LEMMA 1. *At time $t-1$, i.e., immediately before J_t is scheduled, $M2$'s schedule is not steady.*

Proof. Immediately before J_t is scheduled, the total load on the machines $M_1, \ldots, M_k$ is at least

$$\begin{aligned} \overline{L}_l &= k(c-1+\epsilon)\tfrac{L}{m} \\ &> k(c-1)\tfrac{L}{m} \\ &= ((c-1)k - \tfrac{j}{2})\tfrac{L}{m} + \tfrac{j}{2}\tfrac{L}{m} \\ &= \alpha(c-1)(m-k)\tfrac{L}{m} + \tfrac{j}{2}\tfrac{L}{m}. \end{aligned}$$

If $M2$'s schedule was steady, then the total load on machines $M_{k+1}, \ldots, M_m$ would be at least $\frac{1}{\alpha}\overline{L}_l$. Thus the total load before the assignment of J_t would be at least

$$\begin{aligned} L_{t-1} &\ge (1+\tfrac{1}{\alpha})\overline{L}_l \\ &> k(c-1)\tfrac{L}{m} + (m-k)(c-1)\tfrac{L}{m} + \tfrac{1}{\alpha}\tfrac{j}{2}\tfrac{L}{m}. \end{aligned}$$

Here we used the facts $\overline{L}_l > k(c-1)\frac{L}{m}$ and $\overline{L}_l > \alpha(c-1)(m-k)\frac{L}{m} + \frac{j}{2}\frac{L}{m}$. Thus

$$\begin{aligned} L_{t-1} &> (c-1)L + \tfrac{j}{2\alpha}\tfrac{L}{m} \\ &= L + (c-2)L + \tfrac{j}{2\alpha}\tfrac{L}{m} \\ &> L \end{aligned}$$

because $\alpha = \frac{(c-1)k-j/2}{(c-1)(m-k)} \le \frac{(c-1)-j/m}{c-1} \le \frac{7}{10}$ and hence $(c-2)L + \frac{j}{2\alpha}\frac{L}{m} > -0.077L + 0.2L > 0$. We have a contradiction. ∎

3.2.1. Analyzing load.

Definition 2. *At any time s, $1 \leq s \leq t$, a machine is called* full *if its load is at least* $(c-1+\epsilon)\frac{L}{m}$.

Recall that at time $t-1$, all machines have a load of at least $(c-1+\epsilon)\frac{L}{m}$ and, thus, are full.

For $i = 1, \ldots, m$, let t_i be the most recent time when exactly i machines were full. Note that

$$t_1 < t_2 < \cdots < t_m = t-1.$$

Of particular interest to us will be the time $t_{m-\lfloor j \rfloor}$ when exactly $m - \lfloor j \rfloor$ machines were full. Let t', $t_{m-\lfloor j \rfloor} \leq t' < t-1$, be the most recent time when *M*2's schedule was steady. If *M*2's schedule was not steady during the time interval $[t_{m-\lfloor j \rfloor}, t-1]$, then let $t' = t_{m-\lfloor j \rfloor}$. Let f be the number of machines that are full at time t'.

Our goal is to show that at time t', the total load on the nonfull machines $M_1, \ldots, M_{m-f}$ in *M*2's schedule is at most $(c-1.5)(m-f)\frac{L}{m}$. We will show this using the following two lemmas. Let

$$X = \tfrac{(c-1)}{c}L,$$
$$Y = \tfrac{(c-1)^2}{c}L - \tfrac{j}{2}\tfrac{L}{m}.$$

Lemma 2. *If at time t' the total load on the nonfull machines $M_1, \ldots, M_{m-f}$ in M2's schedule were greater than $(c-1.5)(m-f)\frac{L}{m}$, then the total load at time t would satisfy $L > X + Y(1-\frac{c}{m})^{-(\lfloor j \rfloor+1)}$.*

The proof of Lemma 2 is presented in the appendix.

Lemma 3. $X + Y(1-\frac{c}{m})^{-(\lfloor j \rfloor+1)} \geq L$.

Proof. We have

$$\left(1-\frac{c}{m}\right)^{-(\lfloor j \rfloor+1)} \geq \left(1-\frac{c}{m}\right)^{-j} \geq e^{cj/m}.$$

The first inequality follows because $\lfloor j \rfloor + 1 \geq j$. Thus,

$$\begin{aligned} X + Y(1-\tfrac{c}{m})^{-(\lfloor j \rfloor+1)} &\geq \tfrac{(c-1)}{c}L + (\tfrac{(c-1)^2}{c}L - \tfrac{j}{2m}L) \cdot e^{cj/m} \\ &= (1-\tfrac{1}{1.923})L + (\tfrac{0.923^2 - 1.923 \cdot 0.145}{1.923}L) \cdot e^{0.29 \cdot 1.923}. \end{aligned}$$

Evaluating the last expression gives that it is at least $1 \cdot L$. □

We summarize the results of Lemmas 2 and 3.

Lemma 4. *At time t', the total load on the nonfull machines $M_1, \ldots, M_{m-f}$ is at most $(c-1.5)(m-f)\frac{L}{m}$.*

3.2.2. Tracing the assignment of large jobs. We now identify jobs with a processing time of at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$.

Lemma 5. *During the time interval $(t_{m-k}, t']$, $f-m+k$ jobs, each of size at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$, are scheduled.*

Proof. At time t_{m-k}, $m-k$ machines are full. At time t', f machines are full, where $f \geq m - \lfloor j \rfloor$. Consider the $f-m+k$ steps in $(t_{m-k}, t']$ at which the number of full machines increases. Since at least $m-k$ machines are full, the number of full machines can increase only if a job is scheduled on the least loaded machine. By Lemma 4, at time t', the total load on the $m-f$ least loaded machines is at most

$(c-1.5)(m-f)\frac{L}{m}$. This implies that at time t', the load on the least loaded machine is at most $(c-1.5)\frac{L}{m}$. Thus, at any of the $f-m+k$ steps in $(t_{m-k}, t']$ at which the number of full machines increases, the load on the least loaded machine is at most $(c-1.5)\frac{L}{m}$. Hence jobs of size at least $(c-1+\epsilon)\frac{L}{m} - (c-1.5)\frac{L}{m} = (\frac{1}{2}+\epsilon)\frac{L}{m}$ are introduced. ◻

LEMMA 6. *At time t_{m-k}, each of the machines $M_1, \ldots, M_{f-m+k+1}$ has a load of at most $(c-1.5)\frac{L}{m}$. The total load on the machines $M_{f-m+k+1}, \ldots, M_k$ is at most $(c-1.5)(m-f)\frac{L}{m}$.*

Proof. The machine with the $(f-m+k+1)$st smallest load at time t_{m-k} becomes the least loaded machine no later than time t', when f machines are full. Thus, if at time t_{m-k}, machine $M_{f-m+k+1}$ (or any machine M_i with $i \le f-m+k$) had a load greater than $(c-1.5)\frac{L}{m}$, then the load of the least loaded machine at time t' would be greater than $(c-1.5)\frac{L}{m}$. Lemma 4 implies that this is impossible. Similarly, if at time t_{m-k}, machines $M_{f-m+k+1}, \ldots, M_k$ had a total load of at least $(c-1.5)(m-f)\frac{L}{m}$, then the total load on $M_1, \ldots, M_{m-f}$ at time t' would be at least $(c-1.5)(m-f)\frac{L}{m}$. Again, Lemma 4 gives the desired statement. ◻

LEMMA 7. *During the time interval $(t', t-1]$, $m-f$ jobs of size at least $(\frac{1}{2}+\epsilon)\frac{L}{m}$ are scheduled.*

Proof. By the definition of t', $M2$'s schedule is not steady during $[t'+1, t-1]$. Let $s \in [t'+1, t-1]$ be any of the $m-f$ time steps in $[t'+1, t-1]$ at which the number of full machines has just increased. If the load on the least loaded machine is at most $(c-1.5)\frac{L}{m}$ when J_s is scheduled, then $p_s \ge (c-1+\epsilon)\frac{L}{m} - (c-1.5)\frac{L}{m} = (\frac{1}{2}+\epsilon)\frac{L}{m}$.

Suppose that immediately before the assignment of J_s, the least loaded machine has a load greater than $(c-1.5)\frac{L}{m}$. Let l_{k+1}^{s-1} be the load of machine M_{k+1} and suppose $l_{k+1}^{s-1} = (c-1+\epsilon+\delta)\frac{L}{m}$ for some nonnegative δ. By the definition of t', at least $m - \lfloor j \rfloor$ machines are full at any time in $[t', t-1]$. Thus,

$$\begin{aligned} L_{s-1} &\ge (m-\lfloor j\rfloor)(c-1+\epsilon)\tfrac{L}{m} + \lfloor j\rfloor(c-1.5)\tfrac{L}{m} + (m-k)\delta\tfrac{L}{m} \\ &\ge (c-1)L - \tfrac{1}{2}\lfloor j\rfloor\tfrac{L}{m} + (m-\lfloor j\rfloor)\epsilon\tfrac{L}{m} + \tfrac{\delta}{2}L \\ &\ge (c-1)L - \tfrac{j}{2}\tfrac{L}{m} + (m-j)\epsilon\tfrac{L}{m} + \tfrac{\delta}{2}L. \end{aligned}$$

The second inequality follows because $m-k \ge \frac{1}{2}m$. Since $M2$'s schedule is not steady, $M2$ would prefer to schedule J_s on machine M_{k+1} but cannot because

$$l_{k+1}^{s-1} + p_s > c(L_{s-1} + p_s)/m.$$

Hence,

$$\begin{aligned} p_s &\ge (\tfrac{c}{m}L_{s-1} - l_{k+1}^{s-1})/(1-\tfrac{c}{m}) \\ &\ge \tfrac{c}{m}L_{s-1} - l_{k+1}^{s-1} \\ &\ge c(c-1-\tfrac{j}{2m})\tfrac{L}{m} - (c-1)\tfrac{L}{m} + c(1-\tfrac{j}{m})\epsilon\tfrac{L}{m} - \epsilon\tfrac{L}{m} + \tfrac{c\delta}{2}\tfrac{L}{m} - \delta\tfrac{L}{m}. \end{aligned}$$

The load $l_{k+1}^{s-1} = (c-1+\epsilon+\delta)\frac{L}{m}$ cannot be greater than $(3-c-\epsilon)\frac{L}{m}$ since otherwise we would have, at time $t-1$, $m-k$ machines each with a load greater than $(3-c-\epsilon)\frac{L}{m}$ and k machines each with a load of at least $(c-1+\epsilon)\frac{L}{m}$, resulting in a total load greater than L. Thus, $\delta \le 4-2c-2\epsilon$ and

$$p_s \ge ((c-1)^2 - \tfrac{cj}{2m})\tfrac{L}{m} + (c-1-\tfrac{cj}{m})\epsilon\tfrac{L}{m} + 2(\tfrac{c}{2}-1)(2-c-\epsilon)\tfrac{L}{m}$$

$$= ((c-1)^2 - (c-2)^2 - \tfrac{cj}{2m})\tfrac{L}{m} + (1 - \tfrac{cj}{m})\epsilon\tfrac{L}{m}$$
$$\geq 0.567\tfrac{L}{m} + 0.442\epsilon\tfrac{L}{m}$$
$$\geq (\tfrac{1}{2} + \epsilon)\tfrac{L}{m}$$

for all $\epsilon \leq 0.12$. Recall that our ϵ is at most $2 - c = 0.077$. □

We now consider the time $t_{m-k-\lfloor j\rfloor}$ when exactly $m - k - \lfloor j\rfloor$ machines are full. Let t'' be the earliest point of time in the interval $[t_{m-k-\lfloor j\rfloor}, t_{m-k}]$ at which the machine with the $(k+1)$st smallest load has a load greater than $(c-1.5)\frac{L}{m}$.

LEMMA 8. *During the time interval $(t'', t_{m-k}]$, every job is scheduled on the least loaded machine.*

Proof. We first show that at any time $s \in [t'', t_{m-k}]$, *M*2's schedule is steady. Lemma 6 implies that at time s the total load on the lightly loaded machines $M_1, \ldots, M_k$ is at most $\overline{L}_l = k(c-1.5)\frac{L}{m}$. By the definition of t'', the total load on the heavily loaded machines $M_{k+1}, \ldots, M_m$ at time s is at least

$$\begin{aligned}\overline{L}_h &= (m-k-\lfloor j\rfloor)(c-1)\tfrac{L}{m} + \lfloor j\rfloor(c-1.5)\tfrac{L}{m}\\ &= (m-k)(c-1)\tfrac{L}{m} - \tfrac{\lfloor j\rfloor}{2}\tfrac{L}{m}.\end{aligned}$$

We show that at time s, the total load on the lightly loaded machines is at most α times the load on the heavily loaded machines. This holds if $\overline{L}_l \leq \alpha\overline{L}_h$, i.e., if

$$k(c-1.5)\tfrac{L}{m} \leq \tfrac{k(c-1)-j/2}{(c-1)(m-k)}((m-k)(c-1)\tfrac{L}{m} - \tfrac{\lfloor j\rfloor}{2}\tfrac{L}{m}),$$

which is equivalent to

$$(c-1)(k\lfloor j\rfloor + (m-k)j) - j\tfrac{\lfloor j\rfloor}{2} \leq (c-1)k(m-k).$$

This in turn holds if

$$jm - \tfrac{j^2}{2(c-1)} \leq k(m-k).$$

The left-hand side is at most $0.245m^2$, and the right-hand side is $\frac{1}{4}m^2$ for even m and at least $0.246m^2$ for odd $m \geq 9$. Thus, at time s, *M*2's schedule is steady.

Now consider job J_{s+1} scheduled at time $s+1$. Let l_1^s be the load on the least loaded machine at time s. We have $l_1^s \leq (c-1.5)\frac{L}{m}$. Let p be a processing time such that $l_1^s + p = (c-1.5)\frac{L}{m}$. The property stated in Lemma 6 must also hold at time s because the load on the lightly loaded machines $M_1, \ldots, M_k$ can only be smaller. Thus, if J_{s+1} has a processing time of at most p, scheduling J_{s+1} on the least loaded machine results in a total load of at most $k(c-1.5)\frac{L}{m}$ on machines $M_1, \ldots, M_k$. Since the total load on machines $M_{k+1}, \ldots, M_m$ is at least $(m-k)(c-1)\frac{L}{m} - \frac{\lfloor j\rfloor}{2}\frac{L}{m}$, the calculations of the preceding paragraph show that *M*2's schedule must be steady after the assignment.

Suppose that J_{s+1} has a processing time $p_{s+1} > p$ and that scheduling J_{s+1} on the least loaded machine results in a load of $k(c-1.5)\frac{L}{m} + \delta\frac{L}{m}$ on machines $M_1, \ldots, M_k$, for some $\delta > 0$. This implies that at time $s+1$, the load on any of the machines $M_{k+1}, \ldots, M_{k+\lfloor j\rfloor}$ must be at least $(c-1.5+\delta)\frac{L}{m}$. With the above definitions of $\overline{L}_l$ and $\overline{L}_h$, we conclude that after the assignment of J_{s+1} to the least loaded machine, the total load on $M_1, \ldots, M_k$ is at most $\overline{L}_l + \delta\frac{L}{m}$ and the total load on $M_{k+1}, \ldots, M_m$ is at least $\overline{L}_h + \lfloor j\rfloor\delta\frac{L}{m}$. Since, for $m \geq 8$, we have $\lfloor j\rfloor \geq 2$ and $\alpha \geq \frac{1}{2}$, *M*2's schedule must be steady. ∎

LEMMA 9. *At time $t'' - 1$, the load on machine M_{k+1} is at most $(c - 1.5)\frac{L}{m}$.*

Proof. If $t'' > t_{m-k-\lfloor j \rfloor}$, then the lemma follows from the definition of t''. We show that t'' cannot be equal to $t_{m-k-\lfloor j \rfloor}$. Recall that $f \geq m - \lfloor j \rfloor$. Thus, Lemma 6 implies that at time t_{m-k}, machine $M_{k-\lfloor j \rfloor+1}$ has a load of at most $(c-1.5)\frac{L}{m}$. If $t'' = t_{m-k-\lfloor j \rfloor}$, then there are $\lfloor j \rfloor$ steps in $(t'', t_{m-k}]$ at which the number of full machines increases. By Lemma 8, at all these steps, the jobs are assigned to the least loaded machine. Thus at time t'', the load of machine M_{k+1} cannot be greater than the load of machine $M_{k-\lfloor j \rfloor+1}$ at time t_{m-k}. This means that M_{k+1} has a load of at most $(c-1.5)\frac{L}{m}$ at time t'', contradicting the choice of t''. ∎

LEMMA 10. *During time interval $(0, t_{m-k}]$, $m - k$ jobs of size at least $(\frac{1}{2} + \epsilon)\frac{L}{m}$ are scheduled.*

Proof. Let i be the number of machines that are full at time t''. Consider the i steps in $(0, t'']$ at which the number of full machines increases. At any of these steps, before the assignment of the job, the load on M_1 and M_{k+1} is at most $(c-1.5)\frac{L}{m}$ each; see Lemma 9. Thus jobs of size at least $(c - 1 + \epsilon)\frac{L}{m} - (c-1.5)\frac{L}{m} \geq (\frac{1}{2} + \epsilon)\frac{L}{m}$ must be scheduled. At the $m - k - i$ steps in $(t'', t_{m-k}]$ at which the number of full machines increases, jobs are scheduled on the least loaded machine (Lemma 8). The least loaded machine has a load of at most $(c-1.5)\frac{L}{m}$ and we conclude again that jobs of size at least $(\frac{1}{2} + \epsilon)\frac{L}{m}$ must be scheduled. ∎

Lemma 5 as well as Lemmas 7 and 10 imply the following statement.

LEMMA 11. *During time interval $(0, t-1]$, m jobs of size at least $(\frac{1}{2} + \epsilon)\frac{L}{m}$ are scheduled.*

By the discussion immediately preceding section 3.2, the proof of Theorem 1 is complete.

4. The lower bound. We develop an improved lower bound for deterministic scheduling algorithms.

THEOREM 2. *Let A be a deterministic online scheduling algorithm. If A is c-competitive for all $m \geq 80$, then $c \geq 1.852$.*

Proof. We will construct a job sequence σ such that $A(\sigma) \geq 1.852 \cdot OPT(\sigma)$. The job sequence consists of several rounds. We assume that m is a multiple of 40.

Round 1: m jobs with a processing time of $w = 0.01$.
Round 2: m jobs with a processing time of $x = 0.06$.
Round 3:
- Subround 3.1: $\frac{19}{20}m$ jobs with a processing time of $y_1 = 0.282$.
- Subround 3.2: $\frac{1}{20}m$ jobs with a processing time of $y_2 = 0.4$.

Round 4:
- Subround 4.1: $\frac{1}{2}m$ jobs with a processing time of $z_1 = 0.5$.
- Subround 4.2: $\frac{1}{4}m$ jobs with a processing time of $z_2 = 1 - y_2 = 0.6$.
- Subround 4.3: $\frac{3}{40}m$ jobs with a processing time of $z_3 = 1 - y_1 = 0.718$.
- Subround 4.4: $\frac{3}{40}m$ jobs with a processing time of $z_4 = 0.84$.
- Subround 4.5: $\frac{1}{10}m + 1$ jobs with a processing time of $z_5 = 1$.

Note that in the fourth round, $m + 1$ jobs have to be scheduled.

In the following, when analyzing the various subrounds, we will often compare the makespan produced by an online algorithm A in a subround to the optimum makespan at the end of the subround. It is clear that the optimum makespan during the subround can only be smaller.

Analysis of Round 1: Clearly, in order to maintain 1.852-competitiveness, online algorithm A must schedule the m jobs in Round 1 on different machines.

Analysis of Round 2: Algorithm A must schedule the m jobs in Round 2 on different machines. Otherwise, A's makespan would be at least $w + 2x = 0.13$. Since the optimum makespan during the round is always at most $w + x = 0.07$ and $\frac{0.13}{0.07} > 1.857$, A would not be 1.852-competitive. At the end of the second round, A has a load of $l_2 = w + x = 0.07$ on each of its machines.

Analysis of Round 3: At the end of Subround 3.1, the optimum makespan is at most $x + y_1 = 0.342$. On each of $\frac{19}{20}m$ machines, OPT schedules an x-job and a y_1-job. The remaining $\frac{1}{20}m$ machines have an x-job and 20 jobs of size w. If A does not schedule the jobs in Subround 3.1 on different machines, then its makespan is at least $w + x + 2y_1 = 0.634 > 1.853(x + y_1)$. The optimum makespan after Subround 3.2 is $y_1 + 2x = 0.402$. In an optimal schedule, $\frac{1}{20}m$ machines have a y_2-job, $\frac{1}{2}m$ machines have a y_1-job and two x-jobs. The remaining machines have a y_1-job and at most three w-jobs. Online algorithm A must schedule the jobs of Subround 3.2 on different machines and these machines may not contain any y_1-jobs since otherwise A's makespan is at least $w+x+y_1+y_2 = 0.752 > 1.87(y_1+2x)$. At the end of Round 3, the least loaded machine in A's schedule has a load of $l_3 = w + x + y_1 = 0.352$.

Analysis of Round 4: *Subround* 4.1: After Subround 4.1, the optimum makespan is $y_1 + y_2 = 0.682$. In an optimal schedule, $\frac{1}{20}m$ machines contain a y_1 and a y_2. $\frac{1}{2}m$ machines contain a z_1, two w and two x. $\frac{9}{20}m$ machines contain two y_1. Algorithm A must schedule all z_1-jobs on different machines. Otherwise its makespan would be at least $l_3 + 2z_1 = 1.352 > 1.98(y_1 + y_2)$.

Subround 4.2: At the end of the subround, the optimum makespan is $y_1 + z_1 = 0.782$. In OPT's schedule, $\frac{1}{2}m$ machines have a y_1 and a z_1. $\frac{1}{20}m$ machines have a y_1 and a y_2. $\frac{1}{5}m$ machines have two y_1, three w, and three x. $\frac{1}{4}m$ machines have a z_2 and some of them have two additional w and x. Algorithm A must schedule each z_2-job on a machine not containing any z_1 or z_2 jobs. Otherwise its makespan would be at least $l_3 + z_1 + z_2 = 1.452$, which is greater than $1.856(y_1 + z_1)$.

Subround 4.3: The optimum makespan after the subround is $3y_1 = 0.846$. In an optimal schedule $\frac{1}{2}m$ machines have a y_1, a z_1, and an x. $\frac{1}{4}m$ machines have a z_2, two x, and four w. $\frac{3}{40}m$ machines have a z_3. $\frac{3}{20}m$ machines have three y_1. $\frac{1}{40}m$ machines have two y_2. As before, A may not schedule any z_3-jobs on a machine containing a z_1, a z_2, or a z_3 because this would result in a makespan of at least $l_3 + z_1 + z_3 = 1.57 > 1.855(3y_1)$.

Subround 4.4: The optimum makespan is $y_2 + z_1 = 0.9$. In OPT's schedule, all the z-jobs are scheduled on different machines. $\frac{1}{20}m$ machines having a z_1 also contain a y_2. $(\frac{1}{2} - \frac{1}{20})m$ machines containing a z_1 also have a y_1, an x, and up to three w. The $\frac{1}{4}m$ machines having a z_2 also have a y_1. Machines having a z_3 also have three x. Machines having a z_4 also have an x. At this point, OPT is left with $\frac{1}{10}m$ machines on which it has to schedule $\frac{1}{4}m$ jobs with a processing time of x and $\frac{1}{4}m$ jobs with a processing time of y_1. This can be done by scheduling (a) $\frac{1}{40}m$ machines with 10 x and one y_1 each and (b) $\frac{3}{40}m$ machines with three y_1. As usual, A may not schedule a z_4-job on a machine already having any z-jobs; otherwise its makespan is at least $l_3 + z_1 + z_4 = 1.692 = 1.88(y_2 + z_1)$.

Subround 4.5: The online algorithm A must schedule one of the z_5-jobs on a machine already containing another z-job, because a total of $m + 1$ jobs have to be scheduled in Round 4. This gives a makespan of at least $w + x + y_1 + z_1 + z_5 = 1.852$.

We will show that OPT can schedule all the jobs with a makespan of 1 if $m \geq 80$. An optimal schedule is as follows. $\frac{1}{10}m$ machines have a z_5. $\frac{1}{4}m$ machines have two z_1. $\frac{3}{40}m$ machines have a z_4, two w, and two x. $\frac{3}{40}m$ machines have a z_3 and a y_1. $\frac{1}{5}m$ machines have a z_2, one y_1, two w, and one x. $\frac{1}{20}m$ machines have a z_2 and a y_2. $\frac{9}{40}m$ machines have three y_1, two w, and two x. OPT has $\frac{1}{40}m$ machines left on which it has to schedule one z_5 job and $\frac{1}{5}m$ jobs of size x. This can be done if at least two machines are left, i.e., if $m \geq 80$. OPT can use one machine for the z_5-job and the remaining machines for the x-jobs. ▯

5. Open problems. An interesting problem is to formulate and analyze a generalization of the algorithm $M2$ that, at any time, is allowed to schedule a new job on any of the m machines. In such an algorithm, the ratio of the load on the ith smallest machine to the load on the $(i+1)$st smallest machine has to be bounded by some α_i, $1 \leq i \leq m-1$. The problem is to specify α_i's and a proper scheduling rule that is able to maintain these values. A first step in this direction is to maintain three sets S_1, S_2, and S_3 of $m/3$ machines with low, medium, and high loads, respectively.

More generally, with respect to the scheduling problem studied here, a fundamental open problem is to develop randomized online algorithms that achieve a competitive ratio smaller than the deterministic lower bound for all m.

Appendix. We prove Lemma 2. For convenience, we state the lemma again.

LEMMA A.1. *If at time t', the total load on the nonfull machines $M_1, \ldots, M_{m-f}$ in $M2$'s schedule were greater than $(c-1.5)(m-f)\frac{L}{m}$, then the total load at time t would satisfy $L > X + Y(1-\frac{c}{m})^{-(\lfloor j \rfloor+1)}$.*

Proof. In order to prove the lemma, we have to keep track of the load on the m machines during the entire time interval $[t', t]$. For $i = f, \ldots, m+1$, let

$$Z_i = X + Y(1-\tfrac{c}{m})^{-(i-m+\lfloor j \rfloor)}.$$

We will show by induction on i that for $i = f, \ldots, m$,

$$L_{t_i} - \Phi_{t_i} > Z_i, \tag{1}$$

where Φ is a nonnegative potential that we will define in a moment. Using the inequality $L_{t_m} - \Phi_{t_m} > Z_m$, we will then prove $L > Z_{m+1}$.

We first explain the purpose of the potential. We want to show that during the time interval $(t', t-1]$, every time another machine becomes full, a job J with a large processing time p must be scheduled. Since, by the definition of t', $M2$'s schedule is not steady in $(t', t-1]$, $M2$ would prefer to assign J to machine M_{k+1}. However, $M2$ schedules J on the least loaded machine, causing another machine to become full. This implies that an assignment of J to machine M_{k+1} results in an increased makespan that exceeds c times the average load on the machines, i.e., J's processing time p must be large. In some cases, when M_{k+1} has a high load, we will not be able to argue that J's processing time is greater than a certain value. In these cases we will pay some "missing processing time" out of the potential. This way we can ensure that J's *amortized processing time* is greater than the desired value. J's amortized processing time is the actual processing time plus the change in potential.

Formally, the potential Φ is defined as follows. At time t', we *color* some of the load in $M2$'s schedule. More precisely, on each of the machines $M_1, \ldots, M_{k+m-f}$ we color the load that is above level $(c-1)\frac{L}{m}$. We can imagine that we draw a horizontal line at level $(c-1)\frac{L}{m}$ across $M2$'s schedule and color the load on machines

$M_1, \dots, M_{k+m-f}$ that is above this line. Note that this way a job might be partially colored. During time interval $(t', t]$, the colored load is updated as follows.

1. Whenever $M2$ schedules a job that causes one more machine to become full, we choose the least loaded machine with colored load among $M_{k+1}, \dots, M_m$ and remove the color from its load.
2. Whenever a job is assigned to a machine with colored load, we color that job.
3. After the final job J_t is scheduled, the color is removed from all machines.

At any time, let $I = \{i | M_i \text{ has colored load}\}$ and let c_i, $i \in I$ be the amount of the load that is colored on M_i. Define

$$\Phi = \sum_{i \in I} c_i.$$

During the interval $(t', t-1]$, the following invariants hold.

I1. Whenever $M2$ schedules a job that causes one more machine to become full, there is a machine in $\{M_{k+1}, \dots, M_m\}$ with colored load.
I2. If a machine has colored load, then all its load above level $(c-1)\frac{L}{m}$ is colored.
I3. At any time, if machine M_{k+1} has load $(c-1+\delta)\frac{L}{m}$ for some positive δ, then $c_i \geq \delta \frac{L}{m}$ for all $i \in I$ with $i \geq k+1$.
I4. At any time, there exists a machine among $M_1, \dots, M_k$ with colored load at least $\epsilon \frac{L}{m}$.

Invariant I1 holds because at time t' there are $m-f$ machines in $\{M_{k+1}, \dots, M_m\}$ with colored load, exactly $m-f$ more jobs are scheduled in $(t', t-1]$ that cause a machine to become full, and every time this happens, by update rule 1, the number of machines in $\{M_{k+1}, \dots, M_m\}$ with colored load is reduced by exactly 1. Invariant I2 follows from update rule 2. Invariant I3 is immediate from I2. Invariant I4 holds because initially, at time t', we color loads on the k lightly loaded machines that are full and these machines remain in the set of lightly loaded machines during $(t', t-1]$.

Base of the induction. In order to prove inequality (1) for $i = f$, we have to evaluate $L_{t'}$, the total load on the m machines at time t'. We will show that

$$L_{t'} - \Phi_{t'} > Z_f. \tag{2}$$

This implies that inequality (1) holds for $i = f$ because, between time t' and time t_f, the number of full machines remains the same and, whenever the load on the m machines increases by p, the potential increases by at most p (see update rule 2).

If $M2$'s schedule is not steady at time t', i.e., $t' = t_{m-\lfloor j \rfloor}$ and $f = m - \lfloor j \rfloor$, then

$$Z_f = X + Y = (c-1)L - \tfrac{j}{2}\tfrac{L}{m} \leq (c-1)L - \tfrac{\lfloor j \rfloor}{2}\tfrac{L}{m}. \tag{3}$$

By assumption, at time t', the load on the nonfull machines is greater than $(c-1.5)(m-f)\frac{L}{m} = (c-1.5)\lfloor j \rfloor \frac{L}{m} = (c-1)\lfloor j \rfloor \frac{L}{m} - \frac{\lfloor j \rfloor}{2}\frac{L}{m}$. The load on the full machines is at least $(c-1)f\frac{L}{m} + \Phi_{t'} = (c-1)(m - \lfloor j \rfloor)\frac{L}{m} + \Phi_{t'}$. We obtain

$$L_{t'} - \Phi_{t'} > (c-1)L - \tfrac{\lfloor j \rfloor}{2}\tfrac{L}{m}. \tag{4}$$

Inequalities (3) and (4) give the desired bound.

We study the case where $M2$'s schedule is steady at time t'. The total load on the nonfull machines is greater than $(c-1.5)(m-f)\frac{L}{m}$. The load on each machine

$M_{m-f+1},\ldots,M_k$ is at least $(c-1+\epsilon)\frac{L}{m}$. Thus the total load $\overline{L}_l$ on the k lightly loaded machines $M_1,\ldots,M_k$ is

$$\begin{aligned}\overline{L}_l &> (c-1+\epsilon)(k-m+f)\tfrac{L}{m} + (c-1.5)(m-f)\tfrac{L}{m}\\ &= (c-1)k\tfrac{L}{m} - \tfrac{1}{2}(m-f)\tfrac{L}{m} + (k-m+f)\epsilon\tfrac{L}{m}\\ &\geq (c-1)k\tfrac{L}{m} - \tfrac{j}{2}\tfrac{L}{m} + \tfrac{1}{2}(f-m+\lfloor j\rfloor)\tfrac{L}{m} + (k-m+f)\epsilon\tfrac{L}{m}.\end{aligned}$$

Note that the load $(k-m+f)\epsilon\frac{L}{m}$ will go into the potential. Since the schedule is steady, the total load $\overline{L}_h$ on the heavily loaded machines $M_{k+1},\ldots,M_m$ is at least $\frac{1}{\alpha}$ times the above expression. Neglecting the term $\frac{1}{\alpha}(k-m+f)\epsilon\frac{L}{m}$, we obtain

$$\overline{L}_h > (c-1)(m-k)\tfrac{L}{m} + \tfrac{1}{2\alpha}(f-m+\lfloor j\rfloor)\tfrac{L}{m}.$$

The load $(c-1)(m-k)\frac{L}{m}$ can fill the machines $M_{k+1},\ldots,M_m$ up to a level of $(c-1)\frac{L}{m}$. Of the additional load $\frac{1}{2\alpha}(f-m+\lfloor j\rfloor)\frac{L}{m}$, at least a fraction of $\frac{f-k}{m-k}$ is located on machines $M_{k+m-f+1},\ldots,M_m$ and does not go into the potential. Thus,

$$\begin{aligned}L_{t'} - \Phi_{t'} &= \overline{L}_l + \overline{L}_h - \Phi_{t'}\\ &> (c-1)L - \tfrac{j}{2}\tfrac{L}{m} + \tfrac{1}{2}(f-m+\lfloor j\rfloor)(1+\tfrac{1}{\alpha}\tfrac{f-k}{m-k})\tfrac{L}{m}\\ &= X + Y + \tfrac{1}{2}(f-m+\lfloor j\rfloor)(1+\tfrac{1}{\alpha}\tfrac{f-k}{m-k})\tfrac{L}{m}.\end{aligned}$$

We have to show that

$$\tfrac{1}{2}(f-m+\lfloor j\rfloor)(1+\tfrac{1}{\alpha}\tfrac{f-k}{m-k})\tfrac{L}{m} \geq Y((1-\tfrac{c}{m})^{-(f-m+\lfloor j\rfloor)}-1) \tag{5}$$

holds for every $f \in \{m-\lfloor j\rfloor,\ldots,m\}$. This proves inequality (2).

We define functions

$$g(x) = \tfrac{1}{2}(x-m+\lfloor j\rfloor)(1+\tfrac{1}{\alpha}\tfrac{x-k}{m-k})\tfrac{L}{m},$$

$$h(x) = Y((1-\tfrac{c}{m})^{-(x-m+\lfloor j\rfloor)}-1).$$

The function $g(x)$ is a polynomial of degree 2, and $h(x)$ is an exponentially increasing function. Obviously, $g(m-\lfloor j\rfloor) = h(m-\lfloor j\rfloor) = 0$. We will show that

$$g'(m-\lfloor j\rfloor) > h'(m-\lfloor j\rfloor) \tag{6}$$

and

$$g(y) > h(y) \quad \text{for some } y > m. \tag{7}$$

This implies that $g(x) > h(x)$ must hold for all $x \in (m-\lfloor j\rfloor, m]$. (If $g(z) \leq h(z)$ were true for some $z \in (m-\lfloor j\rfloor, m]$, then $g(x) < h(x)$ for all $x > z$.)

Recall that, as mentioned in the proof of Lemma 1, $\alpha \leq \frac{7}{10}$. Also, evaluating Y with its actual parameters gives $Y \leq 0.299L$. We have

$$\begin{aligned}g'(m-\lfloor j\rfloor) &= \tfrac{1}{2}(1+\tfrac{1}{\alpha}\tfrac{m-\lfloor j\rfloor-k}{m-k})\tfrac{L}{m}\\ &\geq \tfrac{1}{2}(1+\tfrac{10}{7}\cdot\tfrac{m-k-\lfloor j\rfloor}{m-k})\tfrac{L}{m}\\ &\geq (\tfrac{17}{14}-\tfrac{10}{7}\tfrac{j}{m})\tfrac{L}{m}\\ &= 0.8\tfrac{L}{m}.\end{aligned}$$

The last inequality holds because $\lfloor j \rfloor \le j$ and $m - k \ge m/2$. Also, $h'(m - \lfloor j \rfloor) = Y \ln((1 - \frac{c}{m})^{-1}) \le 0.299\frac{L}{m} \ln((1 - \frac{c}{m})^{-m}) < 0.79\frac{L}{m}$. The last inequality holds because $\ln((1-\frac{c}{m})^{-m})$ is decreasing in m and evaluates to less than 2.63 for $m \ge 4$. This shows (6). For the proof of (7), let $y = m + j - \lfloor j \rfloor$. Then $g(y) \ge \frac{1}{2}0.29m(1 + \frac{10}{7})\frac{L}{m} \ge 0.35L$. Also, $h(y) = Y((1 - \frac{c}{m})^{-0.29m} - 1) \le 0.299L((1 - \frac{c}{m})^{-0.29m} - 1)$. The last expression is decreasing in m and less than $0.341L$ for all $m \ge 4$. The proof of inequality (5) is complete.

Induction step. We show that if inequality (1) holds for $i-1$, then it also holds for i. Let s_i be the earliest point of time when exactly i machines are full. We have $t_{i-1} < s_i \le t_i$. For all $s \in [t_{i-1}, s_i - 1]$,

$$L_s - \Phi_s > Z_{i-1}. \tag{8}$$

This is because of the induction hypothesis and the fact that if the load on the m machines increases by p between time t_{i-1} and time $s_i - 1$, then the potential increases by at most p.

Let L_{s_i-1} be the total load on the m machines at time $s_i - 1$ and let $l = l_{k+1}^{s_i-1}$ be the load on machine M_{k+1} at time $s_i - 1$. Suppose $l = (c - 1 + \delta)\frac{L}{m}$ for some $\delta > 0$. The job J_{s_i} that causes the ith machine to become full is scheduled on the least loaded machine. Since *M*2's schedule is not steady at time s_i, *M*2 would prefer to schedule J_{s_i} on machine M_{k+1}. Since this is not possible, condition (b) in Algorithm *M*2 implies that the processing time p_{s_i} of J_{s_i} must satisfy

$$l + p_{s_i} > \frac{c(L_{s_i-1} + p_{s_i})}{m},$$

which is equivalent to

$$p_{s_i} > (\tfrac{c}{m} L_{s_i-1} - l)/(1 - \tfrac{c}{m}).$$

Consider the change in potential during the assignment of J_{s_i}. Update rule 1 and invariants I1–I3 imply that the potential drops by at least $\delta \frac{L}{m}$.

$$\begin{aligned}
p_{s_i} - \Delta\Phi &> (\tfrac{c}{m} L_{s_i-1} - l)/(1 - \tfrac{c}{m}) + \delta \tfrac{L}{m} \\
&\ge (\tfrac{c}{m}(Z_{i-1} + \Phi_{s_i-1}) - l)/(1 - \tfrac{c}{m}) + \delta \tfrac{L}{m} \\
&\ge (\tfrac{c}{m} Z_{i-1} + \tfrac{c\delta}{m}\tfrac{L}{m} - (c - 1 + \delta)\tfrac{L}{m})/(1 - \tfrac{c}{m}) + \delta \tfrac{L}{m} \\
&= (\tfrac{c}{m} Z_{i-1} - (c-1)\tfrac{L}{m})/(1 - \tfrac{c}{m}) \\
&= \tfrac{cY}{m}(1 - \tfrac{c}{m})^{-(i-m+\lfloor j \rfloor)}.
\end{aligned}$$

The second inequality follows because of inequality (8). The third inequality holds because at time $s_i - 1$, there is at least one machine in $\{M_{k+1}, \dots, M_m\}$ with colored load, i.e., $\Phi_{s_i-1} \ge \delta \frac{L}{m}$. Thus,

$$\begin{aligned}
L_{s_i} - \Phi_{s_i} &\ge L_{s_i-1} - \Phi_{s_i-1} + p_{s_i} - \Delta\Phi \\
&> Z_{i-1} + \tfrac{cY}{m}(1 - \tfrac{c}{m})^{-(i-m+\lfloor j \rfloor)} \\
&= X + Y(1 - \tfrac{c}{m})^{-(i-m+\lfloor j \rfloor)} \\
&= Z_i.
\end{aligned}$$

The induction step is complete because during time interval $(s_i, t_i]$ the inequality is maintained.

We finally have to prove

$$L > Z_{m+1}. \tag{9}$$

Our inductive proof shows $L_{t-1} - \Phi_{t-1} > Z_m$. Job J_t is scheduled on the least loaded machine and by assumption $l_1 + p_t > \frac{c(L_{t-1}+p_t)}{m}$, where $l_1 = (c-1+\epsilon)\frac{L}{m}$ is the load of the least loaded machine at time $t-1$, i.e., immediately before J_t is scheduled. Recall that at time t we remove the color from the load in $M2$'s schedule. Invariant I4 implies that the potential at time t must decrease by at least $\epsilon\frac{L}{m}$. Calculations identical to that in the inductive step show inequality (9). □

REFERENCES

[1] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts, *On-line routing of virtual circuits with applications to load balancing and machine scheduling*, J. ACM, 44 (1997), pp. 486–504.

[2] B. Awerbuch, Y. Azar, E.F. Grove, M.Y. Kao, P. Krishnan, and J.S. Vitter, *Load balancing in the L_p norm*, in Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1995, pp. 383–391.

[3] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra, *New algorithms for an ancient scheduling problem*, J. Comput. System Sci., 51 (1995), pp. 359–366.

[4] Y. Bartal, H. Karloff, and Y. Rabani, *A better lower bound for on-line scheduling*, Inform. Process. Lett., 50 (1994), pp. 113–116.

[5] Y. Bartal, S. Leonardi, A. Marchetti-Speccamela, J. Sgall, and L. Stougie, *Multiprocessor scheduling with rejection*, in Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1996, pp. 95–104.

[6] B. Chen, A. van Vliet, and G. Woeginger, *New lower and upper bounds for on-line scheduling*, Oper. Re. Lett., 16 (1994), pp. 221–230.

[7] U. Faigle, W. Kern, and G. Turan, *On the performance of on-line algorithms for particular problems*, Acta Cybernet., 9 (1989), pp. 107–119.

[8] G. Galambos and G. Woeginger, *An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling*, SIAM J. Comput., 22 (1993), pp. 349–355.

[9] M.R. Garay and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[10] R.L. Graham, *Bounds for certain multi-processing anomalies*, Bell System Tech. J., 45 (1966), pp. 1563–1581.

[11] D.R. Karger, S.J. Phillips, and E. Torng, *A better algorithm for an ancient scheduling problem*, J. of Algorithms, 20 (1996), pp. 400–430.

[12] R. Motwani, S. Phillips, and E. Torng, *Non-clearvoyant scheduling*, in Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 422–431.

[13] D.D. Sleator and R.E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

[14] D. Shmoys, J. Wein, and D.P. Williamson, *Scheduling parallel machines on-line*, in Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1991, pp. 131–140.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 474–491

TIGHT BOUNDS FOR ON-LINE TREE EMBEDDINGS[*]

SANDEEP BHATT[†], DAVID GREENBERG[‡], TOM LEIGHTON[§], AND PANGFENG LIU[¶]

Abstract. Tree-structured computations are relatively easy to process in parallel. As leaf processes are recursively spawned they can be assigned to independent processors in a multicomputer network. However, to achieve good performance the on-line mapping algorithm must maintain load balance, i.e., distribute processes equitably among processors. Additionally, the algorithm itself must be distributed in nature, and process allocation must be completed via message-passing with minimal communication overhead.

This paper investigates bounds on the performance of deterministic and randomized algorithms for on-line tree embeddings. In particular, we study trade-offs between computation overhead (load imbalance) and communication overhead (message congestion). We give a simple technique to derive lower bounds on the congestion that any on-line allocation algorithm must incur in order to guarantee load balance. This technique works for both randomized and deterministic algorithms. We prove that the advantage of randomization is limited. Optimal bounds are achieved for several networks, including multidimensional grids and butterflies.

Key words. graph embedding, randomized algorithm

AMS subject classifications. 65Y05, 68O22, 68O25

PII. S0097539796308710

1. Introduction.

Tree-structured computations arise naturally in diverse applications of the divide-and-conquer paradigm, for example, N-body simulations, the evaluation of functional expressions, backtrack searches, and branch-and-bound procedures. As the computation evolves, the corresponding tree grows and shrinks. Each node of the tree can recursively spawn subprocesses and communicate with its parent. The most significant feature of tree computations is that each node can spawn children independently of and therefore in parallel with other nodes.

How do we exploit this inherent parallelism on a multicomputer network containing as few as 2 or as many as 100,000 processors? For maximum speedup, the processes must be evenly distributed in the network. If neither the structure nor the size of the tree can be predicted at compile time, then a reasonable strategy might be to assign each newly spawned process to a randomly chosen processor. This gives even load distribution with high probability. Unfortunately, it also causes large congestion; many processes are spawned simultaneously and each sends startup information to its assigned processor over a long distance in the network. For any given network,

[*]Received by the editors September 3, 1996; accepted for publication (in revised form) October 2, 1997; published electronically October 5, 1999. The research of the first, second, and fourth authors was supported in part by Air Force grant AFOSR-89-0382, NSF grant CCR-88-07426, and NSF/DARPA grant CCR-89-08285. A preliminary version of this paper appears in the Proceedings of the 1991 ACM-SIAM Symposium on Discrete Algorithms.

http://www.siam.org/journals/sicomp/29-2/30871.html

[†]Bellcore, 445 South Street, Morristown, NJ 07960 (bhatt@bellcore.com). A portion of this work was done while this author was visiting Caltech. The research of this author was supported in part by DARPA order number 6202 and was monitored by ONR under contract N00014-87-K-0745.

[‡]Sandia National Laboratories, Albuquerque, NM 87185 (dsgreen@cs.sandia.gov).

[§]Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139 (ftl@math.mit.edu). The research of this author was supported by Air Force grant AFOSR-89-0271, Army contract DAAL-03-86-K-0171, and DARPA contracts N00014-89-J-1988 and N00014-87-K-825.

[¶]National Chung Cheng University, Computer Science and Engineering, Chiayi 621, Taiwan (pangfeng@cs.ccu.edu.tw).

one might reasonably suspect a trade-off between computational overhead (poor load distribution) and communication overhead (high congestion).

For the boolean hypercube and its derivative networks, e.g., the butterfly and cube-connected-cycle networks, simple and efficient algorithms have recently been developed and analyzed [1, 4]. These algorithms map processes to randomly chosen processors, but the random choice is confined to processors within a short distance from the spawning processor. In particular, every N-node tree can be dynamically spawned within the N-node hypercube so that, with high probability, each processor receives $O(1)$ tree nodes and such that the maximum congestion in the network is $O(1)$ [4]. Using this randomized tree embedding technique for butterfly, Ranade showed that optimal speedup can be achieved for backtrack search problems [6].

For the complete network Karp and Zhang showed that by assigning newly generated branch-and-bound tree nodes to random processors, the parallel branch-and-bound algorithm achieves linear speedup with high probability [3, 10]. The analysis in [3] was later simplified by Ranade [5].

In this paper we extend our study of on-line embeddings to additional multicomputer networks. We derive lower bounds on the congestion that any on-line, deterministic algorithm must incur in order to guarantee load balance. The bounds are tight, to within constant factors, for several networks, including multidimensional grids and butterflies. The deterministic lower bound for butterflies contrasts sharply with previous randomized upper bounds [4]. We also adapt the techniques of [4] to develop a randomized algorithm for grids and show that its performance is better than the deterministic lower bound. Finally, we give tight lower bounds on the expected congestion for randomized algorithms.

Our lower bounds are all based on the simpler problem of allocating tree nodes to two processors. It is well known that bounded-degree trees have small separators, and this seems to suggest that only a small amount of communication should be needed to distribute tree nodes evenly. Nevertheless, we show that, even in the simple case of two processors, large communication overhead is inevitable for balanced on-line embeddings.

The remainder of this paper is organized as follows. Section 2 defines the on-line embedding model and the cost measures studied in this paper. Section 3 presents lower bounds for deterministic algorithms. Section 4 discusses a deterministic off-line algorithm that embeds dynamic trees under a strict balance requirement. Section 5 presents randomized algorithms with improved performance. Section 6 presents lower bounds for randomized algorithms. Section 7 concludes the paper.

2. Models. We model the problem of growing trees on networks as an on-line embedding problem. Informally, the on-line tree embedding problem may be described as follows. We start with the root of a binary tree placed at some node of a multicomputer network. At any instant, each node with 0 or 1 child may choose to spawn a new child. Each newly spawned child must be assigned to a network node, and a path in the network must be chosen from the parent node to the child node. Furthermore, the decision of where to place the newly spawned child must be entirely local, i.e., the embedding algorithm can perform only local computations and no global data structures are allowed.

The on-line embedding algorithm is further constrained in that processes cannot migrate. Once embedded, a tree node cannot subsequently be moved to a different node in the network. These requirements are critical for fine-grain multicomputers [8]. Disallowing process migration is a common policy in practice; migration can conceivably lead to thrashing and thus cripple a large multicomputer network.

This framework for maintaining a dynamically evolving tree differs substantially from a model studied recently by Wu and Kung [9]. The latter model allows for a global data structure which all processors can access simultaneously. Moreover, newly spawned nodes do not need to be assigned processors immediately; the allocation can be deferred until some processor becomes idle. While the arguments and techniques used to prove tight upper and lower bounds are not affected drastically, the resulting bounds are substantially different.

2.1. Growth sequences. Although many tree nodes can spawn children simultaneously, it will suffice for our lower bound arguments to assume that the tree grows one node at a time. Formally, an instance of the on-line tree embedding problem is a sequence of directed edges, $e_i = (w_j, w_i)$, $1 \leq j < i$, in which w_i is the ith node spawned and has parent w_j. Node w_1 is the root. A *growth sequence* is a sequence of edges such that for each k the set $\{e_i : i \leq k\}$ of edges forms a directed tree with each node having outdegree at most two.

The restriction to binary trees is arbitrary, any bounded degree would do, but since the networks we consider have bounded degree, trees with unbounded fanout lead, uninterestingly, to poor worst-case performance.

2.2. Embeddings and their quality. An embedding of a tree T in a network H is a mapping of nodes of T to vertices of H, together with a mapping from edges of T to paths in H. The mapping of nodes is not required to be one-to-one; multiple processes can share the same processor. An on-line embedding induced by a growth sequence is a sequence of embeddings in which each embedding is an extension of the previous one. This enforces the policy of disallowing process migration.

Two standard measures of the quality of an embedding are dilation and congestion. The *dilation* of an embedding is the maximum path length in the network corresponding to an edge of T. The *congestion* of an embedding is the maximum number of edges of T whose images traverse any edge in H.

An embedding of a k-node tree is said to be *α-balanced*, $\alpha \geq 1$, if no more than $\alpha\lceil k/P \rceil$ tree nodes are assigned to any of the P nodes of H. The overloading factor, α, is typically a small constant, independent of P, the size of the network.

In some tree-structured computations, the computational activity is limited to the leaves of the tree. In such cases, it is important only to evenly distribute the leaves within the network. An on-line embedding of a tree with ℓ leaves is said to be *α-balanced on the leaves*, $\alpha \geq 1$, if no more than $\alpha\lceil \ell/P \rceil$ leaves are assigned to any of the P processors of the network.

Finally, an on-line embedding for a growth sequence ρ is *continuously α-balanced* if for every i the ith embedding (induced by the first i elements of ρ) is α-balanced. On the other hand, if only the final embedding in the sequence is α-balanced, then we say that the on-line embedding is *terminally α-balanced*. Obviously, terminal balance is a weaker requirement than continuous balance. Also note that in order to distinguish the two balance requirements under the on-line model, the length of the growth sequence will be given to the embedding algorithms.

3. Deterministic lower bounds. For a given embedding of a tree T, an edge $e = (u, v)$ in T is said to be a *cut edge* if the end points u and v of e are mapped to distinct processors. Each cut edge corresponds to a child spawned at a remote processor, thus requiring communication through the network. The number of cut edges can critically affect congestion.

3.1. Lower bound on two processor cut edges. Given a deterministic on-line algorithm we show how to construct an N-node binary tree for which the algorithm either requires $\Theta(N)$ cut edges or fails to achieve terminally α-balanced for $\alpha < 2$. We describe an *adversary* which progressively grows a tree such that if the algorithm does not use many cut edges, then the final embedding will not be balanced.

On-line allocation for two processors is equivalent to coloring each tree node either black or white. Without loss of generality, suppose that the root is colored *black*. The adversarial tree construction works in phases. In the first phase, we spawn two children from every black node. This phase ends when all the leaves are white. Phase 2 continues with this strategy of spawning from one color only, spawning two children from every white node until all the leaves are black. The tree is grown in this "layer-by-layer" method, producing leaves of alternating colors at the ends of successive phases.

Since the two-coloring algorithm is deterministic, the color committed on-line to each newly spawned node is "known," i.e., computable by the adversary constructing the growth sequence. Thus for each possible deterministic algorithm the adversary can produce a well-defined tree.

We will show that the number of cut edges grows linearly with the size of the adversarial tree under the requirement of terminal balance. The proof is based on the following observation. At the end of every phase, each leaf is connected to its parent by a cut edge. Since every internal node has two children, more than half the edges of the tree (those incident to leaves) are cut edges.

We use the following terminology. A tree node is in layer i if the path from the root to the node contains $i-1$ cut edges. Let n_i be the number of nodes in layer i. Each node spawned during phase i lies either in layer i or in layer $i+1$.

LEMMA 3.1. *For every deterministic algorithm $\mathcal{A}$, $\alpha < 2$, and $\epsilon < (2-\alpha)/4$ there exists a growth sequence ρ of length $N > \alpha/(2-\alpha-4\epsilon)$ such that if the on-line embedding of ρ is terminally α-balanced, then the number of cut edges is greater than ϵN.*

Proof. We consider two cases: either the embedding of ρ is completed before phase 1 terminates or the embedding is completed in a later phase.

The embedding is completed before phase 1 *terminates.* In this case every white node is a leaf and the number of cut edges equals the number of white leaves. In order to maintain terminal balance, there must be at least $N - \alpha\lceil N/2\rceil$ white nodes. This number is greater than $N-\alpha(N/2+1/2)$, which exceeds ϵN when $N > \alpha/(2-\alpha-4\epsilon)$. Therefore there are at least ϵN cut edges.

The embedding is completed during phase k, $k > 1$. Let n_i be the number of nodes in layer i. We again distinguish between two subcases: either $\sum_1^{k-1} n_i \geq \epsilon N$ or $\sum_1^{k-1} n_i < \epsilon N$.

When $\sum_1^{k-1} n_i \geq \epsilon N$.

We first observe that every node in layer $k-1$ is an internal node. Together with the fact that every internal node has two children, we conclude that there are $1 + \sum_1^{k-1} n_i$ cut edges, each connecting a node in layer $k-1$ to a node in layer k. Thus, in this subcase the number of cut edges is greater than ϵN.

When $\sum_1^{k-1} n_i < \epsilon N$.

In this subcase $n_k+n_{k+1} > N(1-\epsilon)$. From the terminal balance condition, we also have $n_k < \alpha\lceil N/2\rceil$. These two imply that $n_{k+1} > N(1-\epsilon) - \alpha\lceil N/2\rceil > N(1-\epsilon-\alpha/2) - \alpha/2 > \epsilon N$ when $N > \alpha/(2-4\epsilon-\alpha)$. Since the embedding terminated during phase k, and before phase $k+1$ began, it follows that every node in layer $k+1$ is a leaf that is connected to its parent by a cut edge. From the preceding argument, it again follows that the number of cut edges is greater than ϵN. □

Lemma 3.1 implies that every deterministic, on-line, load-balancing algorithm requires a linear number of cut edges on some finite sequence. This lower bound for terminally balanced algorithms holds for continuously balanced algorithms as well.

3.2. Multiple processors. As we show in this section, Lemma 3.1 can be extended in a straightforward manner to any bounded number of processors. With P colors (processors) instead of two, we use a construction similar to the one in Lemma 3.1. The only difference is that each phase grows only one subtree, rooted at a leaf grown in the previous phase. Each phase ends when all the leaves grown during that phase are colored differently from the root of the subtree. We also use the same terminology. A tree node is in the layer i if the path from the node to the root contains $i-1$ cut edges. The symbol n_i denotes the number of nodes in layer i.

LEMMA 3.2. *For every deterministic algorithm $\mathcal{A}$, $P \geq 2$, $\alpha < P$, and $\epsilon < \frac{1}{2}(1-\alpha/P)$ there exists a growth sequence ρ of length $N > \alpha/(1-2\epsilon-\alpha/P)$ such that if the on-line embedding of ρ is terminally α-balanced among P processors, then the number of cut edges is greater than ϵN.*

Proof. We consider two cases: the construction of ρ ends either in the first phase or in a later phase.

The construction ends in the first phase. In this case every node in the second layer has a cut edge to its parent. In order to maintain terminal α-balance, the number of nodes in the second layer must be at least $N - \alpha\lceil N/P\rceil$. Therefore the number of cut edges is at least $N-\alpha\lceil N/P\rceil > N-\alpha(N/P+1) > \epsilon N$ when $N > \alpha/(1-2\epsilon-\frac{\alpha}{P})$.

The construction ends during the kth phase for $k > 1$. Let T_k be the subtree constructed during the kth phase, and let $r \in T_k$ be the root of T_k. T_k is partitioned into two subsets, L and R. L is the set of nodes which are in layer $k+1$ and $R = T_k - L$. Notice that L contains only leaves of T_k since ρ ends in phase k.

Every node in the tree induced by $\rho - T_k$ has 0 or 2 children (except the parent of r); hence the number of leaves in $\rho - T_k$ is $\frac{1}{2}(N - |T_k|)$. Additionally, every *leaf* in $(\rho - T_k) \cup L$ has a cut edge to its parent, so the number of cut edges is at least $\frac{1}{2}(N-|L|-|R|) + |L| \geq \frac{1}{2}(N-|R|)$. Therefore, by bounding the size of R we get a lower bound on the number of cut edges. The size of R is at most $\alpha\lceil\frac{N}{P}\rceil$ because it consists of only one color (the color of r). Therefore the number of cut edges is at least $\frac{1}{2}(N-|R|) \geq \frac{1}{2}(N-\alpha\lceil\frac{N}{P}\rceil) \geq \frac{1}{2}(N-\alpha(\frac{N}{P}+1)) \geq \frac{N}{2}(1-\alpha/P) - \frac{\alpha}{2} > \epsilon N$ when $N > \alpha/(1-\frac{\alpha}{P}-2\epsilon)$. □

As mentioned in the previous section, in some tree-structured computations it is important to evenly distribute only the leaves of the tree. We next generalize Lemma 3.2 to get a lower bound on the number of cut edges when only the leaves need to be terminally balanced.

LEMMA 3.3. *For every deterministic algorithm $\mathcal{A}$, $P \geq 2$, $\alpha < P$, and $\epsilon < \frac{1}{2}(1-\frac{\alpha}{2P})$ there exists a growth sequence ρ of length $N > 2\alpha/(1-\frac{\alpha}{P}-2\epsilon)$ such that if the on-line embedding of ρ is terminally α-balanced on the leaves, then the number of cut edges is greater than ϵN.*

Proof. We use the adversarial strategy in Lemma 3.2 to grow a tree with N nodes, $N > 2\alpha/(1-\frac{\alpha}{P}-2\epsilon)$. In order to simplify the calculation, we assume, without loss of generality, that N is even so that there are exactly $N/2$ leaves in the N-node tree.

In the N-node tree grown by the adversary, let S denote the set of leaves which have the same color as their parents. Every leaf that has the same color as its parent must have been spawned in the last phase. Every leaf not in S is connected to its parent by a cut edge so that the number of cut edges is at least $N/2 - |S|$.

Since all leaves in S are grown in the same phase, they have the same color as the root of the subtree that is grown during that phase. In order to satisfy the terminal α-balance requirement on the leaves, the size of S is at most $\alpha\lceil \frac{N}{2P} \rceil$. Hence the number of cut edges is at least $N/2 - |S| \geq N/2 - \alpha(\frac{N}{2P} + 1) \geq N(\frac{1}{2} - \frac{\alpha}{2P}) - \alpha \geq \epsilon N$ when $N > 2\alpha/(1 - \frac{\alpha}{P} - 2\epsilon)$. $\square$

3.3. Lower bounds on network congestion and dilation. The lower bound of Lemma 3.2 on the number of cut edges required by any deterministic algorithm for P processors can be used to derive lower bounds on the worst-case congestion and dilation in different networks. In this section we derive tight lower bounds for multidimensional grids and butterfly networks.

We partition the given P-processor network into B blocks of equal size. We refer to an edge that connects processors in different blocks as a *cross-link*. An α-balanced embedding of an N-node tree assigns at most $\alpha N/B$ tree nodes per block; for convenience we let N be a multiple of P so that all divisions are exact. For any specified α, we choose B such that $\alpha/B < 1$ so that each block in the partition can receive at most a small fraction of the tree nodes.

By Lemma 3.2, in the worst case, $\Omega(N)$ edges of the tree must be cut. If there are C cross-links in the network partition, then some cross-link must accommodate $\Omega(N/C)$ cut edges of the tree, causing congestion $\Omega(N/C)$. Observe that an upper bound on C yields a lower bound on congestion.

By removing the edges in every $\sqrt{P}/B$th column, the P-node two-dimensional grid can be partitioned into B blocks of equal size with $B\sqrt{P}$ cross-links. Consequently, the congestion is $\Omega(N/\sqrt{P})$. For the P-node butterfly $C = O(P/\log P)$ when the edges at every $O((\log P)/B)$th level are removed; therefore, the congestion is $\Omega(\frac{N \log P}{P})$.

It is equally easy to obtain lower bounds on dilation. For a partition of the network into blocks, define the *boundary* of a block to be the set of processors incident to cross-links. Furthermore, let n_d be the number of processors within distance d from the boundary. Observe that in any embedding with dilation d every tree node incident to a cut edge must lie within distance d from a boundary node of some block. Since there are $\Omega(N)$ cut edges in the worst case, the total number of network nodes within distance d of boundary nodes must be large enough to accommodate $\Omega(N)$ tree nodes.

By removing the edges in every $\sqrt{P}/B$th column, the P-node two-dimensional grid can be partitioned into B blocks of equal size so that $n_d = \Theta(d\sqrt{P})$ for each block. From the balance constraint each processor has at most $\alpha\lceil N/P \rceil$ tree nodes. In order to accommodate $\Omega(N)$ tree nodes, it follows that $d = \Omega(\sqrt{P})$. Similarly, by removing the edges at every $O((\log P)/B)$th level, the P-node butterfly can be partitioned so that $n_d = \Theta(dP/\log P)$. It follows that for the butterfly, $d = \Omega(\log P)$.

These lower bounds can be matched by deterministic algorithms, which map tree nodes to processors in a round-robin manner so that the tree nodes are evenly distributed. First, we observe that the lower bound on dilation is already a constant factor of the diameters of the corresponding networks; therefore the dilation upper bounds immediately follow. Next, we choose the route that connects adjacent tree nodes. For the two-dimensional grids the algorithm picks the direct route from a parent tree node to its children that makes at most one turn in the grid. As a result a tree edge whose image goes through a horizontal network link will have at least one endpoint embedded in the same row of processors as the communication link. Since the number of tree nodes embedded into a row of processors is $O(N/\sqrt{P})$ from the balance requirement, we conclude that the congestion on every horizontal link is $O(N/\sqrt{P})$. A similar argument holds for vertical links, and the congestion bound follows. For butterfly networks we use standard off-line routing algorithms to meet the $O(\frac{N \log P}{P})$ congestion bounds.

The lower bounds on congestion and dilation for two-dimensional grids discussed above extend in a straightforward manner to grids with multiple dimensions. We summarize our observations in the following theorem.

THEOREM 3.4. *For every deterministic algorithm $\mathcal{A}$ and constant $\alpha > 1$, there exists a growth sequence ρ of length N (N sufficiently large) such that if the on-line embedding of ρ is terminally α-balanced, then* (1) *on the P-node, k-dimensional grid (constant k) the dilation is $\Omega(P^{1/k})$ and the congestion is $\Omega(N/P^{1-1/k})$, and* (2) *on the P-node butterfly the dilation is $\Omega(\log P)$ and the congestion is $\Omega(\frac{N \log P}{P})$.*

3.4. Infinite growth sequences. In the previous sections we have shown that every deterministic, balanced (either terminally or continuously) algorithm is required to make $\Omega(N)$ cuts on some growth sequences of sufficiently large length N. Turning the question around, one is led to ask if there is a growth sequence which is universally bad in the sense that every balanced algorithm is forced to make linearly many cuts.

As we will see in the next section, for every length-N growth sequence there is an *off-line* algorithm which is continuously α-balanced between two processors for $1 < \alpha < 2$ and requires only $O(\log^2 N)$ cut edges. However, in this section we show the existence of an infinite growth sequence on which every *on-line* continuously balanced algorithm requires, infinitely often, linearly many cut edges. Formally, we establish the following theorem.

THEOREM 3.5. *There exists an infinite growth sequence ϕ^* such that for every overloading factor $1 < \alpha < 2$ every on-line deterministic algorithm $\mathcal{A}$ which produces a continuously α-balanced embedding on two processors will, on infinitely many prefixes $\phi_1, \phi_2, \ldots$ of lengths $N_1, N_2, \ldots$, make at least ϵN_i cuts with $\epsilon = \alpha(2-\alpha)/4(2+\alpha)$.*

Before proceeding to the proof of the above theorem, we explain the idea behind the proof. The infinite sequence is constructed using a standard diagonalization argument that invokes Lemma 3.1 repeatedly. We begin with an enumeration, $(\mathcal{A}_1, \mathcal{A}_2, \ldots)$ of all algorithms as a sequence in which every algorithm appears infinitely often.

The universal growth sequence is constructed in stages. The portion constructed in phase i is denoted ϕ_i, and the entire growth sequence at the end of phase i is denoted $\mathcal{G}_i$ so that $\mathcal{G}_0$ consists of the root of the tree and $\mathcal{G}_i = \mathcal{G}_{i-1} \circ \phi_i$ for $i > 0$ (the symbol $\circ$ denotes concatenation). The idea is to construct ϕ_i such that algorithm $\mathcal{A}_i$ requires linearly many cuts on $\mathcal{G}_i$. Since the number of cuts made by $\mathcal{A}_i$ on the initial prefix $\mathcal{G}_{i-1}$ can be very small, we make ϕ_i large enough so that the number of cut edges forced in ϕ_i is a fraction of the length of $\mathcal{G}_i$. This latter idea is formalized in the following lemma.

LEMMA 3.6. *Suppose that $\mathcal{A}$ is a continuously α-balanced, deterministic algorithm ($\alpha < 2$) and that ϕ is a growth sequence of length n. Then for every $N > 2\alpha n/(2-\alpha)$ there is a growth sequence $\phi \circ \rho$ of length $n + N$ on which $\mathcal{A}$ makes at least $\epsilon(N+n)$ cut edges, where $\epsilon = \alpha(2-\alpha)/4(2+\alpha)$.*

Proof. For ease of exposition, we assume that $N + n$ is even. We use Lemma 3.1 to grow the sequence ρ of length N from a leaf of ϕ. We define layers on the subtree induced by ρ as in Lemma 3.1 and use n_i to denote the number of nodes in layer i of ρ.

We consider two cases: either the embedding of ρ is complete before phase 1 terminates or the embedding is completed during phase k, $k > 1$.

The embedding is complete before phase 1 *terminates.* In this case, every node in the second layer is incident to a cut edge, so the number of cut edges in ρ equals n_2. In order to maintain α-balance, the number of nodes n_2 must be at least $N - \alpha\lceil \frac{N+n}{2} \rceil$. Therefore, the number of cut edges is at least $N - \alpha(\frac{N+n}{2})$. Since $N > 2\alpha n/(2-\alpha)$, we have that $N > 2\alpha(N+n)/(2+\alpha)$ so that the number of cut edges is greater than $\alpha(2-\alpha)(N+n)/2(2+\alpha) = 2\epsilon(N+n)$.

The embedding is complete after phase 1 *terminates.* We further divide the second case into two subcases, depending on whether $\sum_1^{k-1} n_i \geq \epsilon(N+n)$ or $\sum_1^{k-1} n_i < \epsilon(N+n)$.

When $\sum_1^{k-1} n_i \geq \epsilon(N+n)$.

In this subcase, every node in the first $k-1$ layers is an internal node and has two children, so the number of cut edges between layers k and $k-1$ equals $\sum_1^{k-1} n_i + 1$, which exceeds $\epsilon(N+n)$.

When $\sum_1^{k-1} n_i < \epsilon(N+n)$.

In this subcase $n_k + n_{k+1} > N - \epsilon(N+n)$. From the balance condition we have that $n_k \leq \alpha\lceil \frac{N+n}{2} \rceil$ so that $n_{k+1} > N - \epsilon(N+n) - \frac{\alpha}{2}(N+n)$. Using the inequality $N > 2\alpha(N+n)/(2+\alpha)$, it follows that $n_{k+1} > (\frac{2\alpha}{2+\alpha} - \epsilon - \frac{\alpha}{2})(N+n)$, which exceeds $\epsilon(N+n)$. ∎

Proof of Theorem 3.5. Theorem 3.5 follows from applying Lemma 3.6 in the diagonalization argument outlined earlier with $\epsilon = \alpha(2-\alpha)/4(2+\alpha)$. ∎

4. Deterministic algorithm for off-line model.

In the off-line model the entire growth sequence is given in advance. The off-line embedding algorithm can preprocess this sequence to produce a sequence of embeddings. Since bounded-degree trees have small separators, terminal balance is easy to guarantee between two processors. The more interesting case is when the off-line algorithm is required to be continuously balanced.

Our off-line upper bound contrasts sharply with the on-line lower bound. An N-node binary tree can be partitioned into two subsets, each containing at least $\lfloor N/3 \rfloor$ nodes, by removing a single edge. Therefore an embedding algorithm can, by making a single cut edge, guarantee terminal $\frac{4}{3}$ balance between two processors. Of course, this is possible only if each edge has a distinct identity which can be recognized by the algorithm when the child node adjacent to the edge is spawn.

Less trivially, we can show that for every α, every length-N growth sequence can be *continuously* α-balanced between two processors with $O(\log^2 N)$ cuts for any $1 < \alpha < 2$. This is achieved using multicolor bisectors [2].

LEMMA 4.1 (see [2]). *The nodes of every N-node binary tree, each of whose nodes has one of k distinct colors, can be bisected into two equal-size (to within one) subsets by removing at most $k \log N$ edges. Furthermore, for each of the k colors the set of all nodes of the same color are divided equally (to within one) between the two subsets in the bisection.*

The algorithm first partitions the input growth sequence ρ into groups of consecutively spawned nodes. The partition $\{V_i\}$ with V_i of size N_i is determined as follows: The first N_1 nodes in the growth sequence are placed in the first partition V_1. The number N_1 is chosen to be an even number greater than $2/\gamma$, where $\gamma = \frac{2\alpha-2}{2-\alpha}$. The second group V_2 contains the next N_2 nodes; N_2 is chosen to be the largest even number no greater than γN_1. In general, V_i contains N_i tree nodes, where N_i is the largest even number no greater than $\gamma \sum_{j=1}^{i-1} N_j$.

Once the partition is formed, the off-line algorithm applies Lemma 4.1 to the input tree in which every node in V_i has color i. When the tree is processed on-line, edge by edge, the new child is allocated depending on whether the current edge lies in the separator set S. If it does, the child is placed in the remote processor; otherwise, it is placed in the same processor as the parent.

We now show that this algorithm maintains continuous α-balance. We first bound the number of groups in the partition. To this end, define a sequence, $\{M_i\}$, such that $M_1 = N_1$ and $M_i = \gamma \sum_{j=1}^{i-1} M_j$ for all $i \geq 2$.

LEMMA 4.2. *$M_i = \gamma N_1 (1+\gamma)^{i-2}$ for $i \geq 2$.*

Proof. This lemma is proved by induction on i. Since $M_2 = \gamma N_1$, the basis $i = 2$ is established. For the inductive step

$$\begin{aligned} M_{i+1} &= \gamma \sum_{j=1}^{i} M_j \\ &= \gamma \sum_{j=1}^{i-1} M_j + \gamma M_i \\ &= M_i + \gamma M_i \\ &= (1+\gamma) M_i \\ &= \gamma N_1 (1+\gamma)^{i-1}. \qquad \square \end{aligned}$$

Since for every $i \geq 2$, N_i equals the largest even integer no greater than M_i, we can bound each N_i from below as follows.

LEMMA 4.3. *$N_i > M_i - 2(1+\gamma)^{i-2}$ for $i \geq 2$.*

Proof. This lemma is proved by induction on i. Again, $N_2 > M_2 - 2$ by definition, so the basis is established. For the inductive step,

$$\begin{aligned} N_{i+1} &> \gamma \sum_{j=1}^{i} N_j \, - \, 2 \\ &> \gamma \sum_{j=1}^{i} M_j - \gamma \sum_{j=2}^{i} 2(1+\gamma)^{j-2} \, - \, 2 \\ &= \gamma \sum_{j=1}^{i} M_j - 2\gamma \sum_{j=0}^{i-2} (1+\gamma)^j \, - \, 2 \\ &= M_{i+1} - 2(1+\gamma)^{i-1}. \qquad \square \end{aligned}$$

LEMMA 4.4. *The above off-line algorithm partitions any length-N growth sequence into $O(\log N)$ groups and produces $O(\log^2 N)$ cut edges.*

Proof. From Lemmas 4.2 and 4.3, we conclude that $N_i > (\gamma N_1 - 2)(1+\gamma)^{i-2}$ for all $i \geq 2$. With $N_1 > 2/\gamma$, it follows that the sequence $\{N_i\}$ increases exponentially, so the number of groups in the partition is bounded by $O(\log N)$. Since the number of colors is bounded by $O(\log N)$, the resulting bisector S contains $O(\log^2 N)$ edges as seen in Lemma 4.1. $\square$

We formally establish the continuously balancing property for the above off-line algorithm in the following theorem.

THEOREM 4.5. *For every growth sequence ρ of length N and $1 < \alpha < 2$, the off-line algorithm requires at most $O(\log^2 N)$ cut edges and is continuously α-balanced between two processors for all prefixes of ρ of length $M > N_1$ (where N_1 is as defined above).*

Proof. The number of cut edges is bounded in Lemma 4.4. Since exactly half of the tree nodes in every V_i are assigned to each processor, the numbers of nodes in both processors are always equal at the end of every V_i; therefore we need to show only that the algorithm gives a continuously α-balanced embedding while processing every V_i. First, every V_j, $1 \le j \le i-1$, is perfectly balanced, so the number of white nodes equals the number of black nodes just before V_i is processed. Second, the number of white nodes in V_i equals the number of black nodes. Independent of the order in which the white and black nodes within V_i appear, we claim that α-balance is always guaranteed. To see this, consider the extreme case in which all the black nodes appear before the white nodes. The total number of black nodes after half the nodes in V_i appear equals $\frac{1}{2}(\sum_{j=1}^{i-1} N_j) + \frac{1}{2}N_i$. We claim that this is no more than the quantity allowed under α-balance, which equals $\frac{\alpha}{2}(\sum_{j=1}^{i-1} N_j + N_i/2)$. To see this, consider the difference

$$\begin{aligned}
&\frac{\alpha}{2}\left(\sum_{j=1}^{i-1} N_j + N_i/2\right) - \frac{1}{2}\left(\sum_{j=1}^{i-1} N_j\right) - \frac{1}{2}N_i \\
&= \frac{\alpha-1}{2}\sum_{j=1}^{i-1} N_j - \frac{1-\alpha/2}{2}N_i \\
&\ge \frac{\alpha-1}{2}\sum_{j=1}^{i-1} N_j - \frac{2-\alpha}{4}\cdot\frac{2\alpha-2}{2-\alpha}\sum_{j=1}^{i-1} N_j \\
&= 0. \qquad \square
\end{aligned}$$

5. Randomized algorithms.

5.1. *n*-way balancing. The key tool for our randomized algorithms is the *n-way balancing transformation* [4], which evenly distributes a binary tree of size N into a ring of $n = \log N$ processors. The n-way balancing transformation works as follows. A tree node is *distinguished* if it is in level $i \equiv 0 \bmod n/3$. For each distinguished node v we pick a random number $S(v)$ between 0 and $n/3$ as the *stretch count.* The transformation inserts a single dummy node in each edge in the subtree of height $S(v)$ rooted at v. The resulting tree after this transformation is denoted by $B(T)$. Define *level set* i to be the set of all tree nodes in a level congruent to i modulo n. Let the processors of the ring be p_0, p_1,..., p_{n-1} in clockwise order. The algorithm embeds the tree nodes in level set i of $B(T)$ into processor p_i. Figure 5.1 illustrates an example of six-way balancing.

This transformation embeds every sufficiently large binary tree into a ring evenly with high probability and with dilation two, since at most one dummy node will be inserted into each edge in T.

LEMMA 5.1 (see [4]). *With probability $1 - N^{-c}$, $c > 0$, the above $\log N$-way balancing algorithm dynamically embeds every N-node tree into the $\log N$-node ring so that $O(N/\log N)$ tree nodes are mapped to any ring node.*

When the number of processors in the ring is less than $\log N$, we instead map a virtual $\log N$-length ring onto our smaller ring. In particular, with two processors the

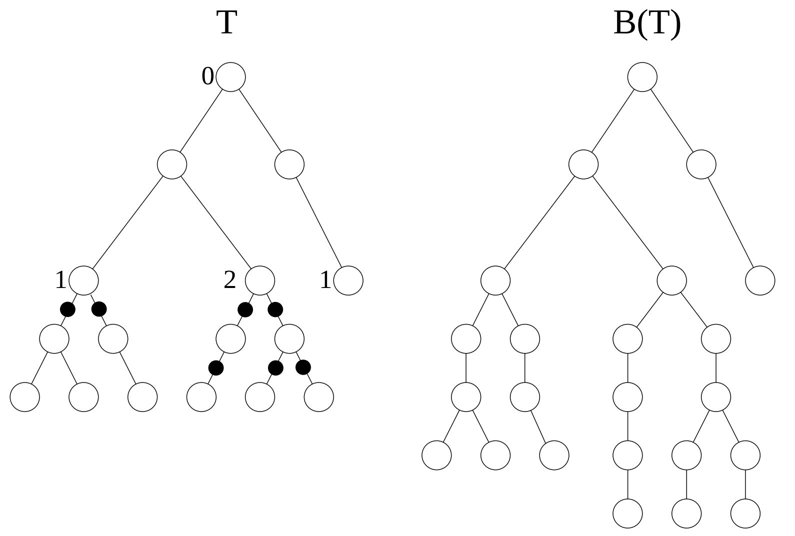


FIG. 5.1. *A 6-way balancing example taken from* [6]. *The solid circles indicate where the dummy nodes will be inserted, and the numbers next to the distinguished nodes are the stretch counts.*

first half of the virtual ring is mapped to one node and the second half to the other. Since the dilation of the mapping on the virtual ring is two, only the eight nodes of the virtual ring within distance two of the breaks contribute to the cut edges. Since these eight nodes contain at most $O(N/\log N)$ tree nodes, the number of cut edges is bounded by $O(N/\log N)$.

When the number of processors in the ring exceeds $\log N$, we divide the ring into $\log N$ groups, each containing an equal number of consecutive processors. We use $\log N$-way balancing to assign tree nodes to groups, and a deterministic strategy is used within each group to distribute nodes evenly in a group. For the P-node ring, the dilation is bounded by $O(P/\log N)$ because each tree edge traverses at most three groups. The bound on dilation is optimal; any mapping of the N-node complete binary tree on the P-node ring requires dilation $\Omega(P/\log N)$. From the dilation result, for any link l in the ring, only those processors that are within distance $O(P/\log N)$ can contribute congestion on l. Each processor can have at most $O(N/P)$ tree nodes in a balanced embedding, and each tree node has at most three edges; therefore, the congestion is $O(N/\log N)$ for any link in the ring. The congestion is a factor of $\log N$ less than the worst-case lower bound for deterministic on-line algorithms and matches the lower bound, shown in section 6, on the expected value of the congestion for any randomized algorithms.

5.2. Cut-edges reduction. Suppose that we wish to equitably allocate the tree nodes on-line among P processors with the goal of minimizing the total number of cut edges. When P is any fixed constant, the strategy mentioned above uses $O(N/\log N)$ cuts. However, when P is $\log N$, each of the $N-1$ edges is cut. By modifying the $\log N$-way balancing technique slightly, we can reduce the total number of cuts to $O(N/\log \frac{N}{P})$. As we will see in the next section, this bound is the best possible.

We modify the algorithm so that we choose ϵ so $1 > \epsilon > 0$, and for each node in every $\epsilon \log \frac{N}{P}$th level set of the transformed tree, we map it to a processor chosen uniformly at random among the P processors. We call these nodes *leaders*. Those nodes not in these level sets are embedded into the processor where their parents are embedded. We show that with high probability this algorithm gives balanced embedding and optimal number of cut edges when $P = O(N/\log^{\frac{1}{1-\epsilon}} N)$.

We need the following lemma [4], which states that with high probability the sum of independent random variables is at most a constant times their expected sum.

LEMMA 5.2 (see [4]). *Let $x_1, \dots, x_m$ be independent random variables in the range from 0 to V with $E(x_i) = \mu_i$. Let $X = \sum x_i$, and let $\mu = \sum_{i=1}^m \mu_i = E(X)$. Then for any constant β, $Pr(X \geq \beta\mu) \leq \exp(-\beta\frac{\mu}{V})$.*

THEOREM 5.3. *With probability $1 - N^{-c}$, $c > 0$, the above algorithm dynamically embeds every N-node binary tree into P processors, where $P = O(N/\log^{\frac{1}{1-\epsilon}} N)$ so that $O(N/P)$ tree nodes are mapped to any processor and the number of cut edges is $O(N/\log\frac{N}{P})$.*

Proof. First we estimate the number of cut edges. Since only edges at every $\epsilon\log\frac{N}{P}$th level of $B(T)$ are cut, from Lemma 5.1 the total number of cut edges is $O(N/\log\frac{N}{P})$. This bound matches the lower bound on the expected number of cut edges, as we will see in the next section.

Let $l = \epsilon\log\frac{N}{P}$. The transformed tree is divided into m subtrees, $t_1, \dots, t_m$, by cutting the edges between levels $k\,l$ and $k\,l-1$ (for any integral k). Each t_i is rooted at a leader, and every node in t_i will follow this leader to a processor. Let M_i be the number of nodes in t_i and $x_{i,p}$ be the number of tree nodes in t_i that will be mapped into a particular processor p. We denote the total number of tree nodes assigned to p by $X_p = \sum_{i=1}^m x_{i,p}$.

The contributions from each subtree to any processor are independent and occur with equal probability. Therefore for all i, $1 \leq i \leq m$, $E(x_{i,p}) = M_i/P$ and $E(X_p) = N/P$. Moreover, all $x_{i,p}$'s are mutually independent, and since each tree contains at most the number of nodes in a complete binary tree of height $\epsilon\log\frac{N}{P}$, all $x_{i,p} \leq 2^{\epsilon\log\frac{N}{P}} = \frac{N}{P}^{\epsilon}$. Applying Lemma 5.2 we get

$$\begin{aligned} Pr\left(X \geq \alpha\frac{N}{P}\right) &= Pr(X \geq \alpha E(X)) \\ &\leq \exp\left(-\alpha\frac{(\frac{N}{P})}{(\frac{N}{P})^{\epsilon}}\right) \\ &= \exp\left(-\alpha(\frac{N}{P})^{1-\epsilon}\right) \\ &\leq \exp(-c'(\log^{\frac{1}{1-\epsilon}} N)^{1-\epsilon}) \\ &< N^{-c'}. \end{aligned}$$

With probability $1 - PN^{-c'} \geq 1 - N^{-c}$, every processor receives $O(N/P)$ tree nodes when $P = O(N/\log^{\frac{1}{1-\epsilon}} N)$. For larger values of P, the same bound is achievable, but the algorithm becomes more complicated and requires global information. ◻

5.3. Two-dimensional grids. Any embedding of the N-node complete binary tree in the P-node two-dimensional grid with at most $O(N/P)$ tree nodes per grid node requires dilation $\Omega(\sqrt{P}/\log N)$ [7]. This follows from the fact that in any such embedding, some pair of tree nodes must be mapped distance $\Omega(\sqrt{P})$ apart, while the distance in the tree between the two nodes is $O(\log N)$. This lower bound on dilation is tight for off-line embeddings. In the previous section we saw that every on-line deterministic algorithm that balances the load requires dilation $\Omega(\sqrt{P})$ in the worst case. In this section we present a randomized algorithm which achieves the $O(\sqrt{P}/\log N)$ bound.

We shall see in section 6 that the expected value of the congestion is $\Omega(\frac{N}{\sqrt{P}\log N})$. The randomized algorithm presented here will meet this bound.

5.3.1. Randomized algorithm for two-dimensional torus. For convenience, we work with the two-dimensional torus instead of the grid. Since the two-dimensional torus can be embedded efficiently within the grid, our bounds for the torus are tight to within a constant factor for the grid.

Our randomized algorithm for the torus extends the previous algorithm for rings. We partition the P-node torus into $\log^2 N$ square blocks, each of size $\frac{\sqrt{P}}{\log N}$ by $\frac{\sqrt{P}}{\log N}$. The block in row i and column j is denoted by B_{ij}. We use the $(\log N)$-way balancing transformation twice, independently for each dimension, to obtain two trees, T_r and T_c. The random stretch counts in the two trees are chosen independently. Every node that is in level set i of T_r and level set j of T_c is mapped to B_{ij}. Within a block the tree nodes are distributed evenly in a deterministic manner.

LEMMA 5.4. *With probability $1 - N^{-c}$, $c > 0$, the above algorithm dynamically embeds every N-node tree T in the $\sqrt{P} \times \sqrt{P}$ grid so that each block column receives $O(\frac{N}{\log N})$ tree nodes.*

Proof. The proof of this lemma results directly from Lemma 5.1. □

After knowing that each column receives $O(N/\log N)$ nodes, we show that within a column, the distribution to a particular block is the sum of many mutually independent random variables. Then by using Lemma 5.2 we can show that the algorithm distributes nodes evenly among the blocks in a column.

LEMMA 5.5. *Given that each column receives $O(N/\log N)$ tree nodes with probability $1 - N^{-c}$, $c > 0$, the above algorithm dynamically embeds every N-node tree T in the $\sqrt{P} \times \sqrt{P}$ grid so that each block receives $O(\frac{N}{\log^2 N})$ tree nodes.*

Proof. The analysis and terminology follows the analysis and terminology in [4] quite closely.

We will examine how tree nodes are distributed among the blocks in one column. Let column i be the block column with the largest number of tree nodes, let I be the set of nodes from T that are in block column i, and call any node in I an i-node. Note that all the i-nodes are in the level set i of T_c.

Recall that $n = \log N$. Tree T is divided into three zones. Zone 0 contains nodes that are in level sets 0 through $n/3-1$, zone 1 contains level sets $n/3$ through $2n/3-1$, and zone 3 contains the rest. Define the *triple* for the ith level set as the level sets i, $i + n/3$, and $i + 2n/3$.

Zone 1 of T is naturally partitioned into a set of forests $f_1, \dots, f_m$. Each forest consists of all the trees in zone 1 that have the same nearest common ancestors at level set 0 (that is the top of zone 0). Let these ancestors be $r_1, \dots, r_m$, respectively. For each forest f_j, $1 \le j \le m$, let M_j be the number of i-nodes in f_j and $x_{j,q}$ be the number of i-nodes that are mapped into the triple of level set q.

We want to show that I is evenly distributed among all the level sets of T_r. Since the number of nodes in any level set is bounded by the number of nodes in its triple, it suffices to show that with high probability any triple receives at most $O(\frac{N}{\log^2 N})$ tree nodes from I.

We show that the distribution from I to the triple of an arbitrary level set q is the sum of many mutually independent random variables. The tree nodes assigned to level set triple q have contributions from each forest. We claim that the size of these contributions, $x_{j,q}$, are mutually independent over the forests. The value of $x_{j,q}$ is defined entirely by the level of the roots of the f_j and by the stretch counts chosen there, which are independent by definition. The level of the roots of f_j depends on the level of r_j and on the stretch count chosen there. Since the stretch counts are independent, the level of the roots and thus the values of $x_{j,q}$ for any q are independent.

Now we can apply Lemma 5.2 by noting that each $x_{j,q}$ is less than $2^{\frac{2}{3}n}$ and that $E(x_{j,q}) = \frac{3M_j}{\log N}$. Let $X_q = \sum_{j=1}^{m} x_{j,q}$. Applying the lemma yields

$$\begin{aligned} Pr\left(X_q \geq \beta \frac{N}{\log^2 N}\right) &\leq \exp\left(-\frac{\beta N}{2^{\frac{2}{3}n}\log^2 N}\right) \\ &\leq \exp\left(-\frac{\beta N^{\frac{1}{3}}}{\log^2 N}\right) \\ &\leq N^{-c'}. \end{aligned}$$

A similar argument is valid for zones 0 and 2, so with probability greater than $1-3N^{-c'}$ the number of i-nodes coming from all zones to level set triple q is $O(\frac{N}{\log^2 N})$. Therefore with probability greater than $1-3nN^{-c'}$ every block in column i receives $O(\frac{N}{\log^2 N})$ nodes. And with probability greater than $1-3n^2N^{-c'} > 1-N^{-c}$ every block in the tori receives $O(\frac{N}{\log^2 N})$ nodes. □

THEOREM 5.6. *With probability* $1-N^{-c}$, $c>0$, *the above algorithm dynamically embeds every* N*-node tree* T *in the* $\sqrt{P}\times\sqrt{P}$ *grid so that each grid node receives* $O(N/P)$ *tree nodes and such that the dilation is* $O(\frac{\sqrt{P}}{\log N})$ *and the congestion is* $O(\frac{N}{\sqrt{P}\log N})$.

Proof. By Lemma 5.5 each block will receive $O(\frac{N}{\log^2 N})$ tree nodes with high probability. Since the choices of processor among one block are completely even, each processor receives $O(N/P)$ tree nodes with high probability.

The image of a tree edge can traverse at most three blocks in each dimension; therefore the dilation is bounded by $6\frac{\sqrt{P}}{\log N}$. The tree edges are mapped to the shortest path in the network with at most one turn. Denote the processor in row i and column j by $P_{i,j}$. Without loss of generality, consider a horizontal communication link ℓ between $P_{i,j}$ and $P_{i,j+1}$. From the dilation bound, at least one endpoint of every tree edge whose image traverses ℓ must lie within the interval $P_{i,j-3\sqrt{P}/\log N}$ through $P_{i,j+3\sqrt{P}/\log N}$. Since each grid node has $O(N/P)$ tree nodes and each tree node has at most degree 3, the total number of tree edges that can possibly go through ℓ is bounded by $O(\frac{N}{\sqrt{P}\log N})$. □

The technique for two-dimensional grids can be easily extended to grids with multiple dimensions. In particular, for P-node grids with a fixed number k of dimensions, the bounds on dilation and congestion are $O(P^{1/k}/\log N)$ and $O(\frac{N}{P^{1-1/k}\log N})$, respectively.

THEOREM 5.7. *With probability* $1-N^{-c}$, $c>0$, *the generalization of the above algorithm (which divides the grid into* $\log N$ *blocks along each dimension) dynamically embeds every* N*-node tree* T *in the* $P^{1/k}\times\cdots\times P^{1/k}$ k*-dimensional grid (constant* k*) so that each grid node receives* $O(N/P)$ *tree nodes and such that the dilation is* $O(P^{1/k}/\log N)$ *and the congestion is* $O(\frac{N}{P^{1-1/k}\log N})$.

6. Randomized lower bound. We again consider the two-processor model for our randomized lower bound. Let $\mathcal{A}$ be any probabilistic on-line algorithm which is α-balanced, $1 \leq \alpha < 2$. For an N-node tree the algorithm guarantees that there are at most $\alpha\lceil \frac{N}{2} \rceil$ nodes in either processor. In what follows we show how to construct an N-node binary tree $\mathcal{T}_{\mathcal{A}}$ such that the expected number of cut edges created by $\mathcal{A}$ for $\mathcal{T}_{\mathcal{A}}$ is at least $(1-\alpha/2)^2 N/18 \log N$. As a consequence, we can conclude that any balanced on-line tree embedding algorithm can be expected to make $\Omega(N/\log N)$ cuts on the worst-case tree.

THEOREM 6.1. *For every algorithm $\mathcal{A}$ and $\alpha > 1$, there exists a growth sequence ρ of length N such that if the on-line embedding of ρ is terminally α-balanced between two processors, then the expected number of cut edges is $\Omega(N/\log N)$.*

Proof. The worst-case tree consists of a sequence of complete binary trees that are grown as follows. We start with the depth-1 complete binary tree with three nodes. Let β_i be the random variable denoting the fraction of edges in the ith level that are cut edges and $r = \frac{1}{3}(1-\alpha/2)$. If $E(\beta_1) \leq r/\log N$, then we grow the tree one more level to form the depth-2 complete binary tree with seven nodes. On the other hand, if $E(\beta_1) > r/\log N$, then we start a new complete binary tree at the rightmost leaf of the current complete binary tree. See Figure 6.1 for an example.

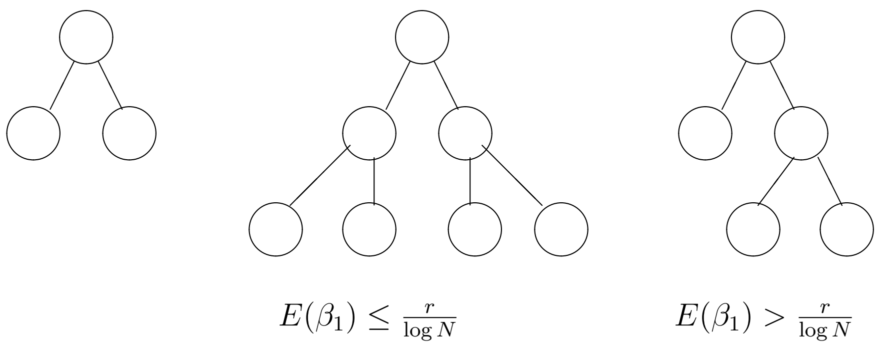

$E(\beta_1) \leq \frac{r}{\log N}$
$E(\beta_1) > \frac{r}{\log N}$

FIG. 6.1. *Tree growth examples.*

In general, we continue growing $\mathcal{T}_{\mathcal{A}}$ using the same rule. If $E(\beta_i) \leq r/\log N$, then we grow $\mathcal{T}_{\mathcal{A}}$ by attaching two leaves to every level-i node. If $E(\beta_i) > r/\log N$, then we grow $\mathcal{T}_{\mathcal{A}}$ by attaching two leaves to just the rightmost level-i node. In other words, we extend the current binary subtree by one level in the former case, and we start a new binary subtree in the latter case. The procedure stops when we have grown N nodes. Figure 6.2 illustrates one possible choice for a 32-node tree. Note that every level (except possibly the last) contains 2^a nodes, where $a \geq 0$.

For the purposes of our discussion, it is useful to consider $\mathcal{T}_{\mathcal{A}}$ as a collection of complete binary subtrees $\{T_i\}$. In particular, we define T_i to be the ith maximal complete binary subtree formed during the construction of $\mathcal{T}_{\mathcal{A}}$. We denote the last subtree formed as T_m. The last level of T_m may be partially empty. We also define n_i to be the number of nodes in T_i less the rightmost leaf for $i < m$ since this node forms the root of the next tree.

Let C be the random variable denoting the number of cut edges created by $\mathcal{A}$ on $\mathcal{T}_{\mathcal{A}}$. In what follows we show that $E(C) \geq (1-\alpha/2)^2 N/18 \log N$ when $N = 2^n$. The proof is divided into two cases, depending on the value of $\sum_{i=1}^{m-1} n_i$.

Case 1. $\sum_{i=1}^{m-1} n_i \geq rN$, where $r = \frac{1}{3}(1-\alpha/2)$.

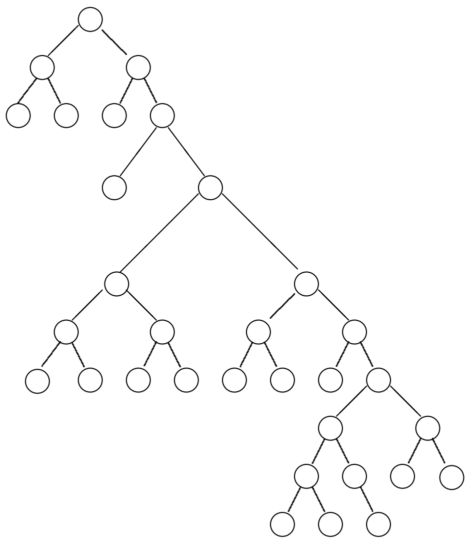


FIG. 6.2. *A 32-node example.*

Let x_i be the random variable denoting the number of cut edges in the last level of T_i, $1 \le i \le m-1$. The last level of T_i has $\frac{1}{2}n_i$ edges and expected fraction of cut edges greater than $r/\log N$, therefore $E(x_i) \ge \frac{1}{2}n_i r/\log N$. Hence,

$$\begin{aligned} E(C) &\ge \sum_{i=1}^{m-1} E(x_i) \\ &> \sum_{i=1}^{m-1} \frac{1}{2} n_i r/\log N \\ &= \frac{1}{2} rN(r/\log N) \\ &= (1-\alpha/2)^2 N/18\log N. \end{aligned}$$

Case 2. $\sum_{i=1}^{m-1} n_i < rN$.

Since $\sum_{i=1}^{m-1} n_i < (1-\alpha/2)N/3$, we know that $n_m > N - (1-\alpha/2)N/3 = (4+\alpha)N/6$ and the number of levels in T_m is $\log N - 1$. Let σ_j be the random variable denoting the fraction of cut edges at level j of T_m for $1 \le j \le \log N - 2$. By assumption, $E(\sigma_j) \le r/\log N$. Also, let s_j be the random variable denoting the number of nodes on level j that remains connected to the root of T_m if all the cut edges are cut. Then $s_0 = 1$ and $s_j \ge 2s_{j-1} - \sigma_j 2^j$ for $1 \le j \le \log N - 2$. Solving the recurrence for s_j, we find that $s_j \ge 2^j(1 - \sum_{i=1}^{j} \sigma_i)$ for $1 \le j \le \log N - 2$.

Since $r < 1/6$, T_m is a complete binary tree with at most $1/6N$ nodes not in the last level. Therefore $s_{\log N - 1} \ge 2s_{\log N-2} - Y - rN$, where Y is the random variable denoting the number of cut edges on the last level of T_m. Let R be the random variable denoting the number of nodes in T_m that are still connected to the root of T_m when all the cut edges are cut.

$$\begin{aligned}
R &\geq \sum_{j=0}^{\log N-1} s_j \\
&\geq \left(\sum_{j=0}^{\log N-2} 2^j \left(1 - \sum_{i=1}^{\log N-2} \sigma_i \right) \right) + s_{\log N-1} \\
&\geq \left(\sum_{j=0}^{\log N-2} 2^j \left(1 - \sum_{i=1}^{\log N-2} \sigma_i \right) \right) + 2s_{\log N-2} - Y - rN \\
&\geq \left(\sum_{j=0}^{\log N-1} 2^j \left(1 - \sum_{i=1}^{\log N-2} \sigma_i \right) \right) - Y - rN \\
&= (N-1) \left(1 - \sum_{i=1}^{\log N-2} \sigma_i \right) - Y - rN.
\end{aligned}$$

Since the largest component of $\mathcal{T}_\mathcal{A}$ can have size at most $\alpha N/2$ once the cut edges have been removed, we know that $R \leq \alpha N/2$. Thus we have that

$$Y \geq (N-1) \left(1 - \sum_{j=1}^{\log N-2} \sigma_j \right) - rN - \alpha N/2.$$

By definition, $C \geq Y$, and therefore

$$\begin{aligned}
E(C) &\geq E(Y) \\
&\geq (N-1)(1-(\log N - 2)r/\log N) - rN - \alpha N/2 \\
&= (1 - r + 2r/\log N)(N-1) - rN - \alpha N/2 \\
&\geq (1 - r - r - \alpha/2)N - 1 \\
&= 1/3(1-\alpha/2)N - 1.
\end{aligned}$$

This concludes the proof that $\mathcal{A}$ is expected to create at least $\frac{(1-\alpha/2)^2 N}{18 \log N}$ cut edges when embedding $\mathcal{T}_\mathcal{A}$ on-line among two processors. □

The preceding result can be generalized to show that for any on-line randomized algorithm for partitioning an N-node binary tree into components of size at most $O(N/P)$, the expected number of cuts is at least $\Omega(N/\log \frac{N}{P})$ in the worst case.

THEOREM 6.2. *For every algorithm $\mathcal{A}$ and $\alpha > 1$, there exists a growth sequence ρ of length N such that if the on-line embedding of ρ is terminally α-balanced among P processors, then the expected number of cut edges is $\Omega(N/\log \frac{N}{P})$.*

Proof. The proof is nearly identical to that above, except that we use a threshold of $\Theta(1/\log \frac{N}{P})$ instead of $\Theta(1/\log N)$ when deciding whether to start a new complete binary tree at each level of $\mathcal{T}_\mathcal{A}$. From Theorem 5.3 this bound is tight, up to constant factors, for all P, $2 \leq P < 2N$. □

7. Conclusion. The execution of divide-and-conquer type algorithms on multicomputers requires a simple strategy for distributing the subprocesses as they are created. Ideally, the distribution would give a balanced load and not require large communication overhead. We have shown that on grid and butterfly networks the worst-case dilation of any deterministic balanced algorithm is as large as the diameter of these networks.

Allowing randomization yields some improvement. Both the dilation and the congestion can be improved by a factor of $\log N$. As shown in [4] this reduces the dilation for the butterfly to a constant. We have also shown that the dilation and expected worst-case congestion for meshes cannot be improved by more than a logarithmic factor.

These large overheads for grids leads to the question of whether there are algorithms with better performance for the kinds of trees that arise in practice.

Acknowledgment. The authors thank Jin-Yi Cai for helpful discussions.

REFERENCES

[1] S. N. Bhatt and J. Y. Cai, *Taking random walks to grow trees in hypercubes*, J. ACM, 40 (1993), pp. 741–764.

[2] S. N. Bhatt and C. E. Leiserson, *How to assemble tree machines*, Adv. Comput. Res., 2 (1984), pp. 95–114.

[3] R. M. Karp and Y. Zhang, *A randomized parallel branch-and-bound procedure*, in 20th ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 290–300.

[4] F. T. Leighton, M. J. Newman, A. G. Ranade, and E. J. Schwabe, *Dynamic tree embeddings in butterflies and hypercubes*, SIAM J. Comput., 21 (1992), pp. 639–654.

[5] A. Ranade, *A Simpler Analysis of the Karp–Zhang Parallel Branch-and-Bound Method*, Technical Report UCB/CSD 90/586, University of California, Los Angeles, CA, 1990.

[6] A. Ranade, *Optimal speedup for backtrack search on a butterfly network*, in 3rd ACM Symposium on Parallel Algorithms and Architecture, Hilton Head, SC, 1991, pp. 40–48.

[7] A. L. Rosenberg and L. Snyder, *Bounds on the costs of data encodings*, Math. Systems Theory, 12 (1978), pp. 9–39.

[8] C. L. Seitz, *Multicomputers*, in Developments in Concurrency and Communication, C. A. R. Hoare, ed., Addison-Wesley, Reading, MA, 1990, pp. 131–201.

[9] I. C. Wu and H. T. Kung, *Communication complexity for parallel divide and conquer*, in IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1991, pp. 151–162.

[10] Y. Zhang, *Parallel Algorithms for Combinatorial Search Problems*, Ph.D. thesis, University of California, Berkeley, CA. 1989.

SIAM J. Comput.
Vol. 29, No. 2, pp. 492–514

A RANDOMIZED FULLY POLYNOMIAL TIME APPROXIMATION SCHEME FOR THE ALL-TERMINAL NETWORK RELIABILITY PROBLEM*

DAVID R. KARGER†

Abstract. The classic all-terminal network reliability problem posits a graph, each of whose edges fails independently with some given probability. The goal is to determine the probability that the network becomes disconnected due to edge failures. This problem has obvious applications in the design of communication networks. Since the problem is $\sharp\mathcal{P}$-complete and thus believed hard to solve exactly, a great deal of research has been devoted to *estimating* the failure probability. In this paper, we give a *fully polynomial randomized approximation scheme* that, given any n-vertex graph with specified failure probabilities, computes in time polynomial in n and $1/\epsilon$ an estimate for the failure probability that is accurate to within a relative error of $1 \pm \epsilon$ with high probability. We also give a deterministic polynomial approximation scheme for the case of small failure probabilities. Some extensions to evaluating probabilities of k-connectivity, strong connectivity in directed Eulerian graphs and r-way disconnection, and to evaluating the Tutte polynomial are also described.

Key words. network reliability, approximation scheme, minimum cut

AMS subject classifications. 05C40, 05C80, 05C85, 68Q25, 68R10, 90B25, 68M15

PII. S0097539796298340

1. Introduction.

1.1. The problem.

We consider a classic problem in reliability theory: given a network on n vertices, each of whose m links is assumed to fail (disappear) independently with some probability, determine the probability that the surviving network is connected. The practical applications of this question to communication networks are obvious, and the problem has therefore been the subject of a great deal of study. A comprehensive survey can be found in [4].

Formally, a network is modeled as a graph G, each of whose edges e is presumed to fail (disappear) with some probability p_e and thus to survive with probability $q_e = 1 - p_e$. Network reliability problems are concerned with determining the probabilities of certain connectivity-related events in this network. The most basic question of *all-terminal network reliability* is determining the probability that the network stays connected. Others include determining the probability that two particular nodes stay connected (two-terminal reliability), and so on.

Most such problems, including the two just mentioned, are $\sharp\mathcal{P}$-complete [25, 24]. That is, they are universal for a complexity class at least as intractable as $\mathcal{NP}$ and therefore seem unlikely to have polynomial time solutions. Attention therefore turned to approximation algorithms. Provan and Ball [24] proved that it is $\sharp\mathcal{P}$-complete even to *approximate* the reliability of a network to within a relative error of ϵ. However, they posited that the approximation parameter ϵ is part of the input, and used an exponentially small ϵ (which can be represented in $O(n)$ input bits) to prove their claim. They note at the end of their article that "a seemingly more difficult unsolved

*Received by the editors January 25, 1996; accepted for publication (in revised form) August 1, 1997; published electronically October 5, 1999.
http://www.siam.org/journals/sicomp/29-2/29834.html

†MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Room NE43-321, Cambridge, MA 02138 (karger@lcs.mit.edu). Parts of this work were done at AT&T Bell Laboratories.

problem involves the case where ϵ is constant, i.e., is not allowed to vary as part of the input list."

Their idea is formalized in the definition of a *polynomial approximation scheme (PAS)*. In this definition, the performance measure is the running time of the approximation algorithm as a function of the problem size n and the error parameter ϵ, and the goal is for a running time that is polynomial in n for each fixed ϵ (e.g., $2^{1/\epsilon}n$). If the running time is also polynomial in $1/\epsilon$, the algorithm is called a *fully polynomial approximation scheme (FPAS)*. An alternative interpretation of an FPAS is that it has a running time polynomial in the input size when ϵ is constrained to be input in unary rather than binary notation. When randomization is used in an approximation scheme, we refer to a *polynomial randomized approximation scheme (PRAS)* or *fully polynomial randomized approximation scheme (FPRAS)*. Such algorithms are required to provide an ϵ-approximation with probability at least 3/4; this probability of success can be increased significantly (e.g., to $1-1/n$ or even $1-1/2^n$) by repeating the algorithm a small number of times [23].

Deterministic FPASs for nontrivial problems seem to be quite rare. However, FPRASs have been given for several $\sharp\mathcal{P}$-complete problems such as counting maximum matchings in dense graphs [7], measuring the volume of a convex polytope [6], and *disjunctive normal form (DNF) counting*—estimating the probability that a given DNF formula evaluates to true if the variables are made true or false at random [18]. In a plenary talk, Kannan [8] raised the problem of network reliability as one of the foremost remaining open problems needing an approximation scheme.

1.2. Our results. In this paper, we provide an FPRAS for the all-terminal network reliability problem. Given a failure probability p for the edges, our algorithm, in time polynomial in n and $1/\epsilon$, returns a number P that estimates the probability FAIL(p) that the graph becomes disconnected. With high probability,[1] P is in the range $(1\pm\epsilon)$FAIL(p). The algorithm is Monte Carlo, meaning that the approximation is correct with high probability but that it is not possible to verify its correctness. It generalizes to the case where the edge failure probabilities are different, to computing the probability the graph is not k-connected (for any fixed k), and to the more general problem of approximating the Tutte polynomial for a large family of graphs. It can also estimate the probability that a Eulerian *directed* graph remains strongly connected under edge failures. Our algorithm is easy to implement and appears likely to have satisfactory time bounds in practice [3, 16].

Some care must be taken with the notion of approximation because approximations are measured by *relative* error. We therefore get different results depending on whether we discuss the failure probability FAIL(p) or the reliability (probability of remaining connected) REL(p) = 1 − FAIL(p). Consider a graph with a very low failure probability, say ϵ. In such a graph, approximating REL(p) by 1 gives a $(1+\epsilon)$-approximation to the reliability, but approximating the failure probability by 0 gives a very poor (infinite) approximation ratio for FAIL(p). Thus, the failure probability is the harder quantity to approximate well. On the other hand, in a very unreliable graph, FAIL(p) becomes easy to approximate (by 1) while REL(p) becomes the challenging quantity. Our algorithm is an FPRAS for FAIL(p). This means that in extremely unreliable graphs, it cannot approximate REL(p). However, it does solve the harder approximation problem on reliable graphs, which are clearly the ones likely to be encountered in practice.

[1]The phrase *with high probability* means that the probability that it does not happen can be made $O(n^{-d})$ for any desired constant d by suitable choice of other constants (typically hidden in the asymptotic notation).

The basic approach of our FPRAS is to consider two cases. When FAIL(p) is large, it can be estimated via direct Monte Carlo simulation of random edge failures. We thus focus on the case of small FAIL(p). Note that a graph becomes disconnected when all edges in some cut fail (a *cut* is a partition of the vertices into two groups; its edges are the ones with one endpoint in each group). The more edges cross a cut, the less likely it is that they will all fail simultaneously. We show that for small FAIL(p), only the smallest graph cuts have any significant chance of failing. We show that there is only a polynomial number of such cuts, and that they can be enumerated in polynomial time. We then use a *DNF counting* algorithm [17] to estimate the probability that one of these explicitly enumerated cuts fails, and take this estimate as an estimate of the overall graph failure probability.

After presenting our basic FPRAS for FAIL(p) in section 2, we present several extensions of it, all relying on our observation regarding the number of small cuts a graph can have. In section 3, we give FPRASs for the network failure probability when every edge has a different failure probability, for the probability that an Eulerian directed graph fails to be strongly connected under random edge failures, and for the probability that two particular "weakly connected" vertices are disconnected by random edge failures. In section 4, we give an FPRAS for the probability that a graph partitions into more than r pieces for any fixed r. In section 5, we give two deterministic algorithms for all-terminal reliability: a simple heuristic that provably gives good approximations on certain inputs and a deterministic PAS that applies to a somewhat broader class of problems. In section 6, we show that our techniques give an FPRAS for the Tutte polynomial on almost all graphs.

1.3. Related work. Previous work gave algorithms for estimating FAIL(p) in certain special cases. Karp and Luby [18] showed how to estimate FAIL(p) in n-vertex planar graphs when the expected number of edge failures is $O(\log n)$. Alon, Frieze, and Welsh [1] showed how to estimate it when the input graph is sufficiently dense (with minimum degree $\Omega(n)$). Other special case solutions are discussed in Colbourn's survey [4]. Lomonosov [21] independently derived some of the results presented here.

A crucial step in our algorithm is the enumeration of minimum and near-minimum cuts. Dinitz et al. [5] showed how to enumerate (and represent) all minimum cuts. Vazirani and Yannakakis [26] showed how to enumerate near-minimum cuts. Karger and Stein [15] gave faster cut enumeration algorithms as well as bounds on the number of cuts that we will use heavily.

A preliminary version of this work appeared in [10]. The author's thesis [9] discusses reliability estimation in the context of a general approach to random sampling in optimization problems involving cuts. In particular, this reliability work relies on some new theorems bounding the number of small cuts in graphs; these theorems have led to other results on applications of random sampling to graph optimization problems [12, 11, 2].

2. The basic FPRAS. In this section, we present an FPRAS for FAIL(p). We use two methods, depending on the value of FAIL(p).

When FAIL(p) is large, we estimate it in polynomial time by direct Monte Carlo simulation of edge failures. That is, we randomly fail edges and check whether the graph remains connected. Since FAIL(p) is large, a small number of simulations (roughly 1/FAIL(p)) gives enough data to estimate it well.

When FAIL(p) is small, we resort to *cut enumeration* to estimate it. Observe that a graph becomes disconnected precisely when all of the edges in some cut of the graph fail. By a *cut* we mean a partition of the graph vertices into two groups. The *cut edges* are those with one endpoint in each group (we also refer to these edges as the ones *crossing* the cut). The *value* of the cut is the number of edges crossing the cut.

We show that when $\mathrm{FAIL}(p)$ is small, only cuts of small value in G have any significant chance of failing. We observe that there is only a polynomial number of such cuts and that they can be found in polynomial time. We therefore estimate $\mathrm{FAIL}(p)$ by enumerating the polynomial-size set of small cuts of G and then estimating the probability that one of them fails.

If each edge fails with probability p, then the probability that a k-edge cut fails is p^k. Thus, the smaller a cut, the more likely it is to fail. It is therefore natural to focus attention on the small graph cuts. Throughout this paper, we assume that our graph has *minimum cut* value c—that is, that the smallest cut in the graph has exactly c edges. Such a graph has a probability of at least p^c of becoming disconnected—namely, if the minimum cut fails.

FACT 2.1. *If each edge of a graph with minimum cut c fails independently with probability p, then the probability that the graph becomes disconnected is at least p^c.*

Clearly, the probability that a cut fails decreases exponentially with the number of edges in the cut. This would suggest that a graph is most likely to fail at its small cuts. We formalize this intuition.

DEFINITION 2.2. *An* α-minimum cut *is a cut with value at most α times the minimum cut value.*

Below, we show how to choose between the two approaches just discussed. If $\mathrm{FAIL}(p) \geq p^c \geq n^{-4}$ then, as we show in subsection 2.1, we can estimate it via Monte Carlo simulation. This works because $\tilde{O}(1/\mathrm{FAIL}(p)) = \tilde{O}(n^4)$ experiments give us enough data to deduce a good estimate ($\tilde{O}(f)$ denotes $O(f \log n)$). On the other hand, when $p^c < n^{-4}$, we know that a given α-minimum cut fails with probability $p^{\alpha c} = n^{-4\alpha}$. We show in subsection 2.2 that there are at most $n^{2\alpha}$ α-minimum cuts. It follows that the probability that *any* α-minimum cut fails is less than $n^{-2\alpha}$—that is, exponentially decreasing with α. Thus, for a relatively small α, the probability that a greater than α-minimum cut fails is negligible. Thus (as we show in subsection 2.3) we can approximate $\mathrm{FAIL}(p)$ by approximating the probability that some less than α-minimum cut fails. Our FPRAS (in subsection 2.4) is based on enumerating these small cuts and determining the probability that one of them fails.

2.1. Monte Carlo simulation.

The most obvious way to estimate $\mathrm{FAIL}(p)$ is through Monte Carlo simulations. Given the failure probability p for each edge, we can "simulate" edge failures by flipping an appropriately biased random coin for each edge. We can then test whether the resulting network is connected. If we do this many times, then the fraction of trials in which the network becomes disconnected should intuitively provide a good estimate of $\mathrm{FAIL}(p)$. Karp and Luby [18] investigated this idea formally, and observed (a generalization of) the following.

THEOREM 2.3. *Performing $O((\log n)/(\epsilon^2 \mathrm{FAIL}(p)))$ trials will give an estimate for $\mathrm{FAIL}(p)$ accurate to within $1 \pm \epsilon$ with high probability.*

COROLLARY 2.4. *If $\mathrm{FAIL}(p) \geq p^c \geq n^{-4}$, then $\mathrm{FAIL}(p)$ can be estimated to within $(1+\epsilon)$ in $\tilde{O}(mn^4/\epsilon^2)$ time using Monte Carlo simulation.*

The criterion that $\mathrm{FAIL}(p)$ not be too small can of course be replaced by a condition that implies it. For example, Alon, Frieze, and Welsh [1] showed that for any *constant* p, there is an FPRAS for network reliability in *dense* graphs (those with minimum degree $\Omega(n)$). The reason is that as n grows and p remains constant, $\mathrm{FAIL}(p)$ is bounded below by a constant on dense graphs and can therefore be estimated in $\tilde{O}(n^2/\epsilon^2)$ time by direct Monte Carlo simulation.

The flaw of the simulation approach is that it is too slow for small values of FAIL(p), namely those less than 1 over a polynomial in n. It is upon this situation that we focus our attention for the remainder of this section. In this case, a huge number of standard simulations would have to be run before we encountered a sufficiently large number of failures to estimate FAIL(p) (note that we expect to run $1/\mathrm{FAIL}(p)$ trials before seeing *any* failures). Karp and Luby [18] tackled this situation for various problems, and showed that it could be handled in some cases by biasing the simulation such that occurrences of the event being estimated became more likely. One of their results was an FPRAS for network reliability in *planar* graphs, under the assumption that the failure probability p of edges is $O((\log n)/n)$ so that the expected number of edges failing is $O(\log n)$. Their algorithm is more intricate than straightforward simulation, and, like ours, relies on identifying a small collection of "important cuts" on which to concentrate.

Another problem where direct Monte Carlo simulation breaks down, and to which Karp and Luby [18], found a solution, is that of *DNF counting*: given a boolean formula in DNF, and given for each variable a probability that it is set to true, estimate the probability that the entire formula evaluates to true. Like estimating FAIL(p), this problem is hard when the probability being estimated is very small. Karp and Luby [18] developed an FPRAS for DNF counting using a biased Monte Carlo simulation. The running time was later improved by Karp, Luby, and Madras [17] to yield the following.

Theorem 2.5. *There is an FPRAS for the DNF counting problem that runs in* $\tilde{O}(s/\epsilon^2)$ *time on a size* s *formula.*

We will use the DNF counting algorithm as a subroutine in our FPRAS.

2.2. Counting near-minimum cuts. We now turn to the case of p^c small. We show that in this case, only the smallest graph cuts have any significant chance of failure. While it is obvious that cuts with fewer edges are more likely to fail, one might think that there are so many large cuts that overall they are more likely to fail than the small cuts. However, the following proposition lets us bound the number of large cuts and show this is not the case.

Theorem 2.6. *An undirected graph has less than* $n^{2\alpha}$ α*-minimum cuts.*

Remark. Vazirani and Yannakakis [26] gave an incomparable bound on the number of small cuts by *rank* rather than by value. $\square$

In this subsection, we sketch a proof of Theorem 2.6. A detailed proof of the theorem can be found in [15] and an alternative proof in [11]. Here, we sketch enough detail to allow for some of the extensions we will need later. We prove the theorem only for unweighted multigraphs (graphs with parallel edges between the same endpoints); the theorem follows for weighted graphs if we replace any weight w edge by a set of w unweighted parallel edges.

2.2.1. Contraction. The proof of the theorem is based on the idea of *edge contraction*. Given a graph $G = (V, W)$ and an edge (v, w), we define a contracted graph $G/(v, w)$ with vertex set $V' = V \cup \{u\} - \{v, w\}$ for some new vertex u and edge set

$$E' = E - \{(v, w)\} \cup \{(u, x) \mid (v, x) \in E \text{ or } (w, x) \in E\}.$$

In other words, in the contracted graph, vertices v and w are replaced by a single vertex u, and all edges originally incident on v or w are replaced by edges incident on u. We also remove self-loops formed by edges parallel to the contracted edge since they cross no cut in the contracted graph.

FACT 2.7. *There is a one-to-one correspondence between cuts in G/e and cuts in G that e does not cross. Corresponding cuts have the same value.*

Proof. Consider a partition (A, B) of the vertices of $G/(v, w)$. The vertex u corresponding to contracted edge (v, w) is on one side or the other. Replacing u by v and w gives a partition of the vertices of G. The same edges cross the corresponding partitions. ◻

2.2.2. The contraction algorithm. We now use repeated edge contraction in an algorithm that selects a cut from G. Consider the following *contraction algorithm.* While G has more than 2 vertices, choose an edge e uniformly at random and set $G \leftarrow G/e$. When the algorithm terminates, we are left with a two-vertex graph that has a unique cut. A transitive application of Fact 2.7 shows that this cut corresponds to a unique cut in our original graph; we will say this cut is *chosen* by the contraction algorithm. We show that any particular minimum cut is chosen with probability at least n^{-2}. Since the choices of different cuts are disjoint events whose probabilities add up to one, it will follow that there are at most n^2 minimum cuts. We then generalize this argument to α-minimum cuts.

LEMMA 2.8. *The contraction algorithm chooses any particular minimum cut with probability at least n^{-2}.*

Proof. Each time we contract an edge, we reduce the number of vertices in the graph by one. Consider the stage in which the graph has r vertices. Suppose G has minimum cut c. It must have minimum degree c, and thus at least $rc/2$ edges. Our particular minimum cut has c edges. Thus a randomly chosen edge is in the minimum cut with probability at most $c/(rc/2) = 2/r$. The probability that we never contract a minimum cut edge through all $n - 2$ contractions is thus at least

$$\begin{aligned}\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right) &= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right)\\ &= \frac{(n-2)(n-3)\cdots(3)(2)(1)}{n(n-1)(n-2)\cdots\cdots(4)(3)}\\ &= \frac{2}{n(n-1)}\\ &= \binom{n}{2}^{-1}\\ &> n^{-2}. \qquad ◻\end{aligned}$$

2.2.3. Proof of Theorem 2.6. We can extend the approach above to prove Theorem 2.6. We slightly modify the contraction algorithm and lower bound the probability so that it chooses a particular α-minimum cut. With r vertices remaining, the probability we choose an edge from our particular α-minimum cut is at most $2\alpha/r$. Let $k = \lceil 2\alpha \rceil$. Suppose we perform random contractions until we have a k-vertex graph. In this graph, choose a vertex partition uniformly at random, so that each of its cuts is chosen with probability 2^{1-k}. It follows that a particular α-minimum cut is chosen with probability

$$\begin{aligned}\left(1-\frac{2\alpha}{n}\right)\left(1-\frac{2\alpha}{(n-1)}\right)\cdots\left(1-\frac{2\alpha}{k+1}\right)2^{1-k} &= \frac{(n-2\alpha)!}{(k-2\alpha)!}\frac{k!}{n!}2^{1-k}\\ &= \frac{\binom{k}{2\alpha}}{\binom{n}{2\alpha}}2^{1-k}\\ &> n^{-2\alpha}.\end{aligned}$$

Note that for α not a half-integer, we are making use of *generalized binomial coefficients* which may have nonintegral arguments. These are discussed in [19, Sections 1.2.5–6]; cf. Exercise 1.2.6.45. There, the Gamma function is introduced to extend factorials to real numbers such that $\alpha! = \alpha(\alpha-1)!$ for all real $\alpha > 0$. Many standard binomial identities extend to generalized binomial coefficients, including the facts that $\binom{n}{2\alpha} < n^{2\alpha}/(2\alpha)!$ and $2^{2\alpha-1} \leq (2\alpha)!$ for $\alpha \geq 1$.

Remark. The contraction algorithm described above is used only to count cuts. An efficient *implementation* given in [15] can be used to *find* all α-minimum cuts in $\tilde{O}(n^{2\alpha})$ time. We use this algorithm in our FPRAS. □

2.3. Cut failure bounds. Using the cut counting theorem just given, we show that large cuts do not contribute significantly to a graph's failure probability. Consider Theorem 2.6; taking $\alpha = 1$, it follows from the union bound that the probability that some minimum cut fails is at most $n^2 p^c$. We now show that the probability that *any* cut fails is only a little bit larger.

THEOREM 2.9. *Suppose a graph has minimum cut c and that each edge of the graph fails independently with probability p, where $p^c = n^{-(2+\delta)}$ for some $\delta > 0$. Then*

1. *the probability that the given graph disconnects is at most $n^{-\delta}(1+2/\delta)$, and*
2. *the probability that a cut of value αc or greater fails in the graph is at most $n^{-\alpha\delta}(1+2/\delta)$.*

Remark. We conjecture that a probability bound of $n^{-\alpha\delta}$ can be proven (eliminating the $(1+2/\delta)$ term). □

Proof. We prove part 1 and then note the small change needed to prove part 2. For the graph to become disconnected, all the edges in some cut must fail. We therefore bound the failure probability by summing the probabilities that each cut fails. Let r be the number of cuts in the graph, and let $c_1, \ldots, c_r$ be the values of the r cuts in increasing order so that $c = c_1 \leq c_2 \leq \cdots \leq c_r$. Let $p_k = p^{c_k}$ be the probability that all edges in the kth cut fail. Then the probability that the graph disconnects is at most $\sum p_k$, which we proceed to bound from above.

We proceed in two steps. First, consider the first n^2 cuts in the ordering (they might not be minimum cuts). Each of them has $c_k \geq c$ and thus has $p_k \leq n^{-(2+\delta)}$, so that

$$\sum_{k \leq n^2} p_k \leq (n^2)(n^{-(2+\delta)}) = n^{-\delta}.$$

Next, consider the remaining larger cuts. According to Theorem 2.6, there are less than $n^{2\alpha}$ cuts of value at most αc. Since we have numbered the cuts in increasing order, this means that $c_{n^{2\alpha}} > \alpha c$. In other words, writing $k = n^{2\alpha}$,

$$c_k > \frac{\ln k}{2 \ln n} \cdot c$$

and thus

$$\begin{aligned} p_k &< (p^c)^{\frac{\ln k}{2 \ln n}} \\ &= \left(n^{-(2+\delta)}\right)^{\frac{\ln k}{2 \ln n}} \\ &= k^{-(1+\delta/2)}. \end{aligned}$$

It follows that

$$\begin{aligned}\sum_{k>n^2} p_k &< \sum_{k>n^2} k^{-(1+\delta/2)} \\ &\leq \int_{n^2}^{\infty} k^{-(1+\delta/2)}\,dk \\ &\leq 2n^{-\delta}/\delta.\end{aligned}$$

Summing the bounds for the first n^2 and for the remaining cuts gives a total of $n^{-\delta} + 2n^{-\delta}/\delta$, as claimed.

The proof of part 2 is the same, except that we sum only over those cuts of value at least αc. □

Remark. A slightly stronger version of part 1 was first proved by Lomonosov and Polesskii [22] using different techniques that identified the cycle as the most unreliable graph for a given c and n. We sketch their result, which we need for a different purpose, in subsection 4.3.2. However, part 2 is necessary for the FPRAS and was not previously known. □

2.4. An approximation algorithm. Our proof that only small cuts matter leads immediately to an FPRAS. First we outline our solution. Given that $\mathrm{FAIL}(p) < n^{-4}$, Theorem 2.9 shows that the probability that a cut of value much larger than c fails is negligible, so we need only determine the probability that a cut of value near c fails. We do this as follows. First, we enumerate the (polynomial size) set of near-minimum cuts that matter. From this set we generate a polynomial size boolean expression (with a variable for each edge, true if the edge has failed) that is true if any of our near-minimum cuts has failed. We then need to determine the probability that this boolean expression is true; this can be done using the DNF counting techniques of Karp, Luby, and Madras [18, 17]. Details are given in the following theorem.

THEOREM 2.10. *When* $\mathrm{FAIL}(p) < n^{-4}$*, there is a (Monte Carlo) FPRAS for estimating* $\mathrm{FAIL}(p)$ *running in* $\tilde{O}(mn^4/\epsilon^3)$ *time.*

Proof. Under the assumption, the probability that a particular minimum cut fails is $p^c \leq \mathrm{FAIL}(p) \leq n^{-4}$. We show there is a constant α for which the probability that any cut of value greater than αc fails is at most $\epsilon\mathrm{FAIL}(p)$. This proves that to approximate to the desired accuracy we need only determine the probability that some cut of value less than αc fails. It remains to determine α. Write $p^c = n^{-(2+\delta)}$; by hypothesis $\delta \geq 2$. Thus by Theorem 2.9, the probability that a cut larger than αc fails is at most $2n^{-\delta\alpha}$. On the other hand, we know that $n^{-(2+\delta)} = p^c \leq \mathrm{FAIL}(p)$, so it suffices to find an α for which $2n^{-\delta\alpha} \leq \epsilon n^{-(2+\delta)}$. Solving this shows that $\alpha = 1 + 2/\delta - (\ln(\epsilon/2))/\delta \ln n \leq 2 - \ln(\epsilon/2)/2\ln n$ suffices and that we therefore need only examine the smallest $n^{2\alpha} = O(n^4/\epsilon)$ cuts.

We can enumerate these cuts in $O(n^{2\alpha}\log^3 n)$ time using certain randomized algorithms [14, 11] (a somewhat slower deterministic algorithm can be found in [26]). Suppose we assign a boolean variable x_e to each edge e; x_e is true if edge e fails and false otherwise. Therefore, the x_e are independent and true with probability p. Let E_i be the set of edges in the ith small cut. Since the ith cut fails if and only if all edges in it fail, the event of the ith small cut failing can be written as $F_i = \wedge_{e\in E_i} x_e$. Then the event of some small cut failing can be written as $F = \vee_i F_i$. We wish to know the probability that F is true. Note that F is a formula in DNF. The size of the formula is equal to the number of clauses ($n^{2\alpha}$) times the number of variables per clause (at most αc), namely, $O(cn^{2\alpha})$. The FPRAS of Karp, Luby, and Madras [17] estimates the truth probability of this formula, and thus the failure probability of the small cuts, to within $(1 \pm \epsilon)$ in $\tilde{O}(cn^{2\alpha}/\epsilon^2) = \tilde{O}(cn^4/\epsilon^3) = \tilde{O}(mn^4/\epsilon^3)$ time.

We are therefore able to estimate to within $(1 \pm \epsilon)$ the value of a probability (the probability that some α-minimum cut fails) that is within $(1 \pm \epsilon)$ of the probability of the event we really care about (the probability that some cut fails). This gives us an overall estimate accurate to within $(1 \pm \epsilon)^2 \approx (1 \pm 2\epsilon)$. ∎

COROLLARY 2.11. *There is an FPRAS for* FAIL(p) *running in* $\tilde{O}(mn^4/\epsilon^3)$ *time.*

Proof. Suppose we wish to estimate the failure probability to within a $(1 \pm \epsilon)$ ratio. If FAIL$(p) > n^{-4}$, then we can estimate it in $\tilde{O}(mn^4/\epsilon^2)$ time by direct Monte Carlo simulation as in Corollary 2.4. Otherwise, we can run the $\tilde{O}(mn^4/\epsilon^3)$-time algorithm of Theorem 2.10. ∎

If the graph is sparse (with $O(n)$ edges) and the minimum cut is $\tilde{O}(1)$ (both these conditions apply to, e.g., planar graphs) then the time for a Monte Carlo trial is $O(n)$, while the size of the formula for the DNF counting step above is $\tilde{O}(n^{2\alpha})$. Thus if we use a different FAIL(p) threshold for deciding which algorithm to use, we can improve the running time bound to $\tilde{O}(n^{3.8}/\epsilon^2)$.

While this time bound is still rather poor, experiments have suggested that performance in practice is significantly better—typically $\tilde{O}(n^3)$ on sparse graphs [16].

3. Extensions.

We now discuss several extensions of our basic FPRAS. In this section, we will consider many cases in which it is sufficient to consider the probability that an α-minimum cut fails for some $\alpha = O(1 - \log\epsilon/\log n)$ (as in the previous section) that is understood in context but not worth deriving explicitly. We will refer to these α-minimum cuts as the *weak* cuts of the graph.

3.1. Varying failure probabilities.

The analysis and algorithm given above extend to the case where each edge e has its own failure probability p_e. To extend the analysis, we transform a graph with varying edge failure probabilities into one with identical failure probabilities. Given the graph G with specified edge failure probabilities, we build a new graph H all of whose edges have the same failure probability p, but that has the same failure probability as G. Choose a small parameter θ. Replace an edge e of failure probability p_e by a "bundle" of k_e parallel edges, each with the same endpoints as e but with failure probability $1 - \theta$, where

$$k_e = \lceil -(\ln p_e)/\theta \rceil .$$

This bundle of edges keeps its endpoints connected unless all the edges in the bundle fail; this happens with probability

$$(1-\theta)^{\lceil -(\ln p_e)/\theta \rceil}.$$

As $\theta \to 0$, this failure probability converges to p_e. Therefore, the reliability of H converges as $\theta \to 0$ to the reliability of G. Thus, to determine the failure probability of G, we need determine only the failure probability of H in the limit as $\theta \to 0$.

Since H has all edge failure probabilities the same, our section 2 analysis of network reliability applies to H. In particular, we know that it suffices to enumerate the weak cuts of H and then determine the probability that one of them fails. To implement this idea, note that changing the parameter θ scales the values of cuts in H without changing their relative values (modulo a negligible rounding error). We therefore build a weighted graph F by taking graph G and giving a weight $\ln 1/p_e$ to edge e. The weak cuts in F correspond to the weak cuts in H. We find these weak cuts in F using the contraction algorithm (which works for weighted graphs [15]) as before.

Given the weak cuts in H, we need to determine the limiting probability that one of them fails as $\theta \to 0$. We have already argued that as $\theta \to 0$, the probability a cut in H fails converges to the probability that the corresponding cut in G fails. Thus we actually want to determine the probability that one of a given set of cuts in G fails. We do this as before. We build a boolean formula with variables for the edges of G and with a clause for each weak cut that is true if all the edges of the cut fail. The only change is that variable x_e is set to true with probability p_e. The algorithm of [17] works with these varying truth probabilities and computes the desired quantity.

THEOREM 3.1. *There is an FPRAS for the all-terminal network reliability problem with varying edge failure probabilities.*

One might be concerned by the use of logarithms to compute edge weights. However, it is easy to see that in fact approximate logarithms suffice for the purpose of enumerating small cuts. If we approximate each logarithm to within relative error .1, then every α-minimum cut in F remains an $11\alpha/9$-minimum cut in the approximation to F. Thus we can enumerate a slightly larger set of near-minimum cuts in order to find the weak cuts. Once we find the weak cuts, we use the original p_e values in the DNF counting algorithm.

In the case of varying failure probabilities, we cannot bound the number of edges in any particular weak cut by a quantity less than m (a weak cut may have $m - n$ edges with large failure probabilities). Thus the size of the DNF formula, and thus the running time of the DNF counting algorithm, may be as large as $mn^{2\alpha} \approx mn^4/\epsilon$.

All the other extensions described in this paper can also be modified to handle varying failure probabilities. But for simplicity, we focus on the uniform case.

3.2. Multiterminal reliability. The multiterminal reliability problem is a generalization of the all-terminal reliability problem. Instead of asking whether the graph becomes disconnected, we consider a subset K of the vertices and ask if some pair of them becomes disconnected. If some pair of vertices in K is separated by a cut of value $O(c)$, then we can use the same theorem on the exponential decay of cut failure probabilities to prove that we need only to examine the small cuts in the graph to determine whether some pair of vertices in K becomes disconnected.

LEMMA 3.2. *If some pair of vertices in K is separated by a cut of value $O(c)$, then there is an FPRAS for the multiterminal reliability problem with source vertices K.*

Proof. We focus on the case of uniform failure probability p; the generalization to arbitrary failure probabilities is as before. Suppose a cut of value βc separates vertices in K. Then the probability that K gets disconnected when edges fail with probability p is at least $p^{\beta c}$. If $p^c > n^{-4}$, then $p^{\beta c} > n^{-4\beta} = n^{-O(1)}$ and we use Monte Carlo simulation as before to estimate the failure probability. If $p^c < n^{-4}$, then by Theorem 2.9, the probability that a cut of value exceeding αc fails is $O(n^{-2\alpha})$. Thus, choosing α such that $n^{-2\alpha} \leq \epsilon p^{\beta c}$, we can enumerate the weak cuts and apply DNF counting. $\square$

3.3. *k*-connectivity. Just as we estimated the probability that the graph fails to be connected, we can estimate the probability that it fails to be k-edge connected for any constant k. Note that the graph fails to be k-edge connected only if some cut has less than k of its edges survive. The probability of this event decays exponentially with the value of the cut, allowing us to prove (as with Theorem 2.9) that if the probability that fewer than k edges in a minimum cut survive is $O(n^{-(2+\delta)})$, then the probability that fewer than k edges survive in a nonweak cut is negligible. Thus, if direct Monte Carlo simulation is not applicable, we need only determine the probability that some weak cut keeps less than k of its edges. But this is another DNF counting problem. For any particular weak cut containing $C \leq m$ edges, we enumerate all $\binom{C}{C-k+1} =$

$O(C^{k-1}) = O(m^{k-1})$ sets of $C-k+1$ edges, and for each add a DNF clause that is true if all the given edges fail.

In fact, one can also adapt the algorithm of [17] to determine the probability that all but $k-1$ variables in some clause of a DNF formula become true; thus we can continue to work with the $O(mn^4/\epsilon)$-size formula we used before.

Corollary 3.3. *For any constant k, there is an FPRAS for the probability that a graph with edge failure probabilities fails to be k-edge connected.*

3.4. Eulerian directed graphs. A natural generalization of the all-terminal reliability problem to directed graphs is to ask for the probability that a directed graph with random edge failures remains strongly connected. A directed graph fails to be strongly connected precisely when all the edges in some *directed* cut fail. In general, the techniques of this paper cannot be applied to directed graphs—the main reason being that a directed graph can have exponentially many minimum directed cuts.

We can, however, handle one special case. In a *Eulerian* directed graph G on vertex set V, the number of edges crossing from any vertex set A to $V-A$ is equal to the number of edges crossing from $V-A$ to A. Thus if we construct an undirected graph H by removing the directions from the edges of G, we know that any (directed) cut in G has value equal to half that of the corresponding (undirected) cut in H. It follows that the α-minimum directed cuts of G correspond to α-minimum undirected cuts of H. Therefore, there are at most $2n^{2\alpha}$ α-minimum directed cuts in G that can be enumerated by enumerating the α-minimum cuts of H (the factor of 2 arises from considering both directions for each cut). As in the undirected case, if the directed failure probability is less than n^{-4}, an analogue of Theorem 2.9 immediately follows, showing that only weak directed cuts are likely to fail. It therefore suffices to enumerate a polynomial number of weak directed cuts to estimate the directed failure probability.

Corollary 3.4. *There is an FPRAS for the probability that a directed Eulerian graph fails to remain strongly connected under random edge failures.*

Corollary 3.5. *For any constant k there is an FPRAS for the probability that a directed Eulerian graph fails to have directed connectivity k under random edge failures.*

3.5. Random orientations. In a similar fashion, we can estimate the probability that, if we orient each edge of the graph randomly, the graph fails to be strongly connected. For each cut, we make a DNF formula with two clauses, one of which is true if all edges point "left" and the other if all edges point "right." (This observation is due to Alan Frieze.) This problem can also be phrased as estimating the number of non-strongly connected orientations of an undirected graph; in this form, it is related to the Tutte polynomial discussed in section 6. Similarly, we can estimate the probability that random orientations fail to produce a k-connected directed graph.

4. Partition into r components.

The quantity FAIL(p) is an estimate of the probability that the graph partitions into more than one connected component. We can similarly estimate the probability that the graph partitions into r or more components for any constant r. Besides its intrinsic interest, the analysis of this problem will be important in our study of some heuristics and derandomizations in section 5 and the Tutte polynomial in section 6.

We first note that a graph partitions into r or more components only if an r-*way cut*—the set of edges with endpoints in different components of an r-way vertex partition—loses all its edges. Note that some of the vertex sets of the partition might induce disconnected subgraphs, so that the r-way partition might induce more than r connected components. However, it certainly does not induce less. Our approach to r-way reliability is the same as for the 2-way case. We show that there are few small r-way cuts and that estimating the probability one fails suffices to approximate the r-way failure probability. As a corollary, we show that the probability of r-way partition is much less than that of 2-way partition.

4.1. Counting multiway cuts. We enumerate multiway cuts using the contraction algorithm as for the 2-way case. Details can be found in [15].

LEMMA 4.1. *In an m-edge unweighted graph the minimum r-way cut has value at most $2m(r-1)/n$.*

Proof. A graph's average degree is $2m/n$. Consider an r-way cut with each of the $r-1$ vertices of smallest degree as its own singleton component and all the remaining vertices as the last component. The value of this cut is at most the sum of the singleton vertex degrees, which is at most $r-1$ times the average degree. ∎

COROLLARY 4.2. *There are at most $\binom{n}{2(r-1)}$ minimum r-way cuts.*

Proof. Suppose we fix a particular r-way minimum cut and run the contraction algorithm until we have $2(r-1)$ vertices. By the previous lemma, the probability that we pick an edge of our fixed cut when k vertices remain is at most $2\frac{r-1}{k}$. Thus the probability that our fixed minimum r-way cut is chosen is

$$\prod_{k=2r-1}^{n}\left(1-\frac{2(r-1)}{k}\right),$$

which is analyzed as in the proof of Theorem 2.6, substituting $r-1$ for α. ∎

COROLLARY 4.3. *For arbitrary $\alpha \geq 1$, there are at most $(rn)^{2\alpha(r-1)}$ α-minimum r-way cuts that can be enumerated in $\tilde{O}((rn)^{2\alpha(r-1)})$ time.*

Proof. First run the contraction algorithm until the number of vertices remaining is $\lceil 2\alpha(r-1)\rceil$. At this point, choose a random r-way partition of what remains. There are at most $r^{2\alpha(r-1)}$ such partitions.

The time bound follows from the analysis of the recursive contraction algorithm [15]. ∎

Remark. We conjecture that in fact the correct bound is $O(n^{\alpha r})$ α-minimum r-way cuts. Subsection 4.3.2 shows this is true for $\alpha = 1$. Proving it for general α would slightly improve our exponents in the following sections. ∎

4.2. An approximation algorithm. Our enumeration of multiway cuts allows an analysis and reduction to DNF counting exactly analogous to the one performed for FAIL(p).

COROLLARY 4.4. *Suppose a graph has r-way minimum cut value c_r, and suppose that each edge fails with probability p, where $p^{c_r} = (rn)^{-(2+\delta)(r-1)}$ for some constant $\delta > 0$. Then the probability that an α-minimum r-way cut fails is at most $(rn)^{-\alpha\delta(r-1)}$ $(1+2/\delta)$.*

Proof. The proof is exactly as for Theorem 2.9, substituting $(rn)^{(r-1)}$ (drawn from Corollary 4.3) for n everywhere. ∎

COROLLARY 4.5. *There is an algorithm for ϵ-approximating the probability that a graph partitions into r or more components, running in $\tilde{O}(m(rn)^{4(r-1)}/\epsilon^3)$ time. The algorithm is an FPRAS with running time $\tilde{O}(mn^{4(r-1)}/\epsilon^3)$ for any fixed r.*

Proof. Exactly as for the 2-way cut case, with $(rn)^{(r-1)}$ replacing n everywhere. Let c_r be the r-way minimum cut value and let δ be defined by $p^{c_r} = (rn)^{-(2+\delta)(r-1)}$. If $p^{c_r} > (rn)^{-4(r-1)}$, estimate the partition probability via Monte Carlo simulation. Otherwise, it follows as in the 2-way cut case that for the same constant α as we chose there, the probability that a greater than α-minimum r-way cut fails is less than ϵp^{c_r}. Thus to estimate the partition probability it suffices to enumerate (in $\tilde{O}((rn)^{4(r-1)}/\epsilon)$ time) the set of α-minimum r-way cuts and perform DNF counting. □

One might wish to compute the probability that a graph partitions into *exactly* r components, but it is not clear that this can be done. In particular, computing $\mathrm{REL}(p)$ can be reduced to this problem (for any r) by adding $r-1$ isolated vertices. There is at present no known FPRAS for $\mathrm{REL}(p)$.

4.3. Comparison to 2-way cuts. For sections 5 and 6, we need to show that the probability of partition into r components is much less than that of partition into 2 components. We give two proofs, the first simpler but with a slightly weaker bound. The following subsections can use the weaker analysis at the cost of worse exponents. In this section, the term "cut" refers exclusively to 2-way cuts unless we explicitly modify it.

4.3.1. A simple argument.

LEMMA 4.6. *If $p^c = n^{-(2+\delta)}$, then the probability that an r-way cut fails is at most $n^{-\delta r/4}(1+2/\delta)$.*

Proof. We show that any r-way cut contains the edges of a (2-way) cut of value $rc/4$. Thus, if an r-way cut fails then an $(r/4)$-minimum 2-way cut fails. The probability that this happens has been upper-bounded by Theorem 2.9.

To show the claim, consider an r-way cut. Contract each component of the r-way partition to a single vertex, yielding an r-vertex graph G'. All edges in this graph correspond to edges of the r-way cut. Every cut in G' corresponds to a cut of the same value in the original graph, so it suffices to show that G' has a 2-way cut of value at least $rc/4$. To see this, note that every vertex in G' has degree at least c, so the number of edges in G' is at least $rc/2$. Consider a random cut of G', generated by assigning each vertex randomly to one side or the other. Each edge has a 1/2 chance of being cut by this partition, so the expected value of this cut is at least $rc/4$. It follows that G' has a cut of value at least $rc/4$ that corresponds to a cut of value at least $rc/4$ in the original graph. □

4.3.2. A better argument. We can get a slightly better bound on the probability that a graph partitions into r components via a small variation on an argument made by Lomonosov and Polesskii [22, 20, 4]. The better bound improves some of our exponents. Their proof uses techniques somewhat different from the remainder of the paper and can safely be skipped.

LEMMA 4.7. *Let $\mathrm{FAIL}_r(G,p)$ denote the probability that G partitions into r or more connected components when each edge fails with probability p. Let G have minimum cut c for some even c. Let C_n be a cycle with $c/2$ edges between adjacent vertices. Then for any r, $\mathrm{FAIL}_r(G,p) \le \mathrm{FAIL}_r(C_n,p)$.*

COROLLARY 4.8. *For any graph G with minimum cut c, if edges fail with probability p where $p^c = n^{-(2+\delta)}$, then the probability the failed graph has r or more connected components is less than $n^{-\delta r/2}$.*

Remark. Note that for $r=2$, the above result gives a slightly stronger bound on $\mathrm{FAIL}(p)$ than we are able to get in Theorem 2.9. Unfortunately, this argument does not appear to extend to proving the bound we need on the probability that a greater than α-minimum r-way cut fails. □

Proof of Corollary 4.8. Thanks to Lemma 4.7, it suffices to prove this claim for the case of G a cycle C_n with $(c/2)$-edge "bundles" between adjacent vertices. The number of components into which C_n is partitioned is equal to the number of bundles that fail, so we need only bound the probability that r or more bundles fail. The probability that a single bundle fails is $p^{c/2} = n^{-(1+\delta/2)}$, so the probability that r particular bundles fail is $n^{-r(1+\delta/2)}$. There are $\binom{n}{r} < n^r$ sets of exactly r bundles. It follows that the probability r or more bundles fail is less than $n^r n^{-r(1+\delta/2)} = n^{-r\delta/2}$. $\square$

Proof of Lemma 4.7. Consider the following time-evolving version of the contraction algorithm on a connected graph G. Each edge of G is given an *arrival time* chosen independently from the exponential distribution with mean 1. Each time an edge arrives, we contract its endpoints if they have not already been contracted. This gives rise to a sequence of graphs $G = G_n, G_{n-1}, \ldots, G_1$, where G_r has r vertices. Let $G[t]$ be the graph that exists at time t. Thus initially $G[0] = G_n$ and eventually $G[\infty]$ has one vertex since all edges have arrived. We draw a correspondence between this model and our edge failure model as follows: at time t, the failed edges are those which have not yet arrived. It follows that each vertex in $G[t]$ corresponds to a connected component of G when each edge has failed (to arrive) independently with probability e^{-t}.

We consider the random variable $T_r(G)$ defined as the time at which the edge that contracts G_r to G_{r-1} arrives. We show that $T_r(C_n)$ *stochastically dominates* $T_r(G)$ for every r—that is,

$$\Pr[T_r(G) \geq t] \leq \Pr[T_r(C_n) \geq t].$$

(See Motwani and Raghavan [23] for additional discussion of this definition.) Assuming this is true, we can prove our result as follows:

$$\begin{aligned}\Pr[G[t] \text{ has } r \text{ or fewer components}] &= \Pr[T_r(G) \leq t]\\ &\geq \Pr[T_r(C_n) \leq t]\\ &= \Pr[C_n[t] \text{ has } r \text{ or fewer components}].\end{aligned}$$

To prove stochastic domination, let $t_r(G) = T_{r-1}(G) - T_r(G)$ denote the length of time for which G_r exists before being contracted to G_{r-1}. Clearly, $t_r(G)$ is just the time it takes for an edge to arrive that has endpoints in different connected components of G_r. It follows that $T_r(G) = \sum_{r'=r}^{n} t_{r'}(G)$. Similarly, $T_r(C_n) = \sum_{r'=r}^{n} t_{r'}(C_n)$. Thanks to the memoryless nature of the exponential distribution, the t_r are mutually independent (this will be justified more carefully later). We will show that $t_r(C_n)$ stochastically dominates $t_r(G)$ for every r. The fact that $T_r(C_n)$ stochastically dominates $T_r(G)$ then follows from the fact that when X dominates X' and Y dominates Y' and the variables are independent, $X + Y$ dominates $X' + Y'$.

To analyze t_r, suppose there are m_r edges in G_r (note m_r is a random variable). The arrival time of each edge in G_r measured from $T_r(G)$ is exponentially distributed with mean 1. Therefore, the arrival time of the first such edge, namely $t_r(G)$, is exponentially distributed with mean $1/m_r$. Now note that G_r is c-connected, so it must have $m_r \geq cr/2$. It follows that $t_r(G)$ is exponentially distributed with mean at most $2/cr$, meaning that it is stochastically dominated by any exponentially distributed variable with mean $2/cr$. On the other hand, when C_n has been reduced to r components, it is isomorphic to C_r. By the same analysis as for G, we know $t_r(C_n)$ is exponentially distributed with mean $2/cr$, and thus stochastically dominates $t_r(G)$.

Our glib claim that the t_r are independent needs some additional justification. Technically, we condition on the values $G_n, \ldots, G_1$ of the evolving graph. We show that regardless of what values G_i we condition on, $T_r(C_n)$ stochastically dominates $T_r(G \mid G_n, \ldots, G_1)$. Since the stochastic domination applies regardless of our conditioning event, it follows even if we do not condition.

Once we have conditioned on the value G_r, t_r is just the time it takes for an edge to arrive that contracts G_r to G_{r-1} and is therefore independent of $t_{r'}$ when $r' \neq r$. But we must ask whether t_r still has the right exponential distribution—the complicating factor being that we know the first edge to arrive at G_r must contract G_r to a specific G_{r-1} and not some other graph. To see that this does not matter, let B be the event that first edge to arrive at G_r is one that creates G_{r-1}. Then

$$\begin{aligned}\Pr[t_r \geq t \mid B] &= \Pr[B \mid t_r \geq t] \Pr[t_r \geq t] / \Pr[B] \\ &= \Pr[B] \Pr[t_r \geq t] / Pr[B] \\ &= \Pr[t_r \geq t]\end{aligned}$$

since, of course, the time of arrival of the edge the contracts G_r has no impact on which of the edges of G_r is the first to arrive. □

5. Heuristics and deterministic algorithms. Until now, we have relied on the fact that the most likely way for a graph to fail is for some of its near-minimum cuts to fail. We now strengthen this argument to observe that most likely, *exactly one* of these near-minimum cuts fails. This leads to two additional results. First, we show that the sum of the individual small-cut failure probabilities is a reasonable approximation to the overall failure probability. This justifies a natural heuristic and indicates that in practice one might not want to bother with the DNF counting phase of our algorithm. In a more theoretical vein, we also give a deterministic PAS for FAIL(p) that applies whenever FAIL$(p) < n^{-(2+\delta)}$. We prove the following theorems.

THEOREM 5.1. *When $p^c < n^{-4}$ (and in particular when* FAIL$(p) < n^{-4}$*), the sum of the weak cuts' failure probabilities is a $(1+o(1))$ approximation to* FAIL(p).

THEOREM 5.2. *When $p^c < n^{-(2+\delta)}$ for any constant δ (and in particular when* FAIL$(p) < n^{-(2+\delta)}$*), there is a deterministic PAS for* FAIL(p) *running in $(n/\epsilon)^{\exp(O(-\log_n \epsilon))}$ time.*

We remark that unlike many PASs whose running times are only polynomial for constant ϵ, our PAS has polynomial running time so long as $\epsilon = n^{-O(1)}$. Its behavior when ϵ is tiny prevents it from being an FPAS, however.

To prove these theorems, we argue as follows. As shown in section 2, it is sufficient to approximate, for the given ϵ, the probability that some α-minimum cut fails, where

$$\alpha = 1 + 2/\delta - (\ln \epsilon)/\delta \ln n.$$

Let us write these α-minimum cuts as C_i, $i = 1, \ldots, n^{2\alpha}$. Let F_i denote the event that cut C_i fails. We can use inclusion-exclusion to write the failure probability as

$$\Pr[\cup F_i] = \sum_{i_1} \Pr[F_{i_1}] - \sum_{i_1<i_2} \Pr[F_{i_1} \cap F_{i_2}] + \sum_{i_1<i_2<i_3} \Pr[F_{i_1} \cap F_{i_2} \cap F_{i_3}] + \cdots .$$

Later terms in this summation measure events involving many cut failures. We show that when many cuts fail, the graph partitions into many pieces, meaning a multiway cut fails. We then argue (using Lemma 4.6 or Corollary 4.8) that this is so unlikely that later terms in the sum can be ignored. This immediately yields Theorem 5.1.

To prove Theorem 5.2, we show that for any fixed ϵ it is sufficient to consider a constant number of terms (summations) on the right-hand side in order to get a good approximation. Observe that the kth term in the summation can be computed deterministically in $O(m(n^{2\alpha})^k)$ time by evaluating the probability of each of the $(n^{2k\alpha})$ intersection events in the sum (each can be evaluated deterministically since it is just the probability that all edges in the specified cuts fail). Thus, our running time will be polynomial so long as the number of terms we need to evaluate is constant.

5.1. Inclusion-exclusion analysis. As discussed above, our analyses use a truncation of the inclusion-exclusion expression for

$$\Pr[\cup F_i] = \sum_{i_1} \Pr[F_{i_1}] - \sum_{i_1<i_2} \Pr[F_{i_1} \cap F_{i_2}] + \sum_{i_1<i_2<i_3} \Pr[F_{i_1} \cap F_{i_2} \cap F_{i_3}] + \cdots .$$

Suppose we truncate the inclusion-exclusion, leaving out the kth and later terms. If k is odd the truncated sum yields a lower bound; if k is even it yields an upper bound. We show that this bound is sufficiently tight. We do so by rewriting the inclusion-exclusion expression involving particular sets of failed cuts failing as an expression based on *how many* cuts fail.

LEMMA 5.3. *Let S_u be the event that u* or more *of the events F_i occur. If the inclusion-exclusion expansion is truncated at the kth term, the error introduced is*

$$\sum_u \binom{u-2}{k-2} \Pr[S_u].$$

Proof. Let T_u be the event that *exactly* u of the events F_i occur. Consider the first summation $\sum F_{i_1}$ in the inclusion-exclusion expansion. The event where precisely the events $F_{j_1}, \ldots, F_{j_u}$ occur (that is, the event that cuts $C_{j_1}, \ldots, C_{j_k}$ fail but no others fail) contributes to the u terms $\Pr[F_{j_1}], \ldots, \Pr[F_{j_u}]$ in the sum. It follows that each sample point contributing to T_u is counted $u = \binom{u}{1}$ times in the summation. Thus,

$$\sum \Pr[F_{i_1}] = \sum_u \binom{u}{1} \Pr[T_u].$$

By the same reasoning,

$$\sum \Pr[F_{i_1} \cap F_{i_2}] = \sum_u \binom{u}{2} \Pr[T_u],$$

and so on. It follows that the error introduced by truncation at term k is

$$\begin{aligned} \sum_{i_1<i_2<\cdots<i_k} \Pr[F_{i_1} \cap F_{i_2} \cap \cdots \cap F_{i_k}] - \sum_{i_1<i_2<\cdots<i_{k+1}} \Pr[F_{i_1} \cap F_{i_2} \cap \cdots \cap F_{i_{k+1}}] + \cdots \\ = \sum_{j \ge k} (-1)^{k-j} \sum_u \binom{u}{j} \Pr[T_u] \\ = \sum_u \sum_{j \ge k} (-1)^{k-j} \binom{u}{j} \Pr[T_u] \\ = \sum_u \binom{u-1}{k-1} \Pr[T_u]. \end{aligned}$$

Now recall that S_u is the event that u *or more* of the F_i occur, meaning that $\Pr[T_u] =$

$\Pr[S_u] - \Pr[S_{u+1}]$. Thus we can rewrite our bound above as

$$\begin{aligned}
&\sum_u \binom{u-1}{k-1}(\Pr[S_u] - \Pr[S_{u+1}]) \\
&\quad = \sum_u \binom{u-1}{k-1} \Pr[S_u] - \sum_u \binom{u-1}{k-1} \Pr[S_{u+1}] \\
&\quad = \sum_u \binom{u-1}{k-1} \Pr[S_u] - \sum_u \binom{u-2}{k-1} \Pr[S_u] \\
&\quad = \sum_u \left(\binom{u-1}{k-1} - \binom{u-2}{k-1}\right) \Pr[S_u] \\
&\quad = \sum_u \binom{u-2}{k-2} \Pr[S_u].
\end{aligned}$$

This completes the proof. $\square$

5.2. A simple approximation. Using the above error bound, we can prove Theorem 5.1. Let F_i denote the event that the ith near-minimum cut fails. Our objective is to estimate $\Pr[\cup F_i]$. Summing the individual cuts' failure probabilities corresponds to truncating our inclusion-exclusion sum at the second term, giving (by Lemma 5.3) an error of $\sum_{u \geq 2} S_u$. We now bound this error by bounding the quantities S_u.

LEMMA 5.4. *If u distinct (2-way) cuts fail then a $\lceil \log(u+1)+1 \rceil$-way cut fails.*

Proof. Consider a configuration in which u distinct cuts have failed simultaneously. Suppose this induces k connected components. Let us contract each connected component in the configuration to a single vertex. Each failed cut in the original graph corresponds to a distinct failed cut in the contracted graph. Since the contracted graph has k vertices, we know that there are at most $2^{k-1}-1$ ways to partition its vertices into two nonempty groups, and thus at most this many cuts. In other words, $u \leq 2^{k-1}-1$. Now solve for u and observe it must be integral. $\square$

COROLLARY 5.5. *If $p^c = n^{-(2+\delta)}$ then $\Pr[S_u] \leq n^{-\lceil \log(u+1)+1 \rceil \delta/2}$.*

Proof. Apply Corollary 4.8 to the previous lemma. $\square$

Thus, for example, S_2 and S_3 are upper bounded by the probability that a 3-way cut fails, which by Corollary 4.8 is at most $n^{-3\delta/2}$. More generally, all 2^k values $S_{2^k}, \ldots, S_{2^{k+1}-1}$ are at most $n^{-(k+2)\delta/2}$. It follows that the error in our approximation by the bound of Theorem 5.1 is

$$\begin{aligned}
\sum_{u \geq 2} S_u &\leq \sum_{k \geq 1} 2^k n^{-(k+2)\delta/2} \\
&= n^{-\delta} \sum_{k \geq 1} (2n^{-\delta/2})^k \\
&= 2n^{-3\delta/2}(1+o(1)),
\end{aligned}$$

whenever $\delta > 0$. This quantity is $o(p^c)$, and thus $o(\mathrm{FAIL}(p))$, whenever $n^{-3\delta/2} = o(n^{-(2+\delta)})$, i.e. $\delta > 4$. This proves Theorem 5.1.

5.3. A PAS. We now use the inclusion-exclusion analysis to give a PAS for $\mathrm{FAIL}(p)$ when $p^c = n^{-(2+\delta)}$ for some fixed $\delta > 0$, thus proving Theorem 5.2. We give an ϵ-approximation algorithm with a running time of $(n/\epsilon)^{\exp(O(-\log_n \epsilon))}$, which is clearly polynomial in n for each fixed ϵ (and in fact, for any $\epsilon = n^{-O(1)}$).

We must eliminate two uses of randomization: in the contraction algorithm for identifying the α-minimum cuts and in the DNF counting algorithm for estimating their failure probability.

The first step is to deterministically identify the near-minimum cuts of G. One approach is to use a derandomization of the contraction algorithm [13]. A more efficient approach is to use a cut enumeration scheme of Vazirani and Yannakakis [26]. This scheme enumerates cuts in increasing order of value, with a "delay" of $\tilde{O}(mn)$ per cut. From the fact that there are only $n^{2\alpha}$ weak cuts, it follows that all weak cuts (in the sense of section 3) can be found in $\tilde{O}(mn^{1+2\alpha})$ time.

We must now estimate the probability one of the near-minimum cuts fails. Let us consider truncating to the first k terms in the inclusion-exclusion expansion. From Corollary 5.5 we know that $\Pr[S_u] \leq n^{-(\log(u+1)+1)\delta/2}$. It follows from Lemma 5.3 that for any $k \leq \frac{1}{3}\delta \log n$, our error from using the k-term truncation of inclusion-exclusion is

$$\begin{aligned}
\sum_u \binom{u-2}{k-2} n^{-(\log(u+1)+1)\delta/2} &\leq n^{-\delta/2} \sum_{u \geq k} (u-2)^{k-2}(u+1)^{-\delta(\log n)/2} \\
&\leq \sum_{u \geq k} (u+1)^{k-2-\delta(\log n)/2} \\
&\leq \sum_{u \geq k} (u+1)^{\delta(\log n)/3-2-\delta(\log n)/2} \\
&\leq \sum_{u \geq k} (u+1)^{-\delta(\log n)/6-1} \\
&\leq \int_{u=k-1}^{\infty} (u+1)^{-\delta(\log n)/6-1} \, \boldsymbol{d}u \\
&= \frac{k^{-\delta(\log n)/6}}{\delta(\log n)/6} \\
&= \frac{n^{-\delta(\log k)/6}}{\delta(\log n)/6} \\
&= O(n^{-\delta(\log k)/6}).
\end{aligned}$$

This quantity is $O(\epsilon n^{-(2+\delta)}) = O(\epsilon p^c) = O(\epsilon \mathrm{FAIL}(p))$ for some $k = 2^{O(-\log_n \epsilon)}$. It follows that for an ϵ-approximation we need only evaluate the inclusion-exclusion up to the kth term. Computing the kth term requires examining every set of k of the $(n/\epsilon)^{O(1)}$ α-minimum cuts; this requires $(n/\epsilon)^{\exp(O(-\log_n \epsilon))}$ time. This concludes the proof of Theorem 5.2.

We can slightly improve our bound on $\Pr[S_u]$, which in turn gives better bounds on k.

LEMMA 5.6. *If u distinct α-minimum cuts fail, then a $u^{1/2\alpha}$-way cut fails.*

Proof. Consider a configuration in which u distinct cuts have failed simultaneously. Suppose this induces k connected components. Let us contract each connected component in the configuration to a single vertex. In this contracted graph (before edges fail), the minimum cut is at least c (since contraction never reduces the minimum cut). Furthermore, each of the u failed cuts is a cut of value at most αc, and thus an α-minimum cut, in the contracted graph. Since the contracted graph has k vertices, we know from Theorem 2.6 that $u < k^{2\alpha}$, meaning that $k > u^{1/2\alpha}$. □

However, this serves only to reduce the values of our constants (and reduce the running time from an exponential to a polynomial dependence on $1/\delta$).

6. The Tutte polynomial.

The Tutte polynomial $T(G;x,y)$ is a polynomial in two variables defined by a graph G. Evaluating it at various points x, y on the so-called Tutte plane yields various interesting quantities regarding the graph. In particular, computing the network reliability $\mathrm{REL}(p)$ is the special case of evaluating the Tutte polynomial at the point $x = 1, y = 1/(1-p)$. Another special case is counting the number of strongly connected orientations of an undirected graph, discussed in subsection 3.5. Yet another is counting the number of forests in a graph. Alon, Frieze, and Welsh [1] showed that for any *dense* graph (one with $\Omega(n^2)$ edges) and fixed x and fixed $y \geq 1$ there is an FPRAS for the Tutte polynomial.

6.1. Results.

In this section, we prove the following.

THEOREM 6.1. *For every $y \geq 1$ there is a $c = O(y \log nxy)$ (in particular, $c = O(\log n)$ for any fixed x and y) such that for all n-vertex m-edge graphs of edge-connectivity greater than c,*

$$T(G;x,y) = \frac{y^m}{(y-1)^{n-1}}(1 + O(1/n)).$$

Thus, a good approximation can be given in constant time. Note that almost all graphs fall under this theorem as the minimum cut of a random graph is tightly concentrated around $n/2 \gg c$.

THEOREM 6.2. *For every $y > 1$ there is a $c = O(y \log nxy)$ such that there is an FPRAS for $T(G;x,y)$.*

This theorem is perhaps unsurprising given the previous theorem. A slightly more challenging quantity is the "second-order term" saying how far a given graph diverges from its approximation in the first theorem.

THEOREM 6.3. *Let*

$$\Delta T(G;x,y) = \frac{y^m}{(y-1)^{n-1}} - T(G;x,y).$$

For any fixed $y > 1$ and fixed x, there is a $c = O(\log n)$ such that there is an FPRAS for $\Delta T(G;x,y)$.

This theorem is stronger than and implies the previous theorem. When ΔT is very close to 0, $\frac{y^m}{(y-1)^{n-1}}$ accurately approximates T but approximating ΔT with small relative error is harder.

6.2. Method.

Our proofs begin with a lemma of Alon, Frieze, and Welsh [1] (which we have slightly rephrased to include what is for them the special case of $x = 1$).

LEMMA 6.4 (see [1]). *When $y > 1$,*

$$T(G;x,y) = \frac{y^m}{(y-1)^{n-1}} E[Q^{\kappa-1}],$$

where $Q = (x-1)(y-1)$, and κ is a random variable equal to the number of connected components of G when each edge of G fails independently with probability $p = 1 - 1/y$. (In the case $Q = 0$ (when $x = 1$), we use the fact that $0^r = 0$ for $r \neq 0$ while $0^0 = 1$.)

In other words, when p_r is the probability that the graph with random edge failures partitions into exactly r components, the Tutte polynomial can be evaluated from

$$E[Q^{\kappa-1}] = \sum_{k=1}^{n} p_r Q^{r-1}.$$

For the remainder of this subsection, we normalize our analysis by considering the quantity $T'(G;x,y) = T(G;x,y)\frac{(y-1)^{n-1}}{y^m} = E[Q^{\kappa-1}]$. Clearly, any results on relative approximations to T' translate immediately into results on relative approximations to T.

We begin with an intuitive argument. From Theorem 2.9, when $p^c = n^{-(2+\delta)}$ (which happens for some $c = O(\log n)$ for any fixed p) we know p_r is negligible for $r \geq 1$. Intuitively, since $p_1 \approx 1$ and all other $p_r \approx 0$, we might as well approximate T' by Q. Extending this argument, we know that compared to p_2, all terms p_r for $r > 2$ are negligible. Therefore, the error in the approximation of T' by Q is almost entirely determined by $p_2 Q^2$, which we can determine by computing p_2.

To prove our results formally, we have to deal with the fact that the term Q^r in the expectation increases exponentially with r. We prove that the p_r decay fast enough to damp out the increasing values of Q^r. We also need to be careful that when $Q < 0$, the large leading terms do not cancel each other out.

6.3. Proofs. For our formal analysis, instead of the quantities p_r, it is more convenient to work with quantities s_r measuring the probability that the graph partitions into r *or more* components. Note that $s_1 = 1$ and $s_2 = \text{FAIL}(p)$. Since $p_r = s_r - s_{r+1}$, it follows that

$$\begin{aligned}
T'(G;x,y) &= \sum_{r=1}^{n} p_r Q^{r-1} \\
&= \sum_{r=1}^{n} (s_r - s_{r+1}) Q^{r-1} \\
&= \sum_{r=1}^{n} s_r Q^{r-1} - \sum_{r=2}^{n} s_r Q^{r-2} \\
&= 1 + \sum_{r=2}^{n} s_r (Q^{r-1} - Q^{r-2}) \\
&= 1 + (Q-1) \sum_{r=2}^{n} s_r Q^{r-2}.
\end{aligned}$$

Theorem 6.1 will follow directly from the last equation if we can show that the trailing term $(Q-1)\sum_{r=2}^{n} s_r Q^{r-2} = O(1/n)$. Theorem 6.3 will follow if we can give an FPRAS for $\sum_{r=2}^{n} s_r Q^{r-2}$. The fact that the value of this sum is $o(1)$ (Theorem 6.1) means that the FPRAS for it immediately yields an FPRAS for T', thus proving Theorem 6.2.

To prove these results, first consider the case $x = 1$. In this case $Q = 0$, meaning $Q^{r-2} = 1$ for $r = 2$ and 0 for $r > 2$. Thus $T'(G;x,y) = 1 - s_2 = 1 - \text{FAIL}(p) = \text{REL}(p)$. We have already seen in Theorem 2.9 that whenever $p^c = n^{-(2+\delta)}$, the probability that the graph becomes disconnected is at most $n^{-\delta}(1 + 2/\delta)$. This is certainly $O(1/n)$ if $\delta \geq 1$, meaning $\text{REL}(p) = 1 - O(1/n)$. But this in turn is true when $p^c < n^{-3}$, i.e.,

$$c > 3 \ln n / \ln(y/(y-1)) = O(y \ln n).$$

This proves Theorem 6.1 for $Q = 0$. On the other hand, Theorem 6.3 simply claims that there is an FPRAS for $1 - \text{REL}(p) = \text{FAIL}(p)$, which is what section 2 showed. Finally, Theorem 6.2 says that when $\text{FAIL}(p)$ is small, we can approximate $\text{REL}(p)$ (by approximating $\text{FAIL}(p)$).

We now generalize this argument to the case $x > 1$. To derive the appropriate lower bound on c, we state two criteria that will we need in our analysis. First, we require c to be such that $p^c = n^{-(2+\delta)}$ for some $\delta > 1$. Equivalently, we have $1 \leq \delta = -\log(n^2 p^c)/\log n$. Second, we require that $Q < \frac{1}{4}n^{\delta/4}$. Plugging in for δ, we find the equivalent requirement

$$\begin{aligned} Q &< \frac{1}{4}n^{\delta/4} \\ &= \frac{1}{4}(n^2p^c)^{-1/4}, \\ (4Q)^4 &< 1/n^2p^c, \\ n^2(4Q)^4 &< \left(\frac{y}{y-1}\right)^c, \\ \ln(256Q^4n^2)/\ln\left(1+\frac{1}{y-1}\right) &< c. \end{aligned}$$

This is true for some $c = O(y(\ln nQ)) = O(y \ln nxy)$ as claimed.

Given the above relations between Q, n, and δ, we can use Corollary 4.8. Since $p^c = n^{-(2+\delta)}$, we deduce that $s_r \leq n^{-r\delta/2}$. Since $Q < \frac{1}{4}n^{\delta/4} < \frac{1}{2}n^{\delta/2}$ we find that

$$\begin{aligned} \sum_{r=r_0}^{n} s_r Q^{r-2} &\leq Q^{-2} \sum_{r \geq r_0} (Qn^{-\delta/2})^r \\ &\leq Q^{-2}(Qn^{-\delta/2})^{r_0}/(1-(Qn^{-\delta/2})^{r_0}) \\ &\leq Q^{-2}(Qn^{-\delta/2})^{r_0}/\left(1-\frac{1}{2^{r_0}}\right) \\ &\leq 2Q^{-2}(Qn^{-\delta/2})^{r_0}. \end{aligned}$$

Our results follow from this bound. First, taking $r_0 = 2$, we find that the error in approximating $T'(G; x, y)$ by 1 is at most

$$2n^{-\delta} = o(1).$$

This proves Theorem 6.1.

To prove Theorem 6.3, note that the leading term in the summation (6.1) is $s_2 \geq n^{-(2+\delta)}$. We can therefore estimate the sum to within relative error $O(\epsilon)$ by evaluating summation terms up to summation index r_0 where $(Qn^{-\delta/2})^{r_0} \leq \epsilon n^{-(2+\delta)}$. Since the left-hand side decreases exponentially in n as a function of r_0, we can achieve this error bound by taking

$$r_0 = O(\log_n(n^{2+\delta}/\epsilon)) = O(1 + \log_n 1/\epsilon).$$

In other words, we need only to determine $O(1 - \log_n \epsilon)$ terms in the summation. This in turn reduces to determining the quantities s_r appearing in those terms.

We cannot find the s_r exactly. However, for an ϵ-approximation, it suffices to approximate each relevant s_r to within ϵ. We can do so using the algorithm of Corollary 4.5. The running time of this algorithm for estimating the r-way failure probability to within ϵ is $(n^r/\epsilon)^{O(1)}$. We have argued above that we need only to run the algorithm for $r \leq r_0 = O(1 - \log_n \epsilon)$. It follows that the running time of our algorithm is $n^{O(1-\log_n \epsilon)}/\epsilon^{O(1)} = (n/\epsilon)^{O(1)}$, as required. This proves Theorem 6.3.

Finally, we consider the case $x < 1$. Our argument is essentially unchanged from before. We need to be slightly more careful because our sum is now an alternating sum, which means that the leading terms are a good approximation only if they do not cancel each other out. To see that such cancelling does not occur, note that the first term has value $s_2 = n^{-(2+\delta)}$, while the remaining terms (by the analysis above) have total (absolute) value $O(n(Qn^{-3\delta/2}))$. If we choose n large enough that $Q < \frac{1}{4}n^{\delta/4}$, then this bound is $O(\frac{1}{4}n^{-5\delta/4}) < \frac{1}{4}s_2$ for $\delta > 4$, so the remaining terms do not cancel s_2.

7. Conclusion.

We have given an FPRAS for the all-terminal network reliability problem and several variants. In the case of large failure probability, the FPRAS uses straightforward Monte Carlo simulation. For smaller failure probabilities, the FPRAS uses an efficient reduction to DNF counting or a less efficient deterministic computation. An obvious open question is whether there is also a deterministic PAS for the case of large failure probabilities. Another is whether there is also an FPRAS for $\mathrm{REL}(p) = 1 - \mathrm{FAIL}(p)$, the question being open only for the case $\mathrm{REL}(p)$ near 0.

This work has studied probabilistic *edge* failures; a question of equal importance is that of network reliability under *vertex* failures. We are aware of no results on the structure of minimum vertex cuts that could lead to the same results as we have derived here for edge cuts. In particular, graphs can have exponentially many minimum vertex cuts. The same obstacle arises in *directed* graphs (where we wish to measure the probability of failing to be strongly connected).

Although the polynomial time bounds proven here are not extremely small, we expect much better performance in practice since most graphs will not have the large number of small cuts assumed for the analysis. Preliminary experiments [16] have suggested that this is indeed the case.

REFERENCES

[1] N. Alon, A. Frieze, and D. Welsh, *Polynomial time randomized approximation schemes for the Tutte polynomial of dense graphs*, in Proceedings of the 35th Annual Symposium on the Foundations of Computer Science, S. Goldwasser, ed., IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 24–35.

[2] A. A. Benczúr and D. R. Karger, *Approximate s–t min-cuts in $\tilde{O}(n^2)$ time*, in Proceedings of the 28th ACM Symposium on Theory of Computing, G. Miller, ed., ACM Press, 1996, pp. 47–55.

[3] C. C. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein, *Experimental study of minimum cut algorithms*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, M. Saks, ed., ACM-SIAM, New Orleans, LA, 1997, pp. 324–333.

[4] C. J. Colbourn, *The combinatorics of network reliability*, The International Series of Monographs on Computer Science, Vol. 4, Oxford University Press, New York, 1987.

[5] E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov, *On the structure of a family of minimum weighted cuts in a graph*, in Studies in Discrete Optimization, A. A. Fridman, ed., Nauka Publishers, Moscow, 1976, pp. 290–306.

[6] M. E. Dyer, A. M. Frieze, and R. Kannan, *A random polynomial time algorithm for approximating the volume of convex bodies*, J. ACM, 38 (1991), pp. 1–17.

[7] M. Jerrum and A. Sinclair, *Approximating the permanent*, SIAM J. Comput., 18 (1989), pp. 1149–1178.

[8] R. Kannan, *Markov chains and polynomial time algorithms*, in Proceedings of the 35th Annual Symposium on the Foundations of Computer Science, S. Goldwasser, ed., IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 656–671.

[9] D. R. Karger, *Random Sampling in Graph Optimization Problems*, Ph.D. thesis, Stanford University, Stanford, CA, 1994.

[10] D. R. Karger, *A randomized fully polynomial approximation scheme for the all terminal network reliability problem*, in Proceedings of the 27th ACM Symposium on Theory of Computing, ACM Press, New York, 1995, pp. 11–17.

[11] D. R. Karger, *Minimum cuts in near-linear time*, in Proceedings of the 28th ACM Symposium on Theory of Computing, G. Miller, ed., ACM Press, New York, 1996, pp. 56–63.

[12] D. R. Karger, *Random sampling in cut, flow, and network design problems*, Math. Oper.

Res., 24 (1999), pp. 383–413.

[13] D. R. Karger and R. Motwani, *Derandomization through approximation: An $\mathcal{NC}$ algorithm for minimum cuts*, SIAM J. Comput., 26 (1997), pp. 255–272.

[14] D. R. Karger and C. Stein, *An $\tilde{O}(n^2)$ algorithm for minimum cuts*, in Proceedings of the 25th ACM Symposium on Theory of Computing, A. Aggarwal, ed., ACM Press, New York, 1993, pp. 757–765.

[15] D. R. Karger and C. Stein, *A new approach to the minimum cut problem*, J. ACM, 43 (1996), pp. 601–640.

[16] D. R. Karger and R. P. Tai, *Implementing a fully polynomial time approximation scheme for all terminal network reliability*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, M. Saks, ed., ACM-SIAM, New Orleans, LA, Jan. 1997, pp. 334–343.

[17] R. M. Karp, M. Luby, and N. Madras, *Monte Carlo approximation algorithms for enumeration problems*, J. Algorithms, 10 (1989), pp. 429–448.

[18] R. M. Karp and M. G. Luby, *Monte Carlo algorithms for planar multiterminal network reliability problems*, J. Complexity, 1 (1985), pp. 45–64.

[19] D. E. Knuth, *Fundamental algorithms*, The Art of Computer Programming, 2nd ed., Addison-Wesley Publishing Company, Vol. 1, Reading, MA, 1973.

[20] M. V. Lomonosov, *Bernoulli scheme with closure*, Problems Inform. Transmission, 10 (1974), pp. 73–81.

[21] M. V. Lomonosov, *On Monte Carlo estimates in network reliability*, Probab. Engrg. Inform. Sci., 8 (1994), pp. 245–264.

[22] M. V. Lomonosov and V. P. Polesskii, *Lower bound of network reliability*, Problems Inform. Transmission, 7 (1971), pp. 118–123.

[23] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, 1995.

[24] J. S. Provan and M. O. Ball, *The complexity of counting cuts and of computing the probability that a network remains connected*, SIAM J. Comput., 12 (1983), pp. 777–788.

[25] L. Valiant, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

[26] V. V. Vazirani and M. Yannakakis, *Suboptimal cuts: Their enumeration, weight, and number*, in Automata, Languages and Programming, 19th International Colloquium Proceedings, Lecture Notes in Comput. Sci. 623, Springer-Verlag, New York, Berlin, 1992, pp. 366–377.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 515–544

ON THE MAXIMUM SCATTER TRAVELING SALESPERSON PROBLEM[*]

ESTHER M. ARKIN[†], YI-JEN CHIANG[‡], JOSEPH S. B. MITCHELL[§], STEVEN S. SKIENA[¶], AND TAE-CHEON YANG[‖]

Abstract. We study the problem of computing a Hamiltonian tour (cycle) or path on a set of points in order to maximize the minimum edge length in the tour or path. This "maximum scatter" traveling salesperson problem (TSP) is closely related to the bottleneck TSP and is motivated by applications in manufacturing (e.g., sequencing of rivet operations) and medical imaging. In this paper, we give the first algorithmic study of these problems, including complexity results, approximation algorithms, and exact algorithms for special cases. In an attempt to model more accurately the real problems that arise in practice, we also generalize the basic problem to consider a more general measure of "scatter" in which points on a tour or path should be far not only from their immediate predecessor and successor, but also from other near-neighbors along the tour or path.

Key words. traveling salesperson problem, traveling salesman problem, maximum scatter TSP, bottleneck TSP, Hamiltonian cycle/path, approximation algorithms, matching, optimization, Posa conjecture, Dirac's theorem

AMS subject classifications. 68Q25, 68R10, 68U05

PII. S0097539797320281

1. Introduction.

In this paper we study a new class of traveling salesperson problem (TSP) variants that is based on the objective of finding, in an edge-weighted complete graph $G = (S, E)$, a tour (cycle) or a path that is most "scattered." Specifically, the goal is to *maximize* the length of a shortest edge in the tour/path; i.e., to have each point be far from the points that are visited just before or just after it in the tour/path. We refer to this problem as the *maximum scatter TSP*, or the *max-min 1-neighbor TSP*.

More generally, we consider the *max-min m-neighbor TSP* in which the goal is to maximize the minimum distance between any point and all of its "m-neighbors" along the tour/path. An *m-neighbor* of p is a point that is at most m points away

*Received by the editors April 23, 1997; accepted for publication (in revised form) August 5, 1998; published electronically October 5, 1999. An extended abstract of an earlier version of this paper appears in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, LA, 1997, pp. 211–220.

http://www.siam.org/journals/sicomp/29-2/32028.html

†Department of Applied Mathematics and Statistics, SUNY Stony Brook, Stony Brook, NY 11794-3600 (estie@ams.sunysb.edu). The work of this author was supported in part by NSF grant CCR-9504192.

‡Department of Computer and Information Science, Polytechnic University, 6 MetroTech Center, Brooklyn, NY 11201 (yjc@photon.poly.edu). The research of this author was conducted while with the Department of Applied Mathematics and Statistics, SUNY Stony Brook, and was supported in part by NSF grant DMS-9312098.

§Department of Applied Mathematics and Statistics, SUNY Stony Brook, Stony Brook, NY 11794-3600 (jsbm@ams.sunysb.edu). The work of this author was supported in part by Boeing, Hughes Research Laboratories, NSF grant CCR-9504192, and a Fulbright research fellowship at Tel Aviv University.

¶Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794-4400 (skiena@cs.sunysb.edu). The work of this author was supported in part by NSF grant CCR-9109289 and ONR award 400x116yip01.

‖Department of Computer Science and Statistics, Kyungsung University, 110, DaeYeon-dong, Nam-gu, Pusan, 608-736, Korea (tcyang@csd.kyungsung.ac.kr). The work of this author was supported in part by Kyungsung University research grants in 1996.

from p in the tour/path. The problem, then, is to find a tour/path, $(p_1, \ldots, p_n)$, on the n points of S in order to maximize

$$\min_{i \in \{1,\ldots,n\}} \min_{j \in \{i-m,\ldots,i-1,i+1,\ldots,i+m\}} d(p_i, p_j),$$

where $d(p_i, p_j)$ denotes the length of edge (p_i, p_j) (*not* the length along the tour/path linking p_i and p_j). The indices in the minimization should be modified appropriately to account for wrap-around effects in cycles, or endpoint effects in paths.

Motivation. This *maximum scatter TSP* problem arises in some manufacturing processes where it is important to have substantial separation (distance) between consecutive (or *nearly* consecutive) operations on a workpiece. We first encountered the problem in discussions with Boeing, where the problem was that of sequencing the riveting operations when fastening sheets of metal together [21, 22]. To prevent nonuniform deformation of the sheet metal plates that are being joined, it is important to sequence the riveting process so that the distance between one rivet and the next one is large; i.e., the riveting operations should be *scattered.* In operations that involve heating the workpiece, it may be important not just to have each point well separated from its immediate predecessor and successor, but also from its m-neighbors, in order to allow for cooling time in the vicinity of each operation.

The maximum scatter TSP also arises in some medical imaging applications, as has been recently studied by Penavic [20]. When imaging physiological functions using a dynamic spatial reconstructor (DSR), the radiation sources are placed along the top half of a circular ring, with sensors placed directly opposite, in the bottom half of the ring. The "firing sequence" determines the order in which the sources, and their partnered sensors, are activated, usually in a periodic pattern. The sensors collect intensity data of the energy that passes through the patient, who is placed in the center of the ring. When source i is activated, some amount of scattering occurs, so it is important not to activate sensors of nearby sources (e.g., $i+1, i-1, i+2, \ldots$) soon after i is activated. This motivated Penavic to study firing sequence orderings for some specific geometries of DSR hardware, and motivates our study (in section 6) of some one-dimensional versions of the maximum scatter problem.

Related Work. A closely related problem is the *bottleneck traveling salesperson problem* (BTSP), in which the goal is to *minimize* the length of a *longest* edge in a Hamiltonian cycle. (See [17].) The BTSP is known to be NP-complete, and no constant factor approximation algorithm can exist unless P=NP. Assuming the edge lengths satisfy the triangle inequality, there exists an approximation algorithm to produce a tour whose longest edge has a length that is at most twice the optimal, and this approximation factor of two is best possible [18]. By similar techniques we obtain similar hardness results for the maximum scatter TSP (in edge-weighted graphs); however, our approximation algorithm is very different from that given in [18], and neither algorithm works for the other problem.

Another variant of the TSP that is potentially related to our work is the MAX TSP, in which the goal is to find a tour whose length is as long as possible. Several approximation algorithms exist for the MAX TSP (without any assumptions on the edge lengths), including a naive algorithm that achieves a 2/3 approximation factor, and a recent algorithm of Hassin and Rubinstein [10] that achieves a 5/7 approximation factor. (See also Kosaraju [16], who claimed an earlier approximation factor of 5/7; however, Bhatia has recently shown that their algorithm gives a bound of 19/27 instead of the claimed 5/7.) However, neither of these algorithms works for our maximum scatter problem. Also, there exists a polynomial-time approximation scheme for geometric (e.g., Euclidean) versions of the MAX TSP [4]. The noncrossing version of geometric MAX TSP is considered in [2]. Very recently, since the original appearance of our own paper [3], Barvinok et al. [5] show that the MAX TSP can be solved exactly in polynomial time, for points in fixed dimension, under a (constant-complexity) polyhedral metric. Most recently, Fekete [8] shows that the Euclidean MAX TSP is NP-hard for the Euclidean metric, in three or more dimensions. His proof also shows that the Euclidean maximum scatter TSP is NP-hard in three or more dimensions. However, the complexity of the MAX TSP, as well as the maximum scatter TSP and the noncrossing MAX TSP, remains open for points in the Euclidean plane.

The most directly related previous work on the maximum scatter TSP has been done by Penavic [19], who studied the optimal firing sequences for a DSR application in which all sources are equally spaced. Our one-dimensional results generalize this previous work.

Summary of Results.

- We show that the maximum scatter TSP is, in general, NP-complete, and that no constant-factor approximation algorithm exists, unless P=NP. We do not yet know the complexity of the *geometric* maximum scatter TSP problem for points in the plane (and Euclidean distances); we conjecture that it is NP-hard. See section 2.
- In the case that distances obey the triangle inequality, we give approximation algorithms (section 3) guaranteed to get within factor 2 of optimal for the maximum scatter TSP, for both the *tour* (cycle) and *path* versions of the problem. Further, we prove that this factor is best possible.
  The methods also extend to give a factor-2 (best factor) approximation for the max-min 2-neighbor TSP, for the cycle version and some cases ($n \neq 3k+1$) of the path version of the problem, by using a very recent result on the *Pósa conjecture* (see section 3). (These extensions are not practical and are of theoretical interest only.)
- We also give (section 4) practical constant-factor approximation results on the max-min 2-neighbor TSP in the case that distances obey the triangle inequality, for both the cycle (factor 64) and path (factor 32) versions of the problem. Our approach utilizes novel methods of applying matching techniques, together with metric properties of the distance matrix, and may be of independent interest.
  These methods extend (section 5) also to yield a constant-factor ($4 \cdot 8^{\lceil m/2 \rceil}$) approximation for the path version of the max-min m-neighbor TSP, for any constant $m > 2$, when n is a multiple of $m+1$.
- We give algorithms (section 6) that find the optimal solution to the max-min 1-neighbor TSP exactly, in some special cases. In particular, we obtain linear-time (optimal) exact algorithms for the case of points on a line, or on a circle, with Euclidean distances, for both the cycle and the path versions of the problems (for the path version of points on a circle, we consider only the case in which n is an odd integer or the points are equally spaced).

Notation. We consider a complete graph, $G = (S, E)$ on a set of points S, with the full edge set E. We let $d(p,q)$ denote the distance between point p and point q, and we let $d_e = d(p,q)$ denote the length of edge $e = (p,q)$. We let $diam(R)$ denote the *diameter* of the set $R \subseteq S$; i.e., $diam(R) = \max_{p,q \in R} d(p,q)$. The (open) *disk* $\mathcal{D}_p(r)$ of radius r, centered at $p \in S$, is defined to be the set of points in S that are at distance less than r from p. The boundary of $\mathcal{D}_p(r)$ is the *circle* consisting of points of S that are at distance exactly r from p.

An algorithm is said to yield a *factor α approximation* for a maximization problem if the algorithm is guaranteed to produce a solution whose objective function value is at least $(1/\alpha)$ times the value of an optimal solution.

2. Complexity considerations: Hardness results. Our first result demonstrates the importance of restricting our problem to graphs having some structure (e.g., edge lengths obeying triangle inequality).

THEOREM 2.1. *There can exist no polynomial-time, constant-performance-bound approximation algorithm for an arbitrary instance of the maximum scatter TSP (cycle or path version), unless P=NP.*

Proof. We consider the cycle version of the problem; the same argument applies for the path version as well. Assume to the contrary that such an algorithm exists, and its performance bound is ρ. We show that this algorithm can be used to test whether a graph has a Hamiltonian cycle. Given an arbitrary graph, $G = (S, E)$, construct an instance of maximum scatter TSP (cycle version) on the complete graph on S, with edge lengths $d_e = 1$ if $e \notin E$, and $d_e = 1 + \rho$ if $e \in E$. If G has a Hamiltonian cycle, the maximum scatter TSP has an objective value of $\rho + 1$; thus, the approximation algorithm must yield a tour whose shortest edge length is greater than 1 (i.e., it produces a Hamiltonian cycle in G). If G does not have a Hamiltonian cycle, then the maximum scatter TSP has an objective value of 1, and therefore the approximation algorithm also yields a tour for which at least one edge has length 1. Thus G is Hamiltonian if and only if the alleged approximation algorithm produces a tour of minimum edge length greater than 1. $\square$

THEOREM 2.2. *There can exist no polynomial-time algorithm for the maximum scatter TSP (cycle or path version) in a graph satisfying the triangle inequality with a performance bound smaller than* 2 *unless P=NP.*

Proof. The proof follows the previous one, except that the weight $\rho+1$ is replaced by weight 2. (Note that the triangle inequality holds in the resulting graph.) $\square$

While the above two results parallel the similar facts known about the bottleneck TSP (BTSP), there is a significant difference in our current knowledge about the BTSP versus the maximum scatter TSP in the case of the geometric versions of these two problems. Specifically, it is easy to see that the BTSP is NP-hard even for points in the plane, using a reduction from Hamiltonian cycle in a grid graph (see [17]): A tour whose maximum edge length is 1 exists in the complete graph whose edge lengths are given by the Euclidean distances if and only if the grid graph is Hamiltonian. This approach does not apply, however, to show NP-hardness of the maximum scatter TSP in the plane; to date, this problem remains open. (In one dimension, the problem is solvable in linear time, as we show in section 6, while in Euclidean 3-space Fekete [8] has recently shown that the problem is NP-hard.)

Next, we show that the max-min m-neighbor TSP is at least as hard as the max-min 1-neighbor (scatter) TSP.

THEOREM 2.3. *In a graph satisfying the triangle inequality, the max-min m-neighbor TSP with $m > 1$ is at least as hard as the max-min* 1*-neighbor TSP for both the cycle and the path versions of the problems.*

Proof. We reduce the problem with $m = 1$ to that with $m > 1$. From the given node set S of the former, we construct another node set S' for the latter. Set S' consists of the original n nodes of S, plus an additional $\lceil \frac{n}{2} \rceil (m-1)$ nodes. Let D and d be the largest and smallest distances between all pairs of nodes in S. We arrange

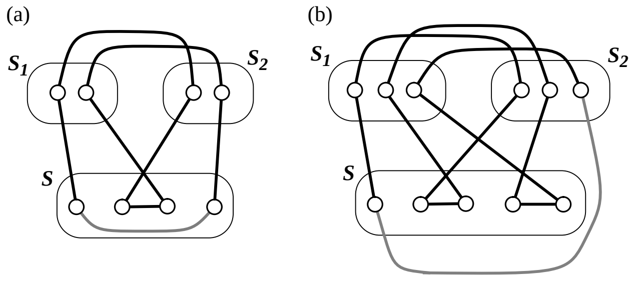


FIG. 2.1. *Examples for the construction in the proof of Theorem* 2.3 *with* $m = 3$: (a) $n = 4$ *is even*; (b) $n = 5$ *is odd. The cycles are given by connecting all* (*solid and dashed*) *edges*, *and the paths are given by removing one edge*, *e.g.*, *the dashed edge*, *from the cycles.*

the $\lceil \frac{n}{2} \rceil (m-1)$ additional nodes of S' into $m-1$ clusters $S_1, \ldots, S_{m-1}$, each with $\lceil \frac{n}{2} \rceil$ nodes. Any pair of nodes in the same cluster are within distance less than d, and any pair of nodes that are in different clusters (treating S as a cluster at this time) are separated by a distance larger than D. There must exist an optimal m-neighbor cycle on S' consisting of $\lceil \frac{n}{2} \rceil$ "rounds," where each round contains exactly two nodes of S and one node from each of $S_1, \ldots, S_{m-1}$ (see Figure 2.1). Observe that for any two nodes in the same cluster (S or S_i), only those in S can possibly be m-neighbors. An optimal 1-neighbor cycle on S can then be obtained from this cycle by skipping the nodes not in S. An optimal m-neighbor path on S' has the same structure except for cutting the cycle open by removing one edge (see Figure 2.1). Again an optimal 1-neighbor path on S can be obtained in the same way. $\square$

The following is immediate from Theorems 2.3 and 2.2.

COROLLARY 2.4. *There can exist no polynomial-time algorithm for the max-min m-neighbor TSP (cycle or path version) in a graph satisfying the triangle inequality with a performance bound smaller than* 2, *unless P=NP.*

3. Approximating the max-min 1-neighbor TSP.

In this section, we consider the maximum scatter (max-min 1-neighbor) TSP, for both the cycle and path versions, on the complete graph $\mathcal{G} = (S, E)$, whose edges are weighted with nonnegative lengths that satisfy the triangle inequality. In the riveting problem, S may be points in the plane, and edges are assigned Euclidean lengths.

We first mention that some of the natural approaches do not work in this case:

(1) The BTSP, in which we must minimize the longest edge (vs. maximizing the shortest edge), has a 2-approximation algorithm [18], which works as follows: First, find a biconnected subgraph of the given graph that minimizes the longest edge; the square of this biconnected graph is Hamiltonian; finally, we trace a Hamiltonian tour in the square of the graph. However, this approach does not yield an approximation for the maximum scatter TSP.

(2) The MAX TSP also has constant factor approximation algorithms (e.g., to get within factor 5/7 of optimal [16]), but again the methods do not yield approximations for the maximum scatter TSP. The reason for this is that the algorithms for MAX TSP generate "long" paths, and then connect them arbitrarily into a cycle, since the added edges will only add to the length of the tour. However, in the scatter problem, adding a very short edge at this stage can drastically reduce the scatter (minimum length edge) obtained, resulting in a tour that is far from optimal.

(3) A natural greedy algorithm (select the next point to be the unvisited one that is furthest away) also fails for the maximum scatter TSP, as there are simple examples showing that it can yield paths or tours that are arbitrarily bad with respect to optimum.

(4) For the geometric version of the maximum scatter TSP in the plane, we have also attempted to use our one-dimensional (exact) results, by projecting the points onto each of a small number of axes, solving the one-dimensional problems on those axes, and then lifting back to two dimensions. However, bad examples exist for this approach as well.

We now present a new approximation method that does yield provably good solutions. We concentrate for now on the cycle version of the problem. We make use of the following theorem due to Dirac [6]: *If every node of a graph having n nodes ($n \geq 3$) has degree $\geq \lceil \frac{n}{2} \rceil$, then the graph has a Hamiltonian cycle.* We devise an efficient algorithmic version of Dirac's proof in our proof of Theorem 3.5 below; we are not aware of an algorithmic proof appearing in the literature.

Our algorithm is very simple: We consider the edges E in order of increasing length. We delete edges, one by one, starting with the shortest one and stopping when the deletion of an edge, say, $e = (u, v)$ (of length $d_e = d(u, v)$), would cause a node (say, v) to have degree *less* than $\lceil \frac{n}{2} \rceil$. Let $\mathcal{G}' = (S, E')$ be the graph that remains. By definition, then, E' includes edge $e = (u, v)$, as well as all other edges of length at least d_e.

By Dirac's theorem, we know that $\mathcal{G}'$ has a Hamiltonian cycle. We can find such a cycle (call it C) in $O(n^2)$ time (see Theorem 3.5). A shortest edge of C has length at least d_e.

Let d^* be the length of a shortest edge of an optimal tour.

LEMMA 3.1. *Let $R \subset S$ be a subset of nodes with $|R| > \lfloor \frac{n}{2} \rfloor$ (resp., $|R| > \lceil \frac{n}{2} \rceil$). Then, in any Hamiltonian cycle (resp., path) on S, there must exist an edge joining two nodes of R.*

Proof. Let $k = \lfloor \frac{n}{2} \rfloor$ and $S_{out} = S \setminus R$. To avoid having any edge joining two nodes of R in a Hamiltonian cycle, all nodes in R must be connected only to nodes in S_{out}, which is impossible since the total number of edges out of R is at least $2k + 2$ but such a number for S_{out} is at most $2k$. We use the same argument to prove the case of a Hamiltonian path. Letting $k = \lceil \frac{n}{2} \rceil$, we have that the total number of edges out of R is at least $2 + 2[(k + 1) - 2] = 2k$ but such a number for S_{out} is at most $2(k - 1)$. ◻

We later also use the following immediate generalization of Lemma 3.1.

LEMMA 3.2. *For any integer $m \geq 1$, let $R \subset S$ be a subset of nodes with $|R| > \lfloor \frac{n}{m+1} \rfloor$ (resp., $|R| > \lceil \frac{n}{m+1} \rceil$). Then, in any Hamiltonian cycle (resp., path) on S, there must exist two nodes of R that are m-neighbors.*

COROLLARY 3.3. *Let $R \subset S$ be a subset of nodes with $|R| = \lfloor \frac{n}{2} \rfloor + 1$. Then, $d^* \leq diam(R)$.*

LEMMA 3.4. *$d^* \leq 2d_e$, where d_e is the length of the edge, $e = (u, v)$, which is the first edge* not *to be deleted in our algorithm, as its deletion would result in the degree of v being $< \lceil \frac{n}{2} \rceil$.*

Proof. Let $R = \{u\} \cup N_v$, where N_v denotes the set of nodes w such that $(v, w) \notin E'$. (Note that $v \in N_v$, since we do not allow self-loops, and that $u \notin N_v$.) Thus, R includes all nodes at distance at most d_e from node v, and node v itself. By specification of the algorithm, $|S \setminus N_v| \geq \lceil \frac{n}{2} \rceil$, while $|S \setminus R| < \lceil \frac{n}{2} \rceil$. Thus, $|R| = \lfloor \frac{n}{2} \rfloor + 1$, so, from Corollary 3.3, $d^* \leq diam(R)$. However, since all nodes of R are at distance

$\leq d_e$ from v, we know, from the triangle inequality, that $diam(R) \leq 2d_e$. Thus, $d^* \leq 2d_e$. ∎

THEOREM 3.5. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an $O(n^2)$-time approximation algorithm for the maximum scatter TSP on S that produces a Hamiltonian tour whose shortest edge is at least $\frac{1}{2}$ times optimal. The approximation factor is the best possible unless P=NP.*

Proof. The approximation factor and its optimality follow from Lemma 3.4 and Theorem 2.2. The $O(n^2)$ running time can be accomplished as follows. First, in overall time $O(n^2)$, we find for each node the median weight (the $\lceil \frac{n}{2} \rceil$th longest) edge among those edges incident on it. By taking the minimum of these n medians, we find the length of the critical edge, e, and therefore can easily identify the edges E' of the graph $\mathcal{G}'$ by deleting all edges of E shorter than e. Our algorithm now resembles a constructive version of Dirac's proof. We start with a Hamiltonian cycle in the complete graph; this is trivially found. Then, we delete edges (u, v), one by one, which are *not* in E'. With each deletion, we modify the existing Hamiltonian cycle, $u = v_1, v_2, \ldots, v_{n-1}, v_n = v$, swapping out edges (u, v) and (v_i, v_{i+1}), and swapping in edges (u, v_{i+1}) and (v_i, v), where $i \in \mathcal{S} \cap \mathcal{T}$ is an index in the intersection of $\mathcal{S} = \{i : (u, v_{i+1}) \in E'\}$ and $\mathcal{T} = \{i : (v_i, v) \in E'\}$. As in Dirac's proof, $\mathcal{S} \cap \mathcal{T}$ must be nonempty, since $|\mathcal{S} \cup \mathcal{T}| < n$ (since $n \notin \mathcal{S}, \mathcal{T}$) and $|\mathcal{S} \cup \mathcal{T}| + |\mathcal{S} \cap \mathcal{T}| = |\mathcal{S}| + |\mathcal{T}| = deg(u) + deg(v) \geq n$. By representing the sets $\mathcal{S}$ and $\mathcal{T}$ with bit vectors of length n each, where bit j indicates if index j belongs to the set, we can find $i \in \mathcal{S} \cap \mathcal{T}$ in $O(n)$ time. Note that the new edges in the resulting Hamiltonian cycle, (u, v_{i+1}) and (v_i, v), are both in E', i.e., we never introduce new edges that need to be deleted. Therefore we need to perform the deletions at most n times, in overall time $O(n^2)$. ∎

Now we consider the path version of the problem. One simple way is to add a *fake point* Q very far away from all points of S so that it does not affect the max-min edge length at all, and then solve the cycle problem. When we get the answer that is a cycle, we simply cut the cycle open at Q to get a path, and then remove Q. Notice that using Lemma 3.1, we have a corollary for the *path version* similar to Corollary 3.3: Any subset R of S with $|R| = \lceil \frac{n}{2} \rceil + 1$ has $d^* \leq diam(R)$, where d^* is now the optimal max-min edge length of Hamiltonian paths. When we apply our cycle algorithm, the problem size is $n' = n + 1$, for which we take the size of R to be $\lfloor \frac{n'}{2} \rfloor + 1 = \lceil \frac{n}{2} \rceil + 1$, which is exactly the desired size of R for the *path* version.

COROLLARY 3.6. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an $O(n^2)$-time approximation algorithm for the maximum scatter TSP on S that produces a Hamiltonian path whose shortest edge is at least $\frac{1}{2}$ times optimal. The approximation factor is the best possible unless P=NP.*

4. Approximating the max-min 2-neighbor TSP.

We turn now to the problem of approximating the max-min 2-neighbor TSP. We begin by pointing out, as observed by Khuller [12], that our method for the max-min 1-neighbor TSP can be directly extended to work for the cycle version of the max-min 2-neighbor TSP, if the following *Pósa conjecture* (which is a generalization of Dirac's theorem) is true: *If every node of an n-node graph has degree $\geq \lceil \frac{2}{3} n \rceil$, then the graph contains the square (i.e., the second power) of a Hamiltonian cycle.* Here the kth *power* of a cycle C is the graph obtained by joining every pair of nodes that are k-neighbors in C. Very recently, the Pósa conjecture has been proved for a very large n [14]. Also, the proof leads to a polynomial-time algorithm [12, 13]. Thus we have the following result.

THEOREM 4.1. *Given a complete graph on a set S of n points, with edge lengths satisfying the triangle inequality, there exists a polynomial-time approximation algorithm that constructs a Hamiltonian cycle on S whose objective value for the max-min 2-neighbor problem is at least $\frac{1}{2}$ times the optimal. The approximation factor is the best possible, unless P=NP.*

Notice that the node-degree condition in the Pósa conjecture does not work well for us in the path version unless $n = 3k$; see Lemma 3.2. For $n = 3k + 2$, we can again add a fake point Q very far away, construct a cycle, and then remove Q to obtain a desirable path. But when $n = 3k + 1$, the same trick requires us to add two fake points Q and Q'. In constructing a cycle, we also need to make sure that Q and Q' are next to each other (so that removing Q and Q' does not break the final path), which might not be easy to do. We do not explore this further here.

COROLLARY 4.2. *Given a complete graph on set S of n points satisfying triangle inequality, where $n = 3k$ or $n = 3k + 2$, there exists a polynomial-time approximation algorithm that constructs a Hamiltonian path on S whose objective value for the max-min 2-neighbor problem is at least $\frac{1}{2}$ times the optimal. The approximation factor is the best possible unless P=NP.*

Remark 1. Theorem 4.1 and Corollary 4.2 are of theoretical interest only. The current proof of the Pósa conjecture is quite complex and uses the *regularity lemma* that holds only for a very large n [14]. To turn the proof into an algorithm, it is necessary to use an algorithmic version of the regularity lemma that runs in polynomial time. Such an algorithm is given in [1]. To get an idea about how large n should be, we quote the following sentences from [1, p. 474]:

> We note that the dependence of $Q(\epsilon, t)$ on ϵ and t, as well as the running time of the algorithm on these two parameters is rather horrible; in fact, $\log^* Q$ is a polynomial (of degree about 20) in $1/\epsilon$ and in t.

As described in [1], the regularity lemma is valid for $n > Q$, where $Q = Q(\epsilon, t)$ is a function of given (fixed) parameters $\epsilon > 0$ and t (a positive integer). In typical applications, ϵ is small (and thus $1/\epsilon$ is large) and t is large. Even if we take $t = 2$ and $\epsilon = 1$, this means that n has to be larger than Q, where $\log^* Q \approx 2^{20}$!

Remark 2. An algorithmic proof of the following *Seymour conjecture* (a generalization of the Pósa conjecture) would also imply a factor-2 (optimal factor) approximation algorithm for the cycle version of the max-min m-neighbor TSP: If every node of an n-node graph has degree $\geq \lceil \frac{m}{m+1} n \rceil$, then the graph contains the mth power of a Hamiltonian cycle. Currently the conjecture is proved for the case when every node has degree $\geq (\frac{m}{m+1} + \epsilon)n$, for any $\epsilon > 0$, by using the regularity lemma [15].

We turn now to describe our own approximation method for the cycle and path versions of the max-min 2-neighbor TSP. Our method is *not* based in any way on the Pósa conjecture. Although the approximation factors are not optimal, the algorithms are simple and practical. Also, unlike Corollary 4.2, the algorithms work for all cases for the path version as well. Furthermore, the techniques can be extended to obtain an approximation for the max-min m-neighbor TSP, for *any* constant $m > 2$ (see section 5), not just $m = 2$.

Our main results are constant-factor approximation algorithms, which utilize novel methods of applying matching techniques, together with metric properties of the distance matrix, to search for "large" triangles (cycles of three nodes) that can then be concatenated to form provably good tours and paths for the 2-neighbor problem. Our overall strategy consists of the following steps:

1. Construct $\lfloor n/3 \rfloor$ vertex-disjoint *large* triangles whose vertices are the points in S. Here, by "large" we mean that all three sides of the triangles have lengths at least a constant fraction of the upper bound, $2r$ (see Lemma 4.3). (Of course, if n is not a multiple of 3, there are one or two points not part of any triangle, but this complication can be handled, as we see later.)
2. Concatenate these large triangles together in order to create a Hamiltonian path or cycle such that the distances between any 2-neighbors are large.

We elaborate these steps in the following sections.

4.1. Constructing large triangles. We describe how to construct large triangles. We assume without loss of generality that there are no two distances between points that are exactly the same. (If $d(p_i, p_j) = d(p_{i'}, p_{j'})$, where $i < j$ and $i' < j'$, then we consider edge (p_i, p_j) to be shorter than edge $(p_{i'}, p_{j'})$ if (i, j) is lexicographically less than (i', j').)

Let $\mathcal{C}$ denote a smallest circle, having a radius denoted by r, centered at one of the n points such that the corresponding (open) disk contains exactly $\lfloor n/3 \rfloor$ points of S (including the center point) interior to the circle. By definition, then, $\mathcal{C}$ passes through exactly one point of S (which is not, of course, counted as one of the $\lfloor n/3 \rfloor$ interior points).

Lemma 4.3. *The max-min distance in the cycle version of a* 2*-neighbor TSP is at most* $2r$.

Proof. The proof is immediate from Lemma 3.2. □

Computationally, r and $\mathcal{C}$ can be found as follows. For each point we find as a candidate the $\lfloor \frac{n}{3} \rfloor$th shortest edge among all edges incident on the point. The shortest edge among these n candidates defines r and the center of $\mathcal{C}$. Clearly, the computation takes $O(n^2)$ time.

For simplicity of exposition, we now assume that n is a multiple of 3. We will handle the other cases later. After computing r and $\mathcal{C}$, we use Algorithm *Big-Triangles* to construct $n/3$ vertex-disjoint large triangles, using n points of S as vertices, as follows.

Algorithm Big-Triangles**.** The algorithm takes r, $\mathcal{C}$, and S as input, and constructs $n/3$ vertex-disjoint large triangles using n points of S as vertices. It consists of the following three phases.

1. *The point-point matching phase.* Consider the bipartite graph $G = (V_1, V_2, E)$, where V_1 is the set of points inside the disk defined by $\mathcal{C}$ and V_2 is the set of points outside the disk, including the point on the boundary $\mathcal{C}$. Edge set E consists of "long" edges from V_1 to V_2, between two nodes whose distance is at least r. Refer to Figure 4.1. Note that $|V_1| = n/3$ and $|V_2| = (2n)/3$. Perform a maximum cardinality matching on G. By Lemma 4.4 (below), all nodes of V_1 are matched. For each matched pair, create a segment e by connecting the two points. Clearly, each such e is of length at least r.

Lemma 4.4. *There is a matching in G in which all nodes of V_1 are matched.*

Proof. We use Hall's theorem [9], which states that a "complete" matching of V_1 into V_2 exists if and only if for every subset A of V_1, $|A| \le |\Gamma(A)|$, where $\Gamma(A)$ is a set of those vertices of V_2 that are adjacent to at least one vertex in A.

We assume, by way of contradiction, that there exists a set $A \subseteq V_1$ for which $|A| > |\Gamma(A)|$. Consider now the nodes of $V_2 \setminus \Gamma(A)$. The cardinality of this set satisfies

$$|V_2 \setminus \Gamma(A)| = |V_2| - |\Gamma(A)| > |V_2| - |A| \ge n/3,$$

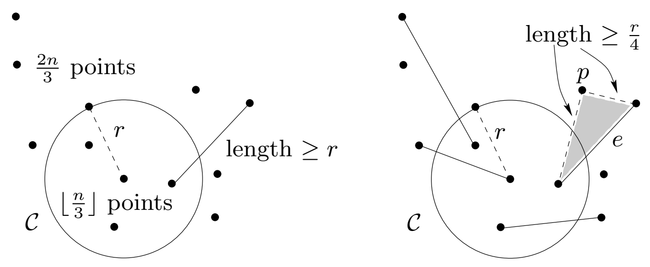


FIG. 4.1. *Left: Point-point matching phase of* Big-Triangles*. Right: Segment-point matching phase.*

since $|A| \leq |V_1| = n/3$ and $|V_2| = (2n)/3$. Furthermore, by definition, all of these points are at a distance less than r from all points of A. A disk, centered at an arbitrary point of A of radius r contains this center point as well as all of the points of $V_2 \setminus \Gamma(A)$, i.e., at least $n/3 + 2$ points in total, in its *interior*, contradicting the original choice of r. ▯

2. *The segment-point matching phase.* We now create a bipartite graph $H = (W_1, W_2, \mathcal{E})$, in which the nodes W_1 are the edges (segments) e that form the point-point matching created in phase 1, and the nodes W_2 are the points of V_2 that are *not* matched in phase 1. Note that $|W_1| = |W_2| = n/3$. The edge set $\mathcal{E}$ is specified as follows: there is an edge between node $e \in W_1$ and point $p \in W_2$ if and only if p is separated from *both* endpoints of the segment e by a distance at least $r/4$. Refer to Figure 4.1. Perform a maximum (maximal suffices) matching on H. Now, for each matched pair, e and p, create a triangle by connecting p with the two endpoints of e. If the matching is a perfect matching, stop (there are $n/3$ triangles whose edge lengths are at least $r/4$); otherwise go to the next phase.

3. *The swapping phase.* Let P be the set of "left-over" points not yet in any triangle, and E' be the set of "left-over" segments (created in phase 1) connecting points of S that are not yet in any triangle. Note that $|P| = |E'| = n/3 - T$, where T is the size of the matching in phase 2.

Let $e = (v_1, v_2) \in E'$ be an arbitrary left-over segment, and let $C_i = \mathcal{D}_{v_i}(r/2)$ and $c_i = \mathcal{D}_{v_i}(r/4)$, for $i = 1, 2$, be the (open) disks centered at v_i with radii $r/2$ and $r/4$, respectively. Since e is not matched with any point of P in phase 2, all points in P are within distances less than $r/4$ from at least one endpoint of e, i.e., $P \subset c_1 \cup c_2$. We define $P_i = c_i \cap P$, $i = 1, 2$; we refer to P_i as a *cluster*. Since e has length at least r, $c_1 \cap c_2 = \emptyset$; thus, (P_1, P_2) partition P. Now, the partition (P_1, P_2) has been defined in terms of the particular edge $e \in E'$; however, the choice of e does not matter as far as the partition is concerned.

LEMMA 4.5. *Each choice of $e \in E'$ results in the same partition of P.*

Proof. Suppose that two distinct edges, $e = (v_1, v_2) \in E'$ and $e' = (v_1', v_2') \in E'$, give rise to two distinct partitions (P_1, P_2) and (P_1', P_2'), respectively. Then, there is some pair of points, $p_1, p_2 \in P$, that is split by one partition but not by the other: without loss of generality, assume that $p_1 \in P_1$ and $p_1 \in P_1'$, while $p_2 \in P_2$ and $p_2 \in P_1'$. Then, $d(p_1, p_2) > r/2$, since $d(v_1, p_1) + d(p_1, p_2) + d(p_2, v_2) \geq d(v_1, v_2) \geq r$ implies that $d(p_1, p_2) > r - d(v_1, p_1) - d(p_2, v_2) \geq r - (r/4) - (r/4) = r/2$. On the other hand, since $p_1, p_2 \in P_1'$, we get that $d(p_1, p_2) \leq d(p_1, v_1') + d(v_1', p_2) < r/4 + r/4 = r/2$ (since $d(p_1, v_1') \neq d(v_1', p_2)$, and $d(p_1, v_1'), d(v_1', p_2) \leq r/4$), a contradiction. ▯

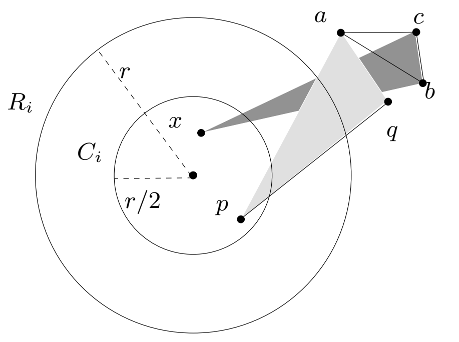


FIG. 4.2. *Swapping: $\triangle abc$ is a good triangle with respect to P_i, having edge lengths at least $r/4$. Then, triangles $\triangle pqa$ and $\triangle xbc$ have edge lengths at least $r/8$.*

LEMMA 4.6. *If $P_i \neq \emptyset$, then each segment of E' has exactly one endpoint in C_i.*

Proof. Let $p \in P_i$ and let $e' \in E'$ be any left-over segment. Since p is not matched in phase 2 with e', we know that $d(p, v') < r/4$ for an endpoint v' of e'. Then, $d(v', v_i) \leq d(v', p) + d(p, v_i) < r/2$, implying that $v' \in C_i$, so e' has at least one endpoint in C_i. Further, e' cannot have both endpoints in C_i, since e' has length greater than r, while C_i has radius $r/2$. ◻

We call (P_1, P_2) a 1-*partition* if some P_i is empty, and a 2-*partition* otherwise. For simplicity, unless otherwise stated, the following discussions assume that (P_1, P_2) is a 2-partition; the case of a 1-partition is easily handled.

For $i = 1, 2$, let $R_i = \mathcal{D}_{v_i}(r)$ be the disk of radius r that is concentric with C_i. We call a triangle $\triangle abc$ constructed from phase 2 *good* with respect to P_i if a, b, and c are all outside or on the boundary of R_i, and *bad* otherwise.

LEMMA 4.7. *Let $\triangle abc$ be a good triangle with respect to P_i whose edge lengths are at least $r/4$. Consider any segment $e = (p, q) \in E'$. Since we know (Lemma 4.6) that e has exactly one endpoint in C_i, we can assume without loss of generality that $p \in C_i$ and that $q \notin C_i$ is closer to b than to a and c. Then, for any $x \in P_i$, the triangles $\triangle pqa$ and $\triangle xbc$ have edge lengths at least $r/8$.*

Proof. Points a, b, and c are each at distance at least $r/2$ from each of x and p, since a, b, and c are each at distance at least r from the center, v_i, of R_i, while x and p are each within distance $r/2$ from v_i. Also, recall that e has length at least r. If $d(b, q) \geq r/8$, then $\triangle pqa$ and $\triangle xbc$ have edge lengths at least $r/8$. Otherwise, $d(q, a) \geq r/8$, since $d(q, a) + d(q, b) \geq d(a, b) \geq r/4$, and again $\triangle pqa$ and $\triangle xbc$ have edge lengths at least $r/8$. Refer to Figure 4.2. ◻

The operation of replacing a good triangle ($\triangle abc$), a left-over edge ($e = (p, q) \in E'$), and a left-over point ($x \in P_i$) with a pair of triangles ($\triangle pqa$ and $\triangle xbc$) is called a *swapping operation*. The above lemma assures us that the edge lengths of the new triangles are "large" (at least $r/8$), if the new triangles are as specified in the lemma.

Phase 3 proceeds by constructing C_i and R_i, $i = 1, 2$, then performing the swapping operation for all points in P_1 (*swapping stage* 1), and then performing the same operation for all points in P_2 (*swapping stage* 2). By Lemma 4.8 (given below), there are always enough good triangles to complete the swapping operations, so that all points are now vertices of $n/3$ triangles, each having edge lengths greater than $r/8$.

LEMMA 4.8. *There are always enough good triangles to complete the swapping operations on all points in P_1 and all points in P_2. Also, after completion of Algorithm* Big-Triangles, *the smallest edge length of all $n/3$ triangles is at least $r/8$.*

Proof. We first consider swapping stage 1. At the end of phase 2, suppose there are T triangles, ℓ unmatched segments (and points), $|P_1| = \ell_1$, and $|P_2| = \ell_2$; then we have $T + \ell = n/3$, and $\ell_1 + \ell_2 = \ell$. Let $T_{1,bad}$ be the number of bad triangles with respect to P_1. If t_1 is the total number of vertices of these triangles that are inside R_1, then $t_1 \geq T_{1,bad}$, since each bad triangle has at least one vertex inside R_1. Recall that R_1 is a disk of radius r; thus, by our choice of r, R_1 contains at most $n/3$ points. These points include ℓ endpoints from unmatched segments, ℓ_1 points of P_1, and t_1 vertices from bad triangles. Therefore we have $n/3 \geq \ell + \ell_1 + t_1 \geq \ell + \ell_1 + T_{1,bad}$. Let $T_{1,good}$ be the number of good triangles with respect to P_1. Then $T_{1,good} = T - T_{1,bad} = (n/3 - \ell) - T_{1,bad} \geq (n/3 - \ell) - (n/3 - \ell - \ell_1) = \ell_1$, i.e., there are enough good triangles for the points in P_1. For swapping stage 2, the situation is the same except that now $\ell = \ell_2$, meaning that we have a 1-partition, and the same counting argument applies. Finally, the smallest edge length of triangles at the end of phase 2 is at least $r/4 = \delta$. During phase 3, let $\triangle abc$ be a good triangle for P_1, and in swapping stage 1 we swap $\triangle abc$ with a segment $e = (p, q) \in E'$ and a point $x \in P_1$ to produce $\triangle pqa$ and $\triangle xbc$, as shown in the proof of Lemma 4.7. Notice that $\triangle pqa$, with edge lengths at least $\delta/2$, is now *bad* for P_2 (and also for P_1) and will not be used for swapping in swapping stage 2. Also, $\triangle xbc$ has edge lengths at least δ rather than $\delta/2$. Applying Lemma 4.7 again in swapping stage 2, the smallest edge length of the final triangles is still at least $\delta/2 = r/8$. □

Now, for n not being a multiple of 3, i.e., $n = 3k + l$, $l = 1$ or 2, we perform exactly the same operations. Note that the disk defined by $\mathcal{C}$ contains k points of S; $2k + l$ points of S lie outside the disk (including the one point on its boundary, $\mathcal{C}$). Applying Algorithm Big-Triangles then gives k large triangles, using up all k points in the interior of $\mathcal{C}$, with one or two points left over, outside, or on the boundary of $\mathcal{C}$.

To analyze the complexity, we note that phases 1 and 2 are just maximum cardinality matching and maximal matching for bipartite graphs, which can be done in time $O(n^{2.5})$ (either by the $O(n^{2.5})$-time algorithm of [11] or by the $O(\sqrt{n}m)$-time algorithm of [7], where $m = O(n^2)$ is the number of edges in the graphs). In phase 3, we consider an arbitrary edge $e \in E'$ and the disks (C_i and R_i) of radii $r/2$ and r centered at its endpoints, identify the corresponding good/bad triangles, and perform local swappings, in $O(n)$ time per swap. Finally, as analyzed before, radius r and circle $\mathcal{C}$ are determined in $O(n^2)$ time.

THEOREM 4.9. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an algorithm that constructs in $O(n^{2.5})$ time $\lfloor n/3 \rfloor$ triangles from points in S such that the smallest edge length of the triangles is at least $r/8$, where r is the radius of a smallest disk centered at a point of S, containing exactly $\lfloor n/3 \rfloor$ points in its interior, and one additional point on its boundary.*

4.2. Concatenating the triangles. We first show how to concatenate the triangles into a Hamiltonian path, and then show how to make a Hamiltonian cycle.

THEOREM 4.10. *Given a list of triangles $\Delta_1, \Delta_2, \ldots, \Delta_k$, whose minimum edge length is δ, the triangles can be chained into a path such that any two nodes that are 2-neighbors in the path are at distance at least $\delta/2$ from each other.*

In fact, the order in which the triangles are placed in the path is arbitrary, so that any permutation on the triangles is possible. This will be used to complete a cycle.

To prove the theorem, we can arrange the order in which the vertices of a triangle is visited, according to the following lemma.

LEMMA 4.11. *We are given a triangle $\triangle abc$, whose nodes are visited in that order by a path, and a triangle $\triangle xyz$ to be added after $\triangle abc$, such that all six edges in both* *triangles are of length at least δ. Then the order of the nodes of triangle $\triangle xyz$ can be chosen such that, in the resulting path, if $p \in \{a, b, c, x, y, z\}$ and $q \in \{a, b, c, x, y, z\}$ are 2-neighbors, then $d(p, q) \geq \delta/2$.*

Proof. We need to satisfy the following requirements: (1) the first two nodes of triangle $\triangle xyz$ visited must each be at distance at least $\delta/2$ from c, and (2) the first node of triangle $\triangle xyz$ visited must be at distance at least $\delta/2$ from b. This task can be carried out as follows: (1) among nodes x, y, and z, the one that is closest to c (say, x) is visited last, and (2) between the remaining two nodes, the one that is closer to b (say, y) is visited second to last; this fixes the entire ordering of visiting the nodes. To verify, consider requirement (1) first. If $d(c, x) \geq \delta/2$, then the requirement is trivially satisfied. If $d(c, x) < \delta/2$, then we have $d(c, y) > \delta/2$, since $d(x, y) \geq \delta$ ((x, y) is an edge of triangle $\triangle xyz$) and $d(c, y) + d(c, x) \geq d(x, y)$. Similarly, we have $d(c, z) > \delta/2$. This completes the verification of requirement (1). Requirement (2) is satisfied by exactly the same argument. ▯

The proof of the theorem is now immediate. Start the path with any triangle, in any order of the vertices, and keep adding new triangles to the path, one at a time, according to the lemma.

Now we can directly apply Theorem 4.10 to obtain a Hamiltonian path, if n is a multiple of 3. Combining Theorem 4.10 and Theorem 4.9, the shortest distance among all 2-neighbors in the path is at least $r/16$. Recall that, from Lemma 3.2 and the definition of r given in section 4.1, the objective value of an optimal Hamiltonian path for max-min 2-neighbor problem is at most $2r$.

THEOREM 4.12. *Given a complete graph on set S of n points satisfying triangle inequality, where n is a multiple of 3, there exists an algorithm that constructs in $O(n^{2.5})$ time a Hamiltonian path on S whose objective value for the max-min 2-neighbor problem is at least $\frac{1}{32}$ times the optimal.*

Now consider the case in which n is *not* a multiple of 3. Let $k = \lfloor n/3 \rfloor$, and $n = 3k + l$, where $l = 1$ or 2. Recall from section 4.1 that at the completion of Algorithm Big-Triangles, we have k triangles and one or two left-over points outside or on the boundary ($\mathcal{C}$) of the disk of radius r. We will combine such left-over points with existing triangles, and then perform the concatenation.

For each left-over point $p \in S$ that is not a vertex of an existing triangle, we claim that we can find an existing triangle $\triangle abc$ (with edge length at least δ $(= r/8)$) such that p is at a distance at least r $(> \delta)$ from each of a, b, and c. We call $\triangle abc$ a *good* triangle for p. To prove this claim, note, from the definition of r, that there are at most k points (including p) in the open disk $\mathcal{D}_p(r)$. Thus, there are at most $k - 1$ triangles having at least one vertex in $\mathcal{D}_p(r)$, meaning that there exists some triangle $\triangle abc$ whose vertices are each outside (or on the boundary of) $\mathcal{D}_p(r)$, as claimed. We create a special "triangle," *abcp*.

If $n = 3k + 2$, let p and q denote the two left-over points. We know that each has a good triangle. If each has a distinct good triangle we obtain two special "triangles," each with four vertices. Otherwise, p and q have a common good triangle, $\triangle abc$. We claim then that $d(p, q) \geq r > \delta$; otherwise, the disk $\mathcal{D}_p(r)$ contains q, as well as at least one vertex of each of the $k - 1$ other large triangles, contradicting the choice of r. Thus we obtain one special "triangle" with five vertices, *abcpq*, with pairwise distances of at least δ.

We now treat the special "triangles" (*abcp*, *abcq*, or *abcpq*) as if they were triangles during the concatenation process. In order to concatenate it with its predecessor triangle, we treat it as triangle $\triangle abc$. The predecessor will fix a specific ordering of points a, b, and c to be visited; we then visit the remaining points of the "triangle" (p, q, or both p and q). In order to concatenate a triangle after it, we consider only the last point (p or q) and its two previously visited points as a triangle, and apply the same lemma (Lemma 4.11).

Combining Algorithm Big-Triangles with Theorem 4.10, we obtain an algorithm to construct a Hamiltonian path whose max-min 2-neighbor distance is at least $\frac{1}{32}$ times the optimal max-min 2-neighbor distance of a Hamiltonian cycle. We have proved the following lemma.

LEMMA 4.13. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an algorithm that constructs in $O(n^{2.5})$ time a Hamiltonian path on S whose max-min 2-neighbor distance is at least $\frac{1}{32}$ times the optimal max-min 2-neighbor distance of a Hamiltonian cycle.*

In order to construct a Hamiltonian path whose objective function value is at least $\frac{1}{32}$ times that of an optimal *path* (as opposed to an optimal *cycle*, as in the above lemma), considerably more effort is required. The modifications to the algorithm and the analysis are given in Appendix A, where we prove the following theorem.

THEOREM 4.14. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an algorithm that constructs in $O(n^{2.5})$ time a Hamiltonian path on S whose objective value for the max-min 2-neighbor problem is at least $\frac{1}{32}$ times the optimal.*

Since the Hamiltonian path on S obtained in Lemma 4.13 above has a max-min 2-neighbor distance provably close to the optimal max-min 2-neighbor distance in a *cycle*, we can use it to obtain an approximation algorithm for the cycle version of the problem, by appropriately "closing" the path into a tour.

THEOREM 4.15. *Given a complete graph on set S of n points satisfying triangle inequality, there exists an algorithm that constructs in $O(n^{2.5})$ time a Hamiltonian cycle on S whose max-min 2-neighbor distance is at least $\frac{1}{64}$ times the optimal.*

Proof. Given a list of triangles (with possibly one or two being special triangles), whose minimum edge length is at least $\delta = r/8$, we use the following process to chain the triangles into a cycle such that any two nodes that are 2-neighbors in the cycle are at distance at least $\delta/4$.

We first check if there is any pair of triangles $\triangle abc$ and $\triangle xyz$ such that two vertices of $\triangle abc$, say, a and b, are each at a distance at least $\delta/4$ from each of x, y, and z. If so, chain the triangles starting with $\triangle abc$ (visiting in that order) and end with $\triangle xyz$ (in the order required by its predecessor), by applying Lemma 4.11. The ordering of the other triangles is arbitrary.

We assume from now on that no such pair of triangles exists. We consider an arbitrary triangle $\triangle abc$, and construct disks C_a, C_b, and C_c, respectively, centered at a, b, and c, each with radius $\delta/4$. Since the search in step 1 is not successful, C_a, C_b, and C_c must together contain (in their *interiors*) at least two vertices of each of the remaining triangles (each disk contains at most one vertex of a triangle since each triangle edge has length at least δ). We say that a triangle is *covered* by a pair of disks $\{C_a, C_b\}$ if its two vertices are respectively contained in (the interiors of) C_a and C_b.

First, consider the case in which there are no special triangles.

Case (a). *One pair of disks, say, $\{C_b, C_c\}$, covers all triangles.*
Then, all triangles are of type xBC, where B and C denote the vertices, respectively, inside C_b and C_c (all such vertices form *clusters* B and C), and x denotes the third vertex (possibly $x \in A$, where cluster A is defined similarly). It is easy to see that any two points in different clusters are at a distance larger than $\delta/2$. Also, any such point

x is at a distance greater than $\delta/2$ from any point in B (resp., C). (Suppose $\triangle xb'c'$ is one such triangle with $b' \in B$. For any point $b'' \in B$, $d(b', b'') < \delta/2$, but $d(x, b') \geq \delta$, and thus $d(x, b'') \geq d(x, b') - d(b', b'') > \delta/2$.) Construct the Hamiltonian cycle by repeatedly traversing the triangles in the order of xBC, and finally going back to the starting point (denoted by $xBC \rightarrow$ starting point).

Case (b). *Two pairs of disks, say,* $\{C_b, C_c\}$ *and* $\{C_a, C_c\}$, *are necessary and sufficient to cover all triangles.*

Then there are three types of triangles: ABC, xBC, and yAC, where $x \notin A$ is at distance greater than $\delta/2$ from any point of B (resp., C) and $y \notin B$ is at distance greater than $\delta/2$ from any point of A (resp., C), by the same argument above. Construct the Hamiltonian cycle by exhausting all triangles of the same type and then going to the next type, in the order given as follows:
$ABC \rightarrow xBC \rightarrow AyC \rightarrow$ starting point.

Case (c). *All three pairs of disks are needed to cover all triangles.*

Then there are four types of triangles: ABC, xBC, yAC, and zAB, where properties analogous to those in Case (b) hold for x, y, and z. Construct the Hamiltonian cycle as follows:
$ABC \rightarrow xBC \rightarrow AyC \rightarrow ABz \rightarrow$ starting point.

Now consider the case in which there are special triangles. If there is one with five vertices, say, $uvwpq$, then at least two vertices, say, p and q, must each be "far away" from all vertices of $\triangle abc$ (since C_a, C_b, and C_c together can contain at most three vertices of $uvwpq$). Thus we can complete the chaining of triangles in step 1 by starting with $pquvw$ (in that order) and ending with $\triangle abc$ (in the order required by its predecessor). Consider therefore special triangles F with four vertices. If F has less than three vertices in the interiors of C_a, C_b, and C_c, then again F has at least two vertices each far away from all vertices of $\triangle abc$ and hence step 1 can be completed. We thus need to consider only special triangles of type $ABCt$, where t (by the same argument) is at a distance greater than $\delta/2$ from any point of A (resp., of B and of C). For Cases (a)–(c) considered above, the Hamiltonian cycle is given as follows (observe that the sequences are unchanged except for starting with the special triangles):

Case (a): $AtBC \rightarrow xBC \rightarrow$ starting point,

Case (b): $ABtC \rightarrow ABC \rightarrow xBC \rightarrow AyC \rightarrow$ starting point,

Case (c): $ABtC \rightarrow ABC \rightarrow xBC \rightarrow AyC \rightarrow ABz \rightarrow$ starting point.

The approximation factor follows by noting that $\delta/4 = \frac{r}{32}$ and that the optimal objective value is at most $2r$. □

5. Approximating the max-min m-neighbor TSP.

In this section, we show how the techniques developed in sections 4.1–4.2 can be generalized to obtain a constant-factor approximation for the max-min m-neighbor TSP, for any constant $m > 2$. For simplicity, we consider only the *path* version in which n *is a multiple of* $m+1$. The method of "closing" a path into a cycle in Theorem 4.15 and the techniques of handling the case in which n is not a multiple of $m+1$ for the path version (in Appendix A) are not easily extended to the case in which $m > 2$; we pose them as open questions.

Analogous to the techniques for 2-neighbor TSP, we use *large* $(m+1)$-*cliques* to play the role of large triangles. An $(m+1)$-clique K_{m+1} consists of $m+1$ vertices that are the points in S (thus a triangle is a 3-clique); the *clique edges* of K_{m+1} are implicitly defined by its vertices. We want K_{m+1} to be *large*, meaning that each vertex of K_{m+1} is "far away" from all the other vertices of K_{m+1} (i.e., all clique edges of K_{m+1} are "long"). Again, our overall strategy consists of the following steps:

1. Create $n/(m+1)$ vertex-disjoint *large* $(m+1)$-cliques whose vertices are the points in S.
2. Concatenate these large $(m+1)$-cliques together to create a Hamiltonian path such that the distances between any m-neighbors are large.

We elaborate these steps in the following. First, let r be the radius of a smallest circle, $\mathcal{C}$, centered at one of the n points whose corresponding disk contains exactly $n/(m+1)$ points of S (including the center point), and one additional point on its boundary. From Lemma 3.2, $2r$ is an upper bound for the max-min distance in the path version of an m-neighbor TSP (since n is a multiple of $m+1$). Again, $\mathcal{C}$ and r are easily found in $O(n^2)$ time.

After computing r and $\mathcal{C}$, we use Algorithm Big $(m+1)$-Cliques (analogous to Algorithm Big-Triangles in section 4.1), to grow the $n/(m+1)$ points in the *interior* of $\mathcal{C}$ into $n/(m+1)$ vertex-disjoint large $(m+1)$-cliques. The process consists of m phases, where in phase i we grow each i-clique into an $(i+1)$-clique.

Algorithm Big $(m+1)$-Cliques. The algorithm consists of m phases, classified into two parts.

Part 1. *Complete matching phases.* This part consists of $\lfloor m/2 \rfloor$ phases. For each phase $i = 1, 2, \ldots, \lfloor m/2 \rfloor$, perform the following.

Phase i: Construct a bipartite graph $G = (V_1, V_2, E)$, where the nodes V_1 represent the i-cliques constructed in phase $i-1$ (if $i = 1$, then the nodes V_1 are the points in the disk defined by $\mathcal{C}$) and V_2 are the left-over points. Note that $|V_1| = n/(m+1)$ and $|V_2| = n - i \cdot \frac{n}{m+1} = (m+1-i)\frac{n}{m+1}$. Edge set E consists of "long" edges from V_1 to V_2, between two nodes $K_i \in V_1$ and $p \in V_2$, where p is at distance at least r from *all* vertices of K_i. Perform a maximum cardinality matching on G. By Lemma 5.1, all nodes of V_1 are matched. For each matched pair $K_i \in V_1$ and $p \in V_2$, grow the i-clique K_i into an $(i+1)$-clique by including point p as its additional vertex. Clearly, all clique edges of all resulting $(i+1)$-cliques have lengths at least r.

Lemma 5.1. *For each phase $i = 1, 2, \ldots, \lfloor m/2 \rfloor$, there exists a matching in G in which all nodes of V_1 are matched.*

Proof. We use an argument similar to the proof of Lemma 4.4. Again, by way of contradiction, we assume that there exists a set $A \subseteq V_1$ for which $|A| > |\Gamma(A)|$, where $\Gamma(A) \subseteq V_2$ is a set of nodes that are adjacent (via edges of G) to at least one node of A. Now we have $|A| \le |V_1| = n/(m+1)$, $|V_2| = (m+1-i)\frac{n}{m+1}$, and thus $|V_2 \setminus \Gamma(A)| > |V_2| - |A| \ge (m-i)\frac{n}{m+1}$. That is, for each i-clique K_i of A, there are at least $(m-i)\frac{n}{m+1} + 1$ left-over points (in $V_2 \setminus \Gamma(A)$) that are each within distance less than r from *some* vertex of K_i. Consider i (open) disks centered at each of the i vertices of K_i, each with radius r: by the pigeonhole principle, at least one of these disks contains more than $((m-i)\frac{n}{m+1})/i$ points of $V_2 \setminus \Gamma(A)$. Since $i \le \lfloor m/2 \rfloor$, this disk contains at least $\frac{n}{m+1} + 2$ points, including the center point. This contradicts the original choice of r. ∎

Part 2. *Maximal matching and swapping phases.* This part consists of the remaining $\lceil m/2 \rceil$ phases. For each phase $i = \lfloor m/2 \rfloor + 1, \ldots, m$, perform the following.

Phase i: Let δ be the length of the shortest clique edge among all i-cliques at the end of phase $i-1$ (for $i = \lfloor m/2 \rfloor + 1$, take $\delta = r$). Construct a bipartite graph $G = (V_1, V_2, E)$ in the same way as in Part 1, except that the edge set E is specified as follows: there is an edge between two nodes $K_i \in V_1$ and $p \in V_2$ if and only if p is at a distance at least $\delta/4$ from *all* vertices of K_i. Now perform the following steps:

1. *Maximal matching step.* Perform a *maximal* matching (rather than maximum cardinality matching) on G. For each matched pair $K_i \in V_1$ and $p \in V_2$, grow

the i-clique K_i into an $(i+1)$-clique by including point p as its additional vertex. If all nodes in V_1 are matched, stop phase i here (all i-cliques in V_1 are grown into $(i+1)$-cliques); otherwise go to the next step.

2. *Swapping step.* Let $V_1' \subseteq V_1$ be the set of the left-over i-cliques of V_1 that are not matched in step 1, and $P \subseteq V_2$ the set of the left-over points that are not yet in any clique. Perform swapping operations of Lemma 5.3 on the i-cliques of V_1'. By Lemma 5.4, all of them can be grown into $(i+1)$-cliques, with the lengths of all clique edges at least $\delta/8$.

Suppose that the swapping step is performed in phase i. Let $K_i \in V_1'$ be some left-over i-clique whose vertices are $u_1, \dots, u_i$, and let $C_j = \mathcal{D}_{u_j}(\delta/2)$ be the disk centered at u_j with radius $\delta/2$, for each $j = 1, \dots, i$.

LEMMA 5.2. *Independent of the choice of the left-over i-clique K_i, the disks C_j induce a partition of the left-over points P into at most i nonempty clusters $P_1, \dots, P_h$, $h \le i$. For each $j = 1, \dots, h$, all points of P_j are enclosed in the interior of C_j (renumbering the subscripts of the vertices of K_i might be necessary), and each left-over i-clique in V_1' (including K_i) has exactly one vertex in the interior of C_j.*

Proof. This is a direct generalization of Lemmas 4.5 and 4.6. □

In phase i, let $R_j = \mathcal{D}_{u_j}(r)$ be the disk that has the same center, u_j, as C_j and has radius r, for each $j = 1, \dots, h$. We call a clique (either an i-clique or an $(i+1)$-clique) *good* for P_j if the clique has no vertex in the *interior* of R_j, and *bad* otherwise. By Lemma 5.2, each left-over i-clique in V_1' is *bad for all clusters* $P_1, \dots, P_h$.

LEMMA 5.3. *Let K_{i+1} be a good $(i+1)$-clique for cluster P_j. Then for any left-over i-clique $K_i \in V_1'$ and a left-over point $x \in P_j$, we can construct two new $(i+1)$-cliques from K_{i+1}, K_i, and x such that one new $(i+1)$-clique is bad for all $P_1, \dots, P_h$ and has clique edge lengths at least $\delta/8$, and the other has clique edge lengths at least $\delta/4$.*

Proof. Let $\delta' = \delta/4$. Note that all clique edges in K_{i+1} and in K_i have lengths at least δ'. We claim that there is always a vertex v^* of K_{i+1} that is at distance at least $\delta'/2 = \delta/8$ from all vertices of K_i. To see this, consider the vertices $u_1, \dots, u_i$ of K_i, one by one. Initially, all $i+1$ vertices of K_{i+1} are candidates for v^*. Now, u_1 can be "too close" (within distance less than $\delta'/2$) to at most one vertex of K_{i+1}: if there is a vertex v_ℓ of K_{i+1} with $d(u_1, v_\ell) < \delta'/2$, then since all clique edges in K_{i+1} have lengths at least δ', we have $d(v, v_\ell) \ge \delta'$ for each vertex $v \ne v_\ell$ of K_{i+1}, and thus $d(u_1, v) \ge d(v, v_\ell) - d(u_1, v_\ell) > \delta'/2$. Therefore u_1 can filter out at most one candidate for v^*. Now consider u_2 and the remaining i candidates. Again, u_2 can filter out at most one candidate. Repeating this process, there are i constraints (from the vertices of K_i) in total, but we have $i+1$ candidates initially. Therefore such v^* must always exist. This completes the proof of the claim.

Now we construct two new $(i+1)$-cliques from K_{i+1}, K_i, and x as follows. First, we grow K_i into an $(i+1)$-clique by including v^* as its additional vertex. Clearly, this new clique has edge lengths at least $\delta'/2 = \delta/8$. Also, since K_i is originally bad for all clusters $P_1, \dots, P_h$ by Lemma 5.2, this new $(i+1)$-clique continues to be bad for all clusters. Secondly, we construct the other $(i+1)$-clique by swapping out vertex v^* from K_{i+1} and swapping in point $x \in P_j$ as a new vertex. The fact that this clique still has edge lengths at least $\delta' = \delta/4$ follows from the observation that K_{i+1} is good for P_j (hence all vertices of K_{i+1} are at a distance at least r from the center of R_j, C_j) and that x is within distance less than $\delta/2$ from the center of C_j (by Lemma 5.2) and from the triangle inequality. □

LEMMA 5.4. *Suppose that the swapping step is performed in phase i. Then there are always enough good $(i+1)$-cliques for P_j, $j = 1, \dots, h$, such that all left-over*

i-cliques can be grown into $(i+1)$-cliques by the swapping operations of Lemma 5.3, *with final clique edge lengths at least $\delta/8$.*

Proof. This is similar to Lemma 4.8. The swapping step goes through at most h stages, where for each stage $j = 1, \ldots, h$ we perform the swapping operations of Lemma 5.3 for points in P_j, until either all points in P_j are used up (in which case we move to the next stage), or all left-over i-cliques are grown into $(i+1)$-cliques (in which case we stop the entire swapping step). Now consider any such stage j. Right before stage j, suppose there are T $(i+1)$-cliques (and hence $\frac{n}{m+1} - T$ left-over i-cliques), and ℓ left-over points that are not yet in any clique. Let $|P_j| = \ell_j$, then $\Sigma_j \ell_j = \ell$. Also, ℓ is at least $\frac{n}{m+1} - T$, the number of left-over i-cliques. We claim that all points of P_j have enough good $(i+1)$-cliques to swap with. To show this, suppose among the T $(i+1)$-cliques, $T_{j,good}$ of them are good for P_j and $T_{j,bad}$ of them are bad for P_j; $T = T_{j,good} + T_{j,bad}$. Consider all points of S in the interior of R_j: these include ℓ_j points from P_j, $\frac{n}{m+1} - T$ vertices each from a left-over i-clique (by Lemma 5.2), and t_j vertices from bad $(i+1)$-cliques, where $t_j \geq T_{j,bad}$. Since the interior of R_j contains at most $n/(m+1)$ points, we have $n/(m+1) \geq \ell_j + (\frac{n}{m+1} - T) + t_j \geq \ell_j + (\frac{n}{m+1} - T) + T_{j,bad}$. Then we have $T_{j,good} = T - T_{j,bad} \geq \ell_j$, completing the proof of the claim. Since the number of the left-over i-cliques is no more than the number of the left-over points (all of which in turn have enough good $(i+1)$-cliques to swap with), all the left-over i-cliques can eventually be grown into $(i+1)$-cliques. Also, from Lemma 5.3, after each swapping operation, the only new $(i+1)$-clique that can possibly have edge lengths less than $\delta/4$ (but at least $\delta/8$) is *bad for all* $P_1, \ldots, P_h$, and hence it will not be used for swapping hereafter in phase i. Therefore, at the end of phase i all left-over i-cliques can be grown into $(i+1)$-cliques with clique edge lengths at least $\delta/8$. ☐

From Lemma 5.4, we know that in the worst case, the length of the shortest clique edge is reduced by a factor of 8 (from δ to $\delta/8$) in each phase i of Part 2. Therefore the final $(m+1)$-cliques have edge lengths at least $r(1/8)^{\lceil m/2 \rceil}$.

THEOREM 5.5. *Given a constant $m > 2$ and a complete graph on set S of n points satisfying triangle inequality, where n is a multiple of $m+1$, there exists an algorithm that constructs in $O(n^{2.5})$ time $n/(m+1)$ vertex-disjoint $(m+1)$-cliques from points in S such that the smallest edge length of the cliques is at least $r(1/8)^{\lceil m/2 \rceil}$, where r is the radius of a smallest (open) disk centered at a point of S, containing exactly $n/(m+1)$ points of S, and one additional point on its boundary.*

Now we are ready to concatenate the $(m+1)$-cliques into a Hamiltonian path.

THEOREM 5.6. *Given a list of $(m+1)$-cliques whose minimum edge length is δ, the cliques can be chained into a path such that any two nodes that are m-neighbors in the path are at a distance at least $\delta/2$ from each other.*

Proof. This is the same as Theorem 4.10, except that we use an immediate generalization of Lemma 4.11 to chain the $(m+1)$-cliques. ☐

From Theorems 5.5 and 5.6, and the fact that $2r$ is an upper bound for the optimal objective value, we conclude with the following theorem.

THEOREM 5.7. *Given a constant $m > 2$ and a complete graph on set S of n points satisfying triangle inequality, where n is a multiple of $m+1$, there exists an algorithm that constructs in $O(n^{2.5})$ time a Hamiltonian path on S whose objective value for the max-min m-neighbor problem is at least $\frac{1}{4}(\frac{1}{8})^{\lceil m/2 \rceil}$ times the optimal.*

6. Exact algorithms for points on a line or on a circle.

We consider now the special cases in which S is a set of n points on a line or on a circle, and distances are Euclidean. In these cases, we are able to obtain linear-time (optimal) algorithms to solve exactly the maximum scatter TSP, for both the cycle and path versions. (For the path version of points on a circle, we consider only the case in which n is an odd integer or the points are equally spaced.) First, we concentrate on the cycle version of points on a line; we discuss the remaining problems and versions in Appendix B.

In the following, the n points of S lie on a line, labeled $1, 2, \dots, n$ from left to right. We distinguish between two cases, depending on whether n is odd or even. Interestingly, the latter case is more complex than the former case. In fact, when n is odd, an optimal cycle can be *specified* in $O(1)$ time, i.e., without knowing the *positions* of the nodes, a fixed specification based only on the *ordering* of the nodes is guaranteed to be an optimal cycle. Such an optimal cycle can be *constructed* and output (with the length d^* of a shortest edge of the cycle identified) in $O(n)$ time. When n is even, we show that in linear time we can identify d^*, and after that an optimal cycle can be constructed using a purely combinatorial method in $O(n)$ time.

We define the following terminology. A subset I of S containing ℓ consecutive nodes is called an *ℓ-interval*. An edge (u, v) is said to *span* the ℓ nodes in the ℓ-interval whose endpoint nodes are u and v (thus each edge spans at least two nodes). An edge is called an *ℓ-edge* if it spans ℓ nodes. We let $k = \lfloor \frac{n}{2} \rfloor$ throughout this section.

6.1. Cycle version: Points on a line (n odd). We have $n = 2k + 1$ in this case. Recall that d^* is the length of a shortest edge of an optimal cycle. By Corollary 3.3, the length of the shortest $(k + 1)$-interval is an upper bound for d^*. Now we show that this upper bound is achievable by the following specification of a Hamiltonian cycle:

> Starting from node 1, the "right" edges connect node p to node $[(k+1) + p]$, for $p = 1, 2, \dots, k$, and the "left" edges connect node $(k + q)$ to node q, for $q = 1, 2, \dots, k + 1$.

It is easy to verify that the above specification gives a Hamiltonian cycle, and that each "right" edge spans $k+2$ nodes and each "left" edge spans $k+1$ nodes. Therefore this Hamiltonian cycle is optimal.

THEOREM 6.1. *For the cycle version of the maximum scatter TSP, where the n points lie on a line, the distances are Euclidean, and n is an odd integer, there exists an algorithm that specifies an optimal Hamiltonian cycle in $O(1)$ time, and constructs such a cycle in $O(n)$ time.*

6.2. Cycle version: Points on a line (n even). We have $n = 2k$ in this case. Without loss of generality, we assume that all k-intervals on S have distinct lengths. This assumption is clearly nonrestrictive, since we can always break a tie by viewing the interval whose position is to the left of the other as having a shorter length. We say that two intervals I_1 and I_2 *overlap* if $I_1 \cap I_2 \neq \emptyset$.

LEMMA 6.2. *Let I_1 and I_2 be two overlapping k-intervals. Then in any Hamiltonian cycle on S that avoids having an edge joining two nodes of I_1, there must exist an edge joining two nodes of I_2.*

Proof. Let $S_{in} = I_1 \cap I_2$ and $|S_{in}| = \ell > 0$. Also, let $S_{out} = S \setminus (I_1 \cup I_2)$. Then since $|I_1 \cup I_2| = 2k - \ell$, we have $|S_{out}| = \ell$. Suppose on the contrary that there exists a Hamiltonian cycle C that avoids having an edge joining two nodes either both in I_1 or both in I_2. Then consider the ℓ nodes in S_{in}: they can only be connected to the ℓ nodes in S_{out}. Now there are 2ℓ edges of C going out from S_{in}, but the total degree of the nodes of S_{out} in C is 2ℓ; therefore all of the nodes of S_{out} can be connected only to the nodes of S_{in} also. This implies that the nodes in $S_{in} \cup S_{out}$ are disconnected from the other nodes of S, and thus C is not a Hamiltonian cycle, which is a contradiction. ∎

COROLLARY 6.3. *Any shortest edge in any optimal Hamiltonian cycle on S can span at most k nodes.*

Proof. By Lemma 3.1, any such edge e can span at most $k+1$ nodes. Now suppose e spans $k+1$ nodes. Let I_1 be the k-interval consisting of the leftmost k nodes spanned by e, and I_2 be the k-interval of the rightmost k nodes spanned by e. Clearly I_1 and I_2 overlap, and the optimal cycle containing e does not have any edge joining either two nodes of I_1 or two nodes of I_2, contradicting Lemma 6.2. Thus e can span at most k nodes. □

COROLLARY 6.4. *Let I_1, I_2, and I_3 be, respectively, the shortest, the second-shortest, and the third-shortest k-intervals on S, and let e_i be the edge connecting the two endpoint nodes of I_i for all $i = 1, 2, 3$. Then $d^* \leq d_{e_2}$ if I_2 overlaps with I_1, and $d^* \leq d_{e_3}$ otherwise.*

Proof. If I_1 and I_2 do not overlap, then they must be the leftmost and rightmost k-intervals on S and $I_1 \cup I_2 = S$. Thus I_3 must overlap with I_1 or I_2. The stated upper bounds follow from Lemma 6.2 and Corollary 6.3. □

Assume now that we are given the shortest k-interval, I_1, on S. We give an algorithm that specifies an optimal Hamiltonian cycle C^* by distinguishing the following two cases.

Case 1. *I_1 is the leftmost k-interval.*
Place $I_1 = \{1, 2, \ldots, k\}$ and $I_2 = \{k+1, \ldots, 2k\}$ on a *schematic cycle* as shown in Figure 6.1(a), (b). Produce edges $e_1 = (1, k+1)$ and $e_2 = (k, 2k)$.

Start visiting node 1, and then repeatedly visit the kth node in the schematic cycle counted *counterclockwise* from the current node (the current node is counted as the first node), i.e., visiting nodes $k+2, 3, k+4, \ldots$, until reaching an endpoint node of e_2. Note that this visits every other node in I_1 and in I_2 in *increasing* order (nodes $1, 3, \ldots$ in I_1 and nodes $k+2, k+4, \ldots$ in I_2). If k is an even integer (see Figure 6.1(a)), then node $2k \in I_2$ is finally reached; if k is odd (see Figure 6.1(b)), then node $k \in I_1$ is finally reached. In either case, follow e_2 to visit the other endpoint node of e_2. Now, complete the other half of C^* by a symmetric method: repeatedly visit the kth node in the schematic cycle, this time counted *clockwise* from the current node, visiting the remaining nodes in I_1 and in I_2 in *decreasing* order, until reaching node $k+1$. Finally, follow e_1 to go back to node 1 (see Figure 6.1(a), (b)).

It is easy to verify that C^* is a Hamiltonian cycle. Also, all edges in C^* other than e_1 and e_2 span k nodes on one side and $k+2$ nodes on the other side of the schematic cycle, and e_1 and e_2 span $k+1$ nodes on both sides. In addition, all edges in C^* avoid joining two nodes that are both in I_1 or both in I_2. (I_1 is the shortest k-interval, and I_2 may happen to be the second shortest k-interval.) Therefore, C^* achieves the upper bound given in Corollary 6.4 and is an optimal Hamiltonian cycle.

Case 2. *I_1 is not the leftmost k-interval.*
First, relabel the nodes of S, by labeling the nodes of I_1 from 1 to k, and continuing the labeling in a wrap-around fashion (see Figure 6.1(c)). Then, use the new labels and apply the same procedure of Case 1 to produce C^*. Since the wrap-around effect has already been taken into account in Case 1 (via the schematic cycle), it is clear that C^* is an optimal Hamiltonian cycle.

Clearly, by considering each of the $k+1$ possible k-intervals we can, in $O(n)$ time, identify the length d^*. Then, we can apply the above algorithm to construct an optimal Hamiltonian cycle in additional $O(n)$ time.

THEOREM 6.5. *For the cycle version of the maximum scatter TSP, where the n points lie on a line, the distances are Euclidean, and n is an even integer, there exists*

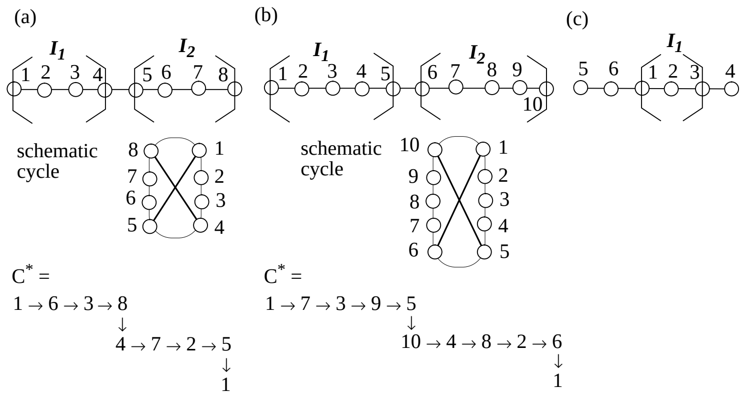


FIG. 6.1. *Examples of C^* specified by the algorithm:* (a) $n = 8$ *and* $k = 4$ *is even*; (b) $n = 10$ *and* $k = 5$ *is odd*; (c) *wrap-around relabeling when* I_1 *is not the leftmost* k*-interval.*

an algorithm that computes an optimal Hamiltonian cycle in $O(n)$ *time.*

In Appendix B we consider extensions of these results to the cycle version of points on a circle (Theorem B.2), and the path version of points on a line (Theorem B.4) or on a circle (Corollary B.5).

7. Conclusion.

We conclude with the following open problems:

- What is the complexity of the geometric maximum scatter TSP for points in the Euclidean plane? We conjecture that it is NP-hard. The recent result of Fekete [8] shows that the problem is indeed NP-hard in Euclidean spaces of dimension three or greater.
- Can better bounds for the max-min m-neighbor TSP for any $m \geq 1$ be obtained in the geometric case?
- For points on a line or on a circle, can we solve the max-min m-neighbor TSP exactly for $m > 1$? We conjecture that there exists a polynomial-time algorithm, perhaps similar to the one we presented for the case $m = 1$.
- Can we solve the maximum scatter TSP exactly for points lying on the boundary of an arbitrary convex body?

Appendix A. Max-min 2-neighbor Hamiltonian path: n not a multiple of 3. We already gave an algorithm to construct a Hamiltonian path whose objective value for the max-min 2-neighbor problem is at least $\frac{1}{32}$ times the optimal, when n is a multiple of 3 (Theorem 4.12). Now we give algorithms achieving the same factor for the remaining cases, when $n = 3k + l$, for $l = 1, 2$. Surprisingly, the case of $n = 3k + 2$ can be easily solved, but the case of $n = 3k+1$ is much more difficult, and we overcome the difficulty by refining our Algorithm Big-Triangles and utilizing metric properties of the distance matrix. Also worth noting is that in the last case of our algorithm, for $n = 3k + 1$, we actually produce an *optimal* Hamiltonian path.

One major difference between the cycle and path versions, when n is not a multiple of 3, is on the upper bounds for the optimal objective values. Let r be the radius of a smallest circle, $\mathcal{C}$, centered at one of the n points that contains exactly $\lceil n/3 \rceil = k + 1$ points in its interior (including the center point), and one additional point on its boundary.

LEMMA A.1. *The max-min distance in the path version of a* 2*-neighbor TSP is at most* $2r$*.*

Proof. The proof is immediate from Lemma 3.2. ∎

Notice that $\mathcal{C}$ now contains $k+1$ points in its *interior*, as opposed to just k points in the interior in the cycle version given in Lemma 4.3.

When $n = 3k + 2$, we add one *fake point* Q far away from all n points in S and then apply Algorithm Big-Triangles on this new set of $n' = 3k + 3$ points. The circle $\mathcal{C}$ is the smallest one containing $n'/3 = k + 1$ points in its interior and one additional point on the boundary, as desired. Suppose $\triangle Qyz$ is the (big) triangle containing Q produced by Algorithm Big-Triangles. Then in applying Theorem 4.10 to chain the $k + 1$ triangles into a Hamiltonian path, we start with $\triangle Qyz$, visiting the vertices in that order, and chain the remaining triangles in an arbitrary order. Finally, we remove Q and obtain a Hamiltonian path on S with approximation factor $\frac{1}{32}$.

When $n = 3k + 1$, we apply the same trick, adding two fake points Q and Q' far away from all n points in S as well as from each other, and apply our algorithm on this new set of $n' = 3k + 3$ points. Again circle $\mathcal{C}$ and its radius r are as desired. If we can make sure that Q and Q' are in the same triangle, say, $\triangle QQ'z$, among the $k + 1$ triangles constructed by Algorithm Big-Triangles, then we can again chain the triangles by visiting Q, Q', and z first (in that order), followed by the remaining triangles, and finally remove Q and Q' from the resulting path. This will give a desired Hamiltonian path on S.

In the rest of this section, we show how to modify Algorithm Big-Triangles so that it constructs $k + 1$ triangles whose shortest edge length is at least $r/8$, such that Q and Q' are in the same triangle. There is one case, however, in which this is not achieved, but instead we are able to produce an optimal path for this case.

Let $a \in S$ be the center point of $\mathcal{C}$. First, we want Algorithm Big-Triangles to produce $\triangle QQ'a$ at the end of phase 2. This can be done as follows: In phase 1 we match $a \in V_1$ with $Q \in V_2$ but leave $Q' \in V_2$ unmatched, and in phase 2 we match segment (Q, a) with point Q' first and then complete the matching. We have to show that there is still a complete matching in phase 1.

Lemma A.2. *In phase* 1 *of Algorithm Big-Triangles, there is a matching in* G *in which all nodes of* V_1 *are matched, with* $a \in V_1$ *and* $Q \in V_2$ *being a matched pair and* $Q' \in V_2$ *being left unmatched.*

Proof. Let $V_1' = V_1 \setminus \{a\}$ and $V_2' = V_2 \setminus \{Q, Q'\}$. We want to match all nodes in V_1' with nodes in V_2'. We use the same proof of Lemma 4.4, except that now $|A| \leq |V_1'| = n'/3 - 1$, and $|V_2'| = (2n')/3 - 2$. Thus $|V_2' \setminus \Gamma(A)| > |V_2'| - |A| \geq n'/3 - 1$, i.e., $|V_2' \setminus \Gamma(A)| \geq n'/3$. Again, any circle centered at an arbitrary point of A with radius r contains at least $n'/3 + 1 = k + 2$ points in its *interior*, contradicting the original choice of r. ∎

In phase 3, we have to make sure that Q and Q' still stay in the same triangle after swapping operations. A natural attempt to achieve this is to try to show, by modifying the argument of Lemma 4.8, that there are still enough good triangles to perform the swappings, even *without using* $\triangle QQ'a$ during the swappings. Unfortunately this method does not work, and thus we have to employ additional techniques to perform careful swappings.

Recall from Lemmas 4.5 and 4.6 that at the beginning of phase 3, the set P of left-over points is partitioned into (P_1, P_2) by the set E' of left-over segments, where (P_1, P_2) is either a 2-partition or a 1-partition. Also, the triangles constructed at the end of phase 2 have edge lengths at least $r/4$.

Lemma A.3. *If* (P_1, P_2) *is a* 2*-partition, then we can complete all swapping operations on* P_1 *and on* P_2 *in phase* 3 *of Algorithm Big-Triangles, constructing* $k+1$

triangles with edge lengths at least $r/8$, *such that* Q *and* Q' *stay in the same triangle.*

Proof. Let $e = (v_1, v_2)$ be a left-over segment in E', where v_i is within distances less than $r/4$ from all points in P_i for $i = 1, 2$ (see Lemma 4.5). Recall that e is constructed in phase 1 and that one of its endpoints, say, v_1, is inside $\mathcal{C}$, and the other endpoint, v_2, is outside or on the boundary of $\mathcal{C}$. Let R_i and C_i, $i = 1, 2$, as defined in the description of phase 3, be centered at v_i. Recall that the radiuses of R_i and C_i are r and $r/2$, respectively; also, a triangle is good for P_i if all three vertices are outside or on the boundary of R_i, and bad otherwise. Now $d(v_1, a) < r$, since v_1 is inside $\mathcal{C}$ and a is the center of $\mathcal{C}$, and thus a is inside R_1. This means that $\triangle aQQ'$ is *bad* for P_1 and is not needed for swapping the points of P_1. We first perform swappings on P_1 (swapping stage 1), with $\triangle aQQ'$ untouched. Now we perform swappings on P_2 (swapping stage 2). Observe that v_2 is outside or on the boundary of $\mathcal{C}$, so $d(a, v_2) \geq r$ and thus $\triangle aQQ'$ is *good* for P_2. Suppose in swapping stage 2 we want to swap $\triangle aQQ'$ with a point $x \in P_2$ and a left-over segment $e' = (p, q)$, where p is closer to P_1 and q is closer to P_2 (and again by Lemma 4.5 p is within distances less than $r/4$ from all points of P_1, and similarly for q and points of P_2). We claim that we can construct triangles $\triangle xQQ'$ and $\triangle apq$ so that their edge lengths are large enough. (Note that Q and Q' stay in the same triangle.) Clearly, $\triangle xQQ'$ satisfies the condition. To prove the claim, observe that q is inside C_2 and thus $d(q, v_2) < r/2$. But since $d(a, v_2) \geq r$, we have $d(a, q) \geq d(a, v_2) - d(q, v_2) > r/2$. Also, by the construction in phase 1, any left-over point $t \in P_1$ (before swapping stage 1) is *outside or on the boundary of* $\mathcal{C}$, i.e., $d(a, t) \geq r$. But as already mentioned, $d(p, t) < r/4$ by Lemma 4.5. Therefore $d(a, p) \geq d(a, t) - d(p, t) > (3r)/4$. This shows that all edge lengths in $\triangle apq$ are also large enough, which completes the proof of the claim. Now the cardinality of P_2 and the number of good triangles for P_2 are both decreased by 1, so there are enough good triangles remaining to swap with the rest of the points in P_2 (as shown in Lemma 4.8), with $\triangle xQQ'$ untouched—actually it is a bad triangle for P_2 now. This shows that the conditions of the lemma are satisfied. ☐

A nice property for (P_1, P_2) being a 2-partition is that the points of P_1 and P_2 are all outside or on the boundary of $\mathcal{C}$, and they also "confine" the endpoints of left-over segments, so that the endpoints of the left-over segments are far enough from a. When (P_1, P_2) is only a 1-partition, we do not have this nice property, and a more delicate technique is needed.

THEOREM A.4. *If* (P_1, P_2) *is a* 1*-partition with* $P_1 = P$, *then we can either complete all swapping operations on* P_1 *in phase* 3 *of Algorithm* Big-Triangles, *constructing* $k + 1$ *triangles with edge lengths at least* $r/8$, *such that* Q *and* Q' *stay in the same triangle*, *or otherwise we can construct an optimal max-min* 2*-neighbor Hamiltonian path on* S.

Proof. Let R_1 and C_1 be centered at any endpoint v_1 of a left-over segment such that v_1 is within distance less than $r/4$ from all points of P_1 (such an endpoint must exist for every segment; see Lemmas 4.5 and 4.6). If there is such v_1 inside $\mathcal{C}$, then $d(a, v_1) < r$. The conditions of the lemma are satisfied in this case, since letting R_1 and C_1 be centered at this v_1 means that a is inside R_1, i.e., $\triangle aQQ'$ is bad for P_1 and thus is not touched in the swapping operations on P_1. Therefore let us assume that all such endpoints v_1 are outside or on the boundary of $\mathcal{C}$ and thus $\triangle aQQ'$ is good for P_1. Let R_1 and C_1 be centered at any arbitrary such v_1, and call this center d. Let R_1' be a circle centered also at d with radius $(5r)/8$. Now we *redefine* a triangle to be good/bad for P_1 using R_1', namely, a triangle is *good* for P_1 if no vertex is inside R_1', and *bad* otherwise. The swapping lemma (Lemma 4.7) still works with this new definition of good/bad triangles, since any point not in the interior of R_1' is at a distance greater than $(5r)/8 - r/2 = r/8$ from each point inside C_1. Also, the two new triangles produced by a swapping operation are both *bad* for P_1, and thus there are no further shrinkings on their edge lengths. In the following, good/bad triangles are all defined by using R'_1. We consider two cases.

Case (I). There are less than $n'/3$ points inside R'_1 or some bad triangle has more than one vertex inside R'_1.

CLAIM A.1. *There are at least* $|P_1|+1$ *good triangles for* P_1, *and thus* $\triangle aQQ'$ *can be left untouched in the swapping operations on* P_1, *i.e., we can perform the desired swappings on* P_1.

Proof. The proof is a refinement of the proof of Lemma 4.8. At the end of phase 2, suppose there are T triangles, $T_{1,bad}$ bad triangles for P_1, t_1 bad triangle vertices inside R'_1, and ℓ left-over segments. Then we have $|P_1| = \ell$, $T + \ell = n'/3$, and $T_{1,bad} \leq t_1$. If R'_1 has less than $n'/3$ points in the interior, then $T_{1,bad}+\ell+\ell \leq t_1+2\ell < n'/3$; if some bad triangle has more than one vertex inside R'_1, then $T_{1,bad} < t_1$, i.e., $T_{1,bad}+\ell+\ell < t_1+2\ell \leq n'/3$. In both cases, we have $T_{1,bad} \leq n'/3-2\ell-1$. The number $T_{1,good}$ of good triangles for P_1 is then given by $T_{1,good} = T - T_{1,bad} \geq (n'/3 - \ell) - (n'/3 - 2\ell - 1) = \ell + 1 = |P_1| + 1$. ☐

Case (II). Otherwise, R'_1 has exactly $n'/3$ points in its *interior* and each bad triangle for P_1 has exactly one vertex inside R'_1. This means that there are exactly $\ell = |P_1|$ good triangles, *including* $\triangle aQQ'$. There are three subcases.

(II.1). There exists some point s inside $\mathcal{C}$, $s \neq a$, such that $d(s,a) \geq r/8$.

CLAIM A.2. *We can perform the desired swappings on* P_1.

Proof. We perform the task by considering the cases below.

Case (a). s is an endpoint of some left-over segment $e = (s,q)$. Since q is outside or on the boundary of $\mathcal{C}$, we have $d(a,q) \geq r$. Also, we already have $d(a,s) \geq r/8$. We swap $\triangle aQQ'$ with segment e and point $x \in P_1$ and produce two triangles $\triangle asq$ and $\triangle xQQ'$. Notice that $\triangle xQQ'$, with Q and Q' staying in the same triangle, is bad for P_1 and will not be touched from now on. Also, both $|P_1|$ and $T_{1,good}$, the number of good triangles for P_1, are decreased by 1 and the swapping operations can be completed as before.

Case (b). No such s exists as an endpoint of a left-over segment (i.e., each left-over segment has one endpoint within distance less than $r/8$ from a) and s is a vertex of some triangle $\triangle suv$.

Note that by the construction in phases 1 and 2, $\triangle suv$ has exactly one vertex inside $\mathcal{C}$; since s is inside $\mathcal{C}$, we have $d(a,u) \geq r$ and $d(a,v) \geq r$. We want to swap $\triangle aQQ'$ with a point $x \in P_1$ and a left-over segment (p,q), where p is inside $\mathcal{C}$ (and thus $d(p,a) < r/8$) and q is inside C_1.

1. $\triangle suv$ is good for P_1. We produce triangles $\triangle xQQ'$, $\triangle asv$, and $\triangle upq$. To verify, observe that $d(a,s) \geq r/8$ and $d(a,v) \geq r$, so $\triangle asv$ is fine. As for $\triangle upq$, we have $d(u,p) > (7r)/8$, since $d(u,a) \geq r$ and $d(p,a) < r/8$. Also, u is outside or on the boundary of R'_1 (since $\triangle suv$ is good), so $d(u,d) \geq (5r)/8$. Recall that d is the center of R'_1 and that C_1 is also centered at d with radius $r/2$. Now q is inside C_1, so $d(q,d) < r/2$. Therefore $d(u,q) \geq d(u,d) - d(q,d) > r/8$, which completes the validity of $\triangle upq$. Note that $\triangle asv$ is good, $\triangle upq$ is bad, and $\triangle xQQ'$ is bad and will not be touched from now on. Again, both $|P_1|$ and $T_{1,good}$ are decreased by 1 and the desired conditions are satisfied.

2. $\triangle suv$ is bad for P_1. Recall that each bad triangle has exactly one vertex inside R'_1. If s is inside R'_1, then both u and v are outside or on the boundary of R'_1; otherwise, one of u and v, say, v, is inside R'_1 (and s and u are outside or on the boundary of R'_1).

In both cases, we again produce triangles $\triangle xQQ', \triangle asv$, and $\triangle upq$. The validities of these triangles are established by the same arguments as before. Now $\triangle asv$ is *bad* (the other two triangles are bad as well), but originally $\triangle suv$ was bad also. Again $|P_1|$ and $T_{1,good}$ are both decreased by 1 and the desired conditions are satisfied. ∎

(II.2). All $n'/3$ points inside $\mathcal{C}$ (including a) are within distances less than $r/8$ from a, but there exists some point t inside R_1', $t \neq d$, such that $d(t, d) \geq r/8$.

Claim A.3. *The interiors of $\mathcal{C}$ and of R_1' contain two disjoint clusters, each with $n'/3$ points, and the distance between any two points in different clusters is larger than $(7r)/8$.*

Proof. Recall that $d(a, d) \geq r$ ($\triangle aQQ'$ has no vertex inside R_1 in the first place). Now any point inside R_1' is within distance less than $(5r)/8$ from d, so any point inside R_1' is at distance at greater than $r - (5r)/8 = (3r)/8$ from a. But any point inside $\mathcal{C}$ is within distance less than $r/8$ from a. Therefore all $n'/3$ points inside R_1' are outside or on the boundary of $\mathcal{C}$ (and thus at distance at least r from a). This shows the disjointness of the two clusters. Let y and z be two points inside R_1' and inside $\mathcal{C}$, respectively. Then $d(y, z) > (7r)/8$, since $d(y, a) > r$ and $d(z, a) < r/8$. ∎

Claim A.4. *We can perform the desired swappings on P_1.*

Proof. Recall that d is an endpoint of some left-over segment $e = (p, d)$, where p is inside $\mathcal{C}$ (and thus $d(p, a) < r/8$). We perform the task by considering the cases below.

Case (a). t is a point in P_1. We produce $\triangle tpd$, which is valid since $d(t, d) \geq r/8$ (the condition of Case (II.2)) and $d(t, p) > (7r)/8$ (t and p are in different clusters). Now $|P_1|$ is decreased by 1 ($|P_1| = \ell - 1$) but $T_{1,good}$ is still ℓ. Thus we can complete the swappings on P_1 *without using* $\triangle aQQ'$.

Case (b). t is an endpoint of a left-over segment $e' = (p', t)$, where p' is inside $\mathcal{C}$. Let x be any point of P_1. We produce $\triangle tpd$ and segment (p', x). Note that $\triangle tpd$ is valid as shown in Case (a), and segment (p', x) has a length larger than $(7r)/8$, since p' and x are in different clusters. Again $|P_1|$ is decreased to $\ell - 1$ but $T_{1,good}$ is still ℓ, so we can complete the desired swappings on P_1.

Case (c). t is a vertex of a (bad) triangle $\triangle tuv$ for P_1. Since each bad triangle has exactly one vertex inside R_1', u and v are outside or on the boundary of R_1', so $d(u, d) \geq (5r)/8$ and $d(v, d) \geq (5r)/8$. Let x be any point in P_1. Since x and (p, d) are the left-over point and segment, by the construction of phase 2, $d(x, d) < r/4$. Therefore $d(u, x) > (5r)/8 - r/4 = (3r)/8$; similarly, $d(v, x) > (3r)/8$. Thus we can produce triangles $\triangle uvx$ and $\triangle tpd$ (whose validity is established as before). Again $|P_1|$ is decreased to $\ell - 1$ but $T_{1,good}$ is still ℓ, so we can complete the desired swappings on P_1. ∎

(II.3). All $n'/3$ points inside $\mathcal{C}$ (including a) are within distance $< r/8$ from a and all $n'/3$ points inside R_1' (including d) are within distance $< r/8$ from d.

Claim A.5. *We can construct an optimal max-min 2-neighbor Hamiltonian path on S.*

Proof. As in Case (II.2), the interiors of $\mathcal{C}$ and of R_1' contain two *disjoint* clusters S_1 and S_2, respectively, each with $n'/3$ points (i.e., $|S_1| = |S_2| = n'/3 = k + 1$), and the distance between any two points in different clusters is larger than $(7r)/8$. Let $S' = (S \setminus S_1) \setminus S_2$; then $|S'| = k - 1$. We disregard the fake points Q and Q' from now on. Any point $p' \in S'$ is outside or on the boundary of $\mathcal{C}$, and thus p' is at a distance greater than $r - r/8 = (7r)/8$ from any point of S_1. Similarly, p' is outside or on the boundary of R_1', and thus p' is at a distance greater than $(5r)/8 - r/8 = r/2$ from any point of S_2. Note that $diam(S_1) < (r/8) \cdot 2 = r/4$, and similarly $daim(S_2) < r/4$.

Without loss of generality, assume that $diam(S_1) < diam(S_2)$. To avoid having any pair of 2-neighbors both in S_1, the Hamiltonian path starts from a point in S_1, and visits two points outside S_1 (one in S_2 and one in S'), then repeats the process. After repeating the process $(k-1)$ times, S_1 and S_2 both have two points left over, and S' is empty. The only possible way to avoid having any pair of 2-neighbors both in S_1 is to visit the kth point of S_1, then the last two points of S_2, and finally the last point of S_1. This means that there must be a pair of points both in S_2 that are 2-neighbors, i.e., $diam(S_2)$ is an upper bound for d^*, the max-min 2-neighbor distance of an optimal path. We can achieve this upper bound by constructing the path $(S_1S_2S')^{k-1}S_1S_2S_2S_1$, where each occurrence of S_1 means visiting a point in S_1, and similarly for the occurrences of S_2 and S', and the last two points in S_2 to be visited are the pair in S_2 defining $diam(S_2)$. Clearly, all other 2-neighbors in this path have distances larger than $r/2$, which is larger than $diam(S_2)$, and the minimum 2-neighbor distance is $diam(S_2)$, the upper bound of d^*. Therefore $d^* = diam(S_2)$ and the path is optimal. ∎

This completes the proof of Theorem A.4. ∎

This completes the proof of Theorem 4.14.

Appendix B. Other exact algorithms for points on a line or a circle.

B.1. Cycle version: Points on a circle. We define an *ℓ-interval* as before, and the *arc* of an ℓ-interval I to be the arc containing the ℓ nodes of I, with endpoints being the endpoint nodes of I. Also, we define the *length* of I to be the length of the longest edge among all edges joining two nodes of I. Observe that in the case of points on a line, the edge joining the two endpoints of I always gives the length of I; this is no longer true for the case of points on a circle—unless the angle subtended by the arc of I is no more than π.

LEMMA B.1. *Let S be a set of n nodes lying on a circle, and d^* be the length of the shortest edge of an optimal Hamiltonian cycle on S. If $n = 2k+1$ is odd, then for any $(k+1)$-interval $I = \{u, \ldots, v\}$ whose arc subtends an angle larger than π, the length of edge (u, v) is larger than d^*. Moreover, the shortest and the second shortest $(k+1)$-intervals are both shorter than edge (u, v) and must each subtend an angle no more than π. Similarly, if $n = 2k$ is even, then for any k-interval $I = \{u, \ldots, v\}$ whose arc subtends an angle larger than π, the length of edge (u, v) is larger than d^*. Moreover, the shortest, the second shortest, and the third shortest k-intervals are all shorter than edge (u, v) and must each subtend an angle no more than π.*

Proof. Consider the case in which n is odd (see Figure B.1(a)). Observe that $I' = \{u, s, \ldots, t\}$ is also a $(k+1)$-interval, whose arc subtends an angle no more than π. This means that edge (u, t) is the longest edge with both endpoints in I'. By Corollary 3.3, the length of (u, t) is an upper bound for d^*. But since (u, v) is longer than (u, t), the length of edge (u, v) is larger than d^*. In fact, (u, v) is longer than *both* (u, t) and (s, v) by the same argument, where (s, v) defines the length of another $(k+1)$-interval $I'' = \{s, \ldots, t, v\}$ with subtended angle no more than π.

Now consider the case in which n is even (see Figure B.1(b)). Again, $I' = \{s, p, \ldots, t\}$, $I'' = \{p, \ldots, t, v\}$, and $I''' = \{u, s, \ldots, q\}$ are three k-intervals each of whose arcs subtends an angle no more than π, and edge (u, v) is longer than all three edges joining both endpoints of I', of I'', and of I'''. This means that the length of (u, v) is larger than the length of the third shortest k-interval. The lemma then follows from Corollary 6.4. ∎

We solve the problem by the following algorithm. When $n = 2k+1$, we apply the algorithm of section 6.1 to construct a Hamiltonian cycle C^*. When $n = 2k$, we first

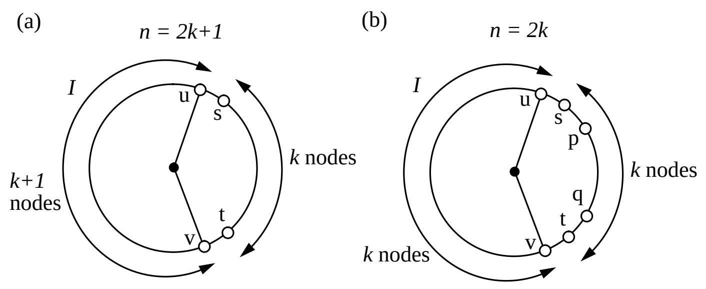


FIG. B.1. *Proof of Lemma* B.1: (a) *n is odd*; (b) *n is even.*

scan all n possible k-intervals to identify the shortest k-interval, I_1. By Lemma B.1, I_1 is found by taking the shortest edge among all edges joining the two endpoint nodes of all k-intervals. Then we label all nodes of S from 1 to $2k$ clockwise along the circle, with the nodes of I_1 labeled 1 to k. Finally, we use the new labels and apply the algorithm of Theorem 6.5 to construct a Hamiltonian cycle C^*.

We claim that the resulting C^* is optimal. First, the wrap-around effect is readily taken care of: when $n = 2k + 1$, each edge on C^* spans $k + 2$ nodes on one side and also spans $k + 1$ nodes on the other side; when $n = 2k$, each edge either spans $k + 1$ nodes on both sides (edges e_1 and e_2 given in the algorithm of section 6.2) or spans k nodes on one side and $k + 2$ nodes on the other side (the remaining edges). Secondly, let I be an $(k + 1)$-interval (resp., k-interval) when $n = 2k + 1$ (resp., $n = 2k$). The issue that an edge spanning all $k + 1$ nodes (resp., all k nodes) of I may not be the longest edge within I is resolved by Lemma B.1. Hence C^* is an optimal Hamiltonian cycle.

THEOREM B.2. *Let S be a set of n points lying on a circle and the distances are Euclidean. Then there exists an algorithm that solves the cycle version of the maximum scatter TSP on S exactly in optimal $O(n)$ time.*

B.2. Path version: Points on a line. Now, consider the path version of points on a line. As we did at the end of section 3, we add a *fake point* Q very far away from all points of S, say, at position ∞, so that it does not affect the max-min edge length, and solve the cycle problem. When we get the answer that is a cycle, we simply cut the cycle open at Q to get a path, and then remove Q. We call this method *fake-point algorithm* for future reference. We denote by C^* the intermediate cycle constructed, and P^* the final path obtained.

When $n = 2k$, adding Q makes the construction of C^* an easy case ($n' = n + 1$ is odd), for which we apply the algorithm of section 6.1. The upper bound for the path version (from Lemma 3.1: every set R of cardinality $\lceil \frac{n}{2} \rceil + 1 = k + 1$ must have an edge inside R) is easily achieved (the edges in the resulting path P^* are $(k + 1)$- or $(k + 2)$-edges), and hence P^* is optimal. When $n = 2k + 1$, adding Q makes the construction of C^* a more complicated case ($n' = 2k + 2$ is even). In this case, a tighter upper bound for an optimal path is needed. In fact, we can show the following lemma similar to Lemma 6.2.

LEMMA B.3. *Suppose $n = 2k + 1$, and let I_1 and I_2 be two* distinct $(k + 1)$-*intervals. Clearly, I_1 and I_2 must overlap. Then in any Hamiltonian path that avoids having any edge joining two nodes of I_1, there must exist an edge joining two nodes of I_2.*

Proof. Let $S_{in} = I_1 \cap I_2$ and $S_{out} = S \setminus (I_1 \cup I_2)$. If $|S_{in}| = \ell$ ($0 < \ell < k + 1$,

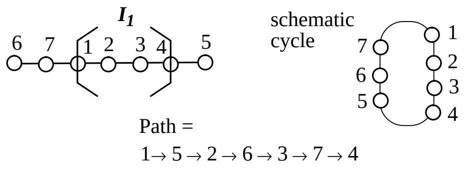


FIG. B.2. *Example of an optimal Hamiltonian path specified by the* direct algorithm *when n is odd* ($n = 7$).

since I_1 and I_2 overlap and also they are distinct), then $|I_1 \cup I_2| = 2k + 2 - \ell$, and thus $|S_{out}| = \ell - 1$. In a Hamiltonian path on S, to avoid having any edge joining two nodes both in I_1 or both in I_2, every node in S_{in} must connect only to nodes in S_{out}. Since $|S_{in}| = |S_{out}| + 1$, this means that the path must alternate between the nodes of S_{in} and the nodes of S_{out}, starting and ending at some nodes both in S_{in}, *without* visiting the nodes outside $S_{in} \cup S_{out}$ (such left-over nodes exist since $|S \setminus (S_{in} \cup S_{out})| = 2k + 1 - (\ell + \ell - 1) > 0$), a contradiction. □

This says that the length of the *second shortest* $(k + 1)$*-interval* gives an upper bound for d^*, the optimal objective value. We claim that using the fake-point algorithm, where we apply the algorithm of section 6.2 to construct cycle C^* on $n' = 2k+2$ points, achieves this upper bound. Recall from section 6.2 that each edge of C^* spans either $k' = k + 1$ and $k' + 2$ nodes on two sides of the schematic cycle (type 1 edge), or $k' + 1$ nodes on both sides (type 2 edge). When we remove Q to obtain P^*, each edge spans one less node on one side of the schematic cycle. The only possibility for an edge of P^* to span less than $k + 1$ nodes is when the k' nodes (on the schematic cycle) spanned by some type 1 edge include Q. But this means that these k' nodes are counted by wrapping around the line, and thus such an edge actually spans $k' + 2$ nodes on the line. Therefore P^* has each edge spanning at least $k + 1$ nodes (while avoiding both endpoints in the shortest $(k + 1)$-interval), and thus is optimal.

We can also use the following algorithm to solve the problem directly (we thus call the method *direct algorithm*):

When $n = 2k$**:**
Connect node p to node $(p + k)$, for all $p = 1, 2, \ldots, k$; also connect node $[(k + 1) + q]$ to node q, for all $q = 1, 2, \ldots, (k - 1)$.
When $n = 2k + 1$**:**
Identify the shortest $(k + 1)$-interval I_1. Relabel the nodes of S in a wrap-around fashion, with the nodes of I_1 labeled $1, \ldots, (k + 1)$. Put the nodes in a schematic cycle as in Figure B.2. Use the new labels to construct the following path: start visiting node 1, and repeatedly visit the $(k + 1)$-st node counted *counterclockwise* from the current node (which is counted as the first node), until reaching node $k + 1$ and stop. The resulting path is $1, (k + 2), 2, (k + 3), 3, \ldots, (k + 1)$ (see Figure B.2).

It is easy to see that when $n = 2k$, the constructed path has each edge spanning $k + 1$ or $k + 2$ nodes (in fact, the path is the same as the one constructed by the fake-point algorithm), and when $n = 2k + 1$, the constructed path has each edge spanning, respectively, $k + 1$ and $k + 2$ nodes on two sides of the schematic cycle while avoiding both endpoints in I_1. Therefore the path is optimal.

THEOREM B.4. *Let S be a set of n points lying on a line and the distances are Euclidean. Then there exists an algorithm that solves the path version of the maximum scatter TSP on S exactly in optimal $O(n)$ time.*

B.3. Path version: Points on a circle. Because of the wrap-around effect, the fake-point algorithm does not work: it is not possible to place Q so that Q is far away from *all* points of S. When $n = 2k + 1$, the above direct algorithm readily takes care of the wrap-around effect. Also, since the length of the second shortest $(k + 1)$-interval is (an upper bound for) d^*, the issue that an edge spanning all $k + 1$ nodes of a $(k + 1)$-interval I may not be the longest edge within I is resolved by Lemma B.1. Therefore we can solve the problem by scanning all n possible $(k + 1)$-intervals to identify the shortest one (whose length is given by the edge joining its two endpoints), and applying the direct algorithm. When $n = 2k$, the problem is more involved: all k edges $(1, k + 1), (2, k + 2), \ldots, (k, 2k)$ each span $k + 1$ nodes on *both* sides of the circle, and thus it is not possible to have a Hamiltonian path with each edge spanning at least $k+1$ nodes (as in the case of points on a line)—a tighter upper bound is needed. We do not explore the problem any further; however, if the points of S are *equally spaced*, then it is easy to see that the above direct algorithm gives an optimal Hamiltonian path.

COROLLARY B.5. *Let S be a set of n points lying on a circle and the distances are Euclidean. If n is an odd integer or the points are equally spaced, then there exists an algorithm that solves the path version of the maximum scatter TSP on S exactly in optimal $O(n)$ time.*

Acknowledgments. We thank Samir Khuller for stimulating discussions and comments, which led to Theorem 4.1 and an improvement on the running time of Theorem 3.5. We thank Fritz Scholz and Karel Zikan at Boeing for originally bringing the riveting problem to our attention and for discussions on the practical aspects of riveting in aircraft assembly. We thank Kresha Penavic for discussions on the applications of maximum scatter TSP in medical imaging. We also thank Vera Sacristan and Amitabh Varshney for useful discussions in the early stages of this research.

REFERENCES

[1] N. Alon, R. A. Duke, H. Lefmann, V. Rödl, and R. Yuster, *The algorithmic aspects of the regularity lemma*, in Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science (FOCS 92), Pittsburgh, PA, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 473–481.

[2] N. Alon, S. Rajagopalan, and S. Suri, *Long non-crossing configurations in the plane*, in Proceedings of the 9th Annual ACM Symposium on Computational Geometry, San Diego, CA, ACM Press, New York, 1993, pp. 257–263.

[3] E. M. Arkin, Y.-J. Chiang, J. S. B. Mitchell, S. S. Skiena, and T. Yang, *On the maximum scatter TSP*, in Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, 1997, pp. 211–220.

[4] A. I. Barvinok, *Two algorithmic results for the traveling salesman problem*, Math. Oper. Res., 21 (1996), pp. 65–84.

[5] A. Barvinok, D. S. Johnson, G. J. Woeginger, and R. Woodroofe, *The maximum traveling salesman problem under polyhedral norms*, in Proceedings of the Sixth International Conference on Integer Programming and Combinatorial Optimization, Lecture Notes in Comput. Sci. 1412, Springer-Verlag, New York, 1998, pp. 195–201.

[6] G. A. Dirac, *Some theorems on abstract graphs*, Proc. London Math. Soc., 2 (1952), pp. 69–81.

[7] S. Even and R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[8] S. P. Fekete, *Simplicity and hardness of the maximum traveling salesman problem under geometric distances*, in Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA 99), Baltimore, MD, ACM Press, New York and SIAM, Philadelphia, 1999, pp. 337–345.

[9] P. Hall, *On representatives of subsets*, J. London Math. Soc., 10 (1935), pp. 26–30.

[10] R. Hassin and S. Rubinstein, *An approximation algorithm for the maximum traveling salesman problem*, Inform. Process. Lett., 67 (1998), pp. 125–130.
[11] J. E. Hopcroft and R. M. Karp, *A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.
[12] S. Khuller, *personal communication*, 1996.
[13] J. Komlós, *personal communication*, 1996.
[14] J. Komlós, G. N. Sárközy, and E. Szemerédi, *On the square of a Hamiltonian cycle in dense graphs*, Random Structures Algorithms, 9 (1996), pp. 193–211.
[15] J. Komlós, G. N. Sárközy, and E. Szemerédi, *On the Pósa–Seymour conjecture*, J. Graph Theory, 29 (1998), pp. 167–176.
[16] S. R. Kosaraju, J. K. Park, and C. Stein, *Long tours and short superstrings*, in Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science (FOCS 94), Santa Fe, NM, IEEE Computer Society Press, Los Alamitos, CA, 1994.
[17] E. L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and D. B. Shmoys, eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, 1985.
[18] R. G. Parker and R. L. Rardin, *Guaranteed performance heuristics for the bottleneck traveling salesman problem*, Oper. Res. Lett., 2 (1984), pp. 269–272.
[19] K. Penavic, *Optimal Firing Sequences for CAT Scans*, manuscript, Department of Applied Mathematics, SUNY Stony Brook, Stony Brook, NY, 1994.
[20] K. Penavic, *private communication*.
[21] F. Scholz, *Coordination Hole Tolerance Stacking*, Technical Report BCSTECH-93-048, Boeing Computer Services, November, 1993.
[22] F. Scholz, *Tolerance Stack Analysis Methods*, Technical Report BCSTECH-95-030, Boeing Computer Services, December, 1995.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 545–570

THE PARAMETRIZED COMPLEXITY OF SOME FUNDAMENTAL PROBLEMS IN CODING THEORY*

ROD G. DOWNEY†, MICHAEL R. FELLOWS‡, ALEXANDER VARDY§, AND GEOFF WHITTLE†

Abstract. The parametrized complexity of a number of fundamental problems in the theory of linear codes and integer lattices is explored. Concerning codes, the main results are that MAXIMUM-LIKELIHOOD DECODING and WEIGHT DISTRIBUTION are hard for the parametrized complexity class $W[1]$. The NP-completeness of these two problems was established by Berlekamp, McEliece, and van Tilborg in 1978 using a reduction from THREE-DIMENSIONAL MATCHING. On the other hand, our proof of hardness for $W[1]$ is based on a parametric polynomial-time transformation from PERFECT CODE in graphs. An immediate consequence of our results is that bounded-distance decoding is likely to be hard for linear codes. Concerning lattices, we address the THETA SERIES problem of determining for an integer lattice Λ and a positive integer k whether there is a vector $x \in \Lambda$ of Euclidean norm k. We prove here for the first time that THETA SERIES is NP-complete and show that it is also hard for $W[1]$. Furthermore, we prove that the NEAREST VECTOR problem for integer lattices is hard for $W[1]$. These problems are the counterparts of WEIGHT DISTRIBUTION and MAXIMUM-LIKELIHOOD DECODING for lattices. Relations between all these problems and combinatorial problems in graphs are discussed.

Key words. parametrized complexity, linear codes, decoding, codes in graphs, integer lattices

AMS subject classifications. 68Q25, 68R10, 94B05, 05C85, 94B35

PII. S0097539797323571

1. Introduction.

Our main objective in this paper is to explore the parametrized complexity of certain fundamental computational problems in the theories of linear codes and integer lattices. There is a natural close relationship between computational problems in these areas. We prove one main combinatorial transformation, which we then use to show hardness for problems in both domains.

There has been a substantial amount of work on the complexity of the problems considered here. Although many of these problems are naturally parametrized, all the prior work was in the framework of NP-completeness. The following three problems, considered by Berlekamp, McEliece, and van Tilborg [5] in 1978, are of importance in the theory of linear codes:

Problem: MAXIMUM-LIKELIHOOD DECODING
Instance: A binary $m \times n$ matrix H, a target vector $s \in \mathbb{F}_2^m$, and an integer $k > 0$.
Question: Is there a set of at most k columns of H that sum to s?
Parameter: k
Problem: WEIGHT DISTRIBUTION
Instance: A binary $m \times n$ matrix H and an integer $k > 0$.
Question: Is there a set of k columns of H that sum to the all-zero vector?
Parameter: k

*Received by the editors June 26, 1997; accepted for publication (in revised form) June 1, 1998; published electronically November 10, 1999.
http://www.siam.org/journals/sicomp/29-2/32357.html

†Department of Mathematics, Victoria University, Private Bag 600, Wellington, New Zealand (downey@math.vuw.ac.nz, geoff.whittle@math.vuw.ac.nz).

‡Department of Computer Science, University of Victoria, Victoria, BC V8W 3P6, Canada (mfellows@csr.uvic.ca).

§Department of Electrical Engineering, Department of Computer Science, University of California, San Diego, 9500 Gilman Drive, MC 0407, La Jolla, CA, 92093 (vardy@ece.ucsd.edu). The research of this author was supported by the Packard Foundation and by the National Science Foundation.

Problem: MINIMUM DISTANCE
Instance: A binary $m \times n$ matrix H and an integer $k > 0$.
Question: Is there a nonempty set of at most k columns of H that sum to the all-zero vector?
Parameter: k

Notice that the difference between the definitions of the MINIMUM DISTANCE and WEIGHT DISTRIBUTION problems is very slight. WEIGHT DISTRIBUTION requires exactly k columns in a solution, while MINIMUM DISTANCE requires at most k columns in a solution.

Berlekamp, McEliece, and van Tilborg [5] proved that MAXIMUM-LIKELIHOOD DECODING and WEIGHT DISTRIBUTION are NP-complete by means of a reduction from THREE-DIMENSIONAL MATCHING. They conjectured that MINIMUM DISTANCE is also NP-complete, and Vardy [32] recently proved this conjecture using a nonparametric reduction from MAXIMUM-LIKELIHOOD DECODING. Since THREE-DIMENSIONAL MATCHING is fixed-parameter tractable, these earlier results do not allow us to conclude anything about the parametrized complexity of the three problems.

Over the past few years, it has been shown that many NP-complete problems are fixed-parameter tractable. For example, VERTEX COVER, a well-known NP-complete problem [22, p. 53] which asks whether a graph G on n vertices has a vertex cover of size at most k, can be solved [3] in time $O(kn + ({}^4\!/_3)^k k^2)$. Loosely speaking, the parametrized complexity hierarchy

$$\mathrm{FPT} = W[0] \;\subseteq\; W[1] \;\subseteq\; W[2] \;\subseteq\; \cdots \subseteq W[\mathrm{P}] \subseteq \cdots \subseteq XP$$

introduced by Downey and Fellows [14, 15] distinguishes between those problems that are fixed-parameter tractable and those that are not. For more details on the W hierarchy of parametrized complexity, see section 4, the Appendix, and references therein, in particular [18].

One of our main results in this paper is a proof that MAXIMUM-LIKELIHOOD DECODING and WEIGHT DISTRIBUTION are hard for the parametrized complexity class $W[1]$. We also show that both problems belong to the class $W[2]$. The proof of $W[1]$-hardness is based on a parametric polynomial-time reduction from the PERFECT CODE problem for graphs. Such a proof establishes both $W[1]$-hardness and NP-completeness at the same time. Furthermore, an immediate consequence of this result is that bounded-distance decoding is likely to be hard for binary linear codes, unless the parametrized complexity hierarchy collapses with $W[1] = \mathrm{FPT}$.

Three closely related problems in the theory of integer lattices are natural counterparts of the three problems concerning linear codes, discussed above. These problems are defined as follows:

Problem: NEAREST VECTOR
Instance: A basis $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{Z}^n$ for a lattice Λ, a target vector $s \in \mathbb{Z}^n$, and an integer $k > 0$.
Question: Is there a vector $x \in \Lambda$ such that $\|x - s\|^2 \le k$?
Parameter: k

Problem: THETA SERIES
Instance: A basis $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{Z}^n$ for a lattice Λ and an integer $k > 0$.
Question: Is there a vector $x \in \Lambda$ such that $\|x\|^2 = k$?
Parameter: k

Problem: Shortest Vector
Instance: A basis $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{Z}^n$ for a lattice Λ and an integer $k > 0$.
Question: Is there a nonzero vector $x \in \Lambda$ such that $\|x\|^2 \leq k$?
Parameter: k

Here, a lattice Λ is the set of all linear combinations with integer coefficients of the elements of its basis X and $\|\cdot\|$ denotes the Euclidean norm. Again, notice that the difference between the definitions of Shortest Vector and Theta Series is very slight. Theta Series requires $\|x\|^2 = k$ in a solution, while Shortest Vector requires only the inequality $\|x\|^2 \leq k$.

Peter van Emde Boas [30] proved in 1980 that Nearest Vector is NP-complete and conjectured that Shortest Vector is also NP-complete. There has been a considerable amount of work devoted to the proof of this conjecture; see [2] and [31] for a discussion. Ajtai [1] has recently proved that the Shortest Vector problem is hard for NP under randomized reductions. Akin to the situation with linear codes, nothing is currently known regarding the parametrized complexity of the three problems discussed above.

Herein, we prove for the first time that Theta Series is NP-hard. Moreover, since our reduction is parametric, it shows that Theta Series is hard for the parametrized complexity class $W[1]$. Along similar lines, we prove that the Nearest Vector problem is also hard for $W[1]$.

Our results are based on a powerful combinatorial transformation that uses many of the ideas employed in the proofs of the main theorems in [14, 15]. We feel that one of the most interesting aspects of our work is a demonstration of the potential utility of parametric methods and perspectives in addressing issues in "classical" complexity theory. We will assume that the reader has some familiarity with the parametrized complexity framework such as can be found in [14, 15, 16, 18], for example. For the benefit of readers that do not have this background, some discussion and essential definitions are presented in section 4 and the appendix.

In the interests of readability, we defer the proof of our main combinatorial transformation to section 3. In the next section, we prove the main results using this transformation. In section 4 we present an outline of the proof of membership in $W[2]$ for several of the problems considered in this paper. We also discuss the remaining open problems and speculate on whether the techniques used herein might be adapted to provide a proof of $W[1]$-hardness for Minimum Distance or of NP-hardness for the Shortest Vector.

2. The main results through red/blue graphs.

We start with some notation and terminology. Let $G = (V, E)$ be a graph. We say that two distinct vertices $u, v \in V$ are *neighbors* if they are adjacent in G, namely, if $(u, v) \in E$. We will assume throughout that $G = (V, E)$ is loopless so that a vertex $v \in V$ is never its own neighbor. A set of vertices $V' \subseteq V$ is said to be a *perfect code* in G if every vertex of $V \setminus V'$ has a *unique* neighbor in V', while the vertices of V' itself do not have neighbors in V'. The starting point for our transformations is the parametrized problem of determining the existence of a k-element perfect code in a graph.

Problem: Perfect Code
Instance: A graph $G = (V, E)$ and an integer $k > 0$.
Question: Is there a k-element perfect code in G?
Parameter: k

This problem was shown to be hard for $W[1]$ in [15]. Kratochvíl [25, 24] was the first to prove, several years earlier, that this problem is NP-complete.

Our general approach in what follows is based on constructing and manipulating linear codes, and other sets of vectors, using bipartite graphs. Let $G = (\mathcal{R}, \mathcal{B}, E)$ be a bipartite graph with the partition of the vertices into the red set $\mathcal{R}$ and the blue set $\mathcal{B}$. We make the following definitions concerning special sets of red vertices in G.

Definition 2.1. *Suppose that $G = (\mathcal{R}, \mathcal{B}, E)$ is a red/blue bipartite graph, and let $R \subseteq \mathcal{R}$ be a nonempty set of red vertices. We say that R is*

- *a* dominating set *if every vertex in $\mathcal{B}$ has at least one neighbor in R,*
- *a* perfect code *if every vertex in $\mathcal{B}$ has a unique neighbor in R,*
- *an* odd set *if every vertex in $\mathcal{B}$ has an odd number of neighbors in R,*
- *an* even set *if every vertex in $\mathcal{B}$ has an even number of neighbors in R.*

Notice that what we define to be a perfect code is not the same for red/blue bipartite graphs and for general (uncolored) graphs. Similarly, our definition of a dominating set in a red/blue graph does not coincide with the conventional definition of dominating sets in general graphs. However, it will always be clear from the context which definition applies in each case.

We can now state the main problems concerning red/blue graphs that we consider in this paper.

Problem: Even Set
Instance: A red/blue graph $G = (\mathcal{R}, \mathcal{B}, E)$ and an integer $k > 0$.
Question: Is there a nonempty set of at most k vertices $R \subseteq \mathcal{R}$ that is an even set in G?
Parameter: k

Problem: Exact Even Set
Instance: A red/blue graph $G = (\mathcal{R}, \mathcal{B}, E)$ and an integer $k > 0$.
Question: Is there a set of k vertices $R \subseteq \mathcal{R}$ that is an even set in G?
Parameter: k

Problem: Odd Set
Instance: A red/blue graph $G = (\mathcal{R}, \mathcal{B}, E)$ and an integer $k > 0$.
Question: Is there a set of at most k vertices $R \subseteq \mathcal{R}$ that is an odd set in G?
Parameter: k

Problem: Exact Odd Set
Instance: A red/blue graph $G = (\mathcal{R}, \mathcal{B}, E)$ and an integer $k > 0$.
Question: Is there a set of k vertices $R \subseteq \mathcal{R}$ that is an odd set in G?
Parameter: k

We shall see shortly that all these problems are NP-complete. We will also prove that Exact Even Set, Odd Set, and Exact Odd Set are hard for $W[1]$. The following theorem will serve as the main combinatorial engine in our proof.

Theorem 2.2. *Let G be a graph on n vertices, and let k be a positive integer. In time polynomial in n and k we can produce a red/blue bipartite graph G' and a positive integer k' such that*

P1. *every dominating set in G' has size at least k',*
P2. *every dominating set in G' of size k' is a perfect code in G',*
P3. *there is a perfect code of size k in G if and only if there is a perfect code of size k' in G'.*

Notice that the red/blue graph G' encodes the information about the existence of a k-element perfect code in G. However, while the graph G is completely arbitrary, the red/blue graph G' has a substantial amount of useful structure, expressed by the properties P1 and P2.

The theorem itself is established in the next section by means of a somewhat complicated graph-theoretic transformation that has a general architecture similar to the one employed in proving the main theorems of [14] and [15]. In what follows, we will use Theorem 2.2 to yield significant results that illustrate the applicability of our graph-theoretic approach to parametrized problems concerning linear codes and integer lattices.

Parametrized complexity of problems concerning linear codes. First, note that we have the following merely by observing that in a red/blue bipartite graph, by definition a perfect code is an odd set and an odd set is a dominating set.

THEOREM 2.3. ODD SET *and* EXACT ODD SET *are hard for* $W[1]$ *and* NP-*complete.*

Proof. Theorem 2.2 immediately implies a polynomial-time parametrized transformation from PERFECT CODE, which is a $W[1]$-hard and NP-complete problem, to ODD SET and to EXACT ODD SET. Given an instance G and k of PERFECT CODE, we construct G' and k' as in Theorem 2.2. It follows from properties P1–P3 that G' contains an odd set of size at most k' if and only if G has a k-element perfect code and the size of this odd set is, in fact, exactly k'. □

We now observe that ODD SET is very similar to MAXIMUM-LIKELIHOOD DECODING. This immediately leads to the following.

THEOREM 2.4. MAXIMUM-LIKELIHOOD DECODING *is hard for* $W[1]$.

Proof. Given an instance $G = (\mathcal{R}, \mathcal{B}, E)$ and k of ODD SET, we construct an instance of MAXIMUM-LIKELIHOOD DECODING as follows. The binary $m \times n$ matrix $H = [h_{ij}]$ is the adjacency matrix of G, whose columns are indicators of the neighborhoods of vertices in $\mathcal{R}$. That is, without loss of generality (w.l.o.g.) we let $\mathcal{R} = \{1, 2, \dots, n\}$ and $\mathcal{B} = \{1, 2, \dots, m\}$ and then set $h_{ij} = 1$ if and only if $i \in \mathcal{B}$ is adjacent to $j \in \mathcal{R}$ in G. Thus G is a Tanner graph for the binary linear code having H as its parity-check matrix (cf. [29, 26]). We set the target vector s equal to the all-one vector $\mathbf{1}$ of length $m = |\mathcal{B}|$. It is easy to see that a set of k columns of H sums to $s = \mathbf{1}$ if and only if the corresponding k vertices of $\mathcal{R}$ constitute an odd set in G. □

We now employ Theorem 2.2 in a slightly more elaborate way to establish the following.

THEOREM 2.5. EXACT EVEN SET *is hard for* $W[1]$ *and* NP-*complete.*

Proof. We again reduce from PERFECT CODE. Given an instance G and k of PERFECT CODE, we first construct $G' = (\mathcal{R}, \mathcal{B}, E)$ and k' as in Theorem 2.2. Next, we let $G_1 = (\mathcal{R}_1, \mathcal{B}_1, E_1)$ and $G_2 = (\mathcal{R}_2, \mathcal{B}_2, E_2)$ denote two identical replicas of G'. We can combine G_1 and G_2 into a single red/blue graph H by creating a new red vertex z and connecting it to all the vertices in $\mathcal{B}_1$ and $\mathcal{B}_2$. Finally, we obtain a red/blue graph H^* by adjoining to H a set of $|\mathcal{R}|$ blue vertices $\mathcal{B}^*$, one for each vertex of $\mathcal{R}$, and connecting them as follows: every $\beta \in \mathcal{B}^*$ is adjacent to one vertex in $\mathcal{R}_1$ and one vertex in $\mathcal{R}_2$, which correspond to the same vertex of $\mathcal{R}$, and every such pair of vertices in $\mathcal{R}_1$ and $\mathcal{R}_2$ is connected through some vertex of $\mathcal{B}^*$. The construction of H^* from G' is illustrated in Figure 2.1. The instance of EXACT EVEN SET is given by H^* and $2k' + 1$.

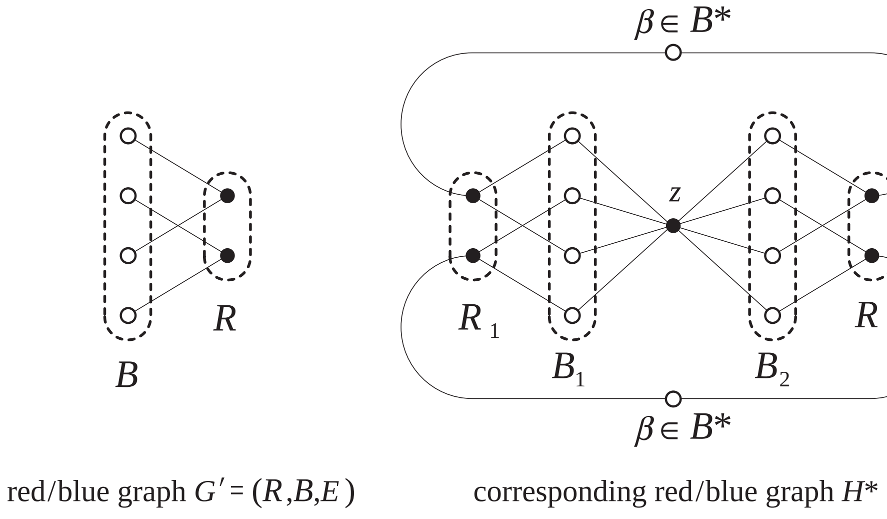

B
R
red/blue graph $G' = (R,B,E)$
$\beta \in B^*$
z
R_1
B_1
B_2
R_2
$\beta \in B^*$
corresponding red/blue graph H^*

FIG. 2.1. *Construction of H^* from G'.*

Now suppose that G has a k-element perfect code. Then by property P3 of Theorem 2.2, the red/blue graph G' has a k'-element perfect code $R \subseteq \mathcal{R}$. It is straightforward to verify that the $2k'$ vertices corresponding to R in $\mathcal{R}_1$ and $\mathcal{R}_2$, together with the vertex z, constitute an even set of size $2k' + 1$ in H^*. Indeed, by construction every blue vertex of $\mathcal{B}_1 \cup \mathcal{B}_2$ is adjacent to z and to exactly one vertex in the replica of R, either in $\mathcal{R}_1$ or in $\mathcal{R}_2$. Every blue vertex of $\mathcal{B}^*$ is either adjacent to both replicas of some vertex in R or to none at all.

In the other direction, suppose that S is an even set of size $2k' + 1$ in H^*. If S contains a vertex $\rho \in \mathcal{R}_1$, then it must also contain the corresponding vertex of $\mathcal{R}_2$, since otherwise the vertex of $\mathcal{B}^*$ adjacent to ρ will have exactly one neighbor in S. It follows that $|S \cap \mathcal{R}_1| = |S \cap \mathcal{R}_2|$ and the vertices of S are paired in this way. Since the size of S is odd, we conclude that S must contain the vertex z. This further implies that $S \cap \mathcal{R}_1$ is a dominating set in G_1, as otherwise there is a vertex $\beta \in \mathcal{B}_1$ which has the single neighbor z in S. Since $|S| = 2k' + 1$ and $|S \cap \mathcal{R}_1| = |S \cap \mathcal{R}_2|$, the size of this dominating set is exactly k'. By property P2 of Theorem 2.2, this dominating set is a perfect code in $G_1 = G'$, and property P3 implies that G has a perfect code of size k. $\square$

It is easy to see that EXACT EVEN SET and WEIGHT DISTRIBUTION are essentially different ways to formulate one and the same problem. Thus we have the following theorem.

THEOREM 2.6. WEIGHT DISTRIBUTION *is hard for* $W[1]$.

Proof. This follows directly from Theorem 2.5 by taking the matrix H in WEIGHT DISTRIBUTION to be the adjacency matrix for the graph $G = (\mathcal{R}, \mathcal{B}, E)$ in EXACT EVEN SET in the same way as it was done in the proof of Theorem 2.4. $\square$

Complexity of bounded-distance decoding. The fact that MAXIMUM-LIKELIHOOD DECODING is hard for $W[1]$, as established in Theorem 2.4, implies hardness for bounded-distance decoding of binary linear codes in a certain sense. We now explain this implication.

Maximum-likelihood decoding is a nearest neighbor search in the space $\mathbb{F}_2^n$ endowed with the Hamming distance $d(x, y)$ = number of positions where $x \in \mathbb{F}_2^n$ and

$y \in \mathbb{F}_2^n$ differ. Let H be an $m \times n$ binary matrix, and let $\mathbb{C}$ be the binary linear code defined by H; that is, $\mathbb{C} = \{x \in \mathbb{F}_2^n \;:\; Hx^t = \mathbf{0}\}$. Then for all $y \in \mathbb{F}_2^n$, a *maximum-likelihood decoder* for $\mathbb{C}$ always finds the closest codeword; that is, $x \in \mathbb{C}$ such that $d(x, y)$ is the minimum possible. If $s = Hy^t$, this is equivalent to finding the smallest set of columns of H that sums to s; hence the corresponding decision problem is precisely Maximum-Likelihood Decoding.

While the complexity of maximum-likelihood decoding has been thoroughly studied [2, 4, 5, 9, 28], almost nothing is presently known regarding the complexity of bounded-distance decoding, even though most of the decoders used in practice are bounded-distance decoders. A decoder is said to be *bounded-distance* if there exists a constant $t > 0$ such that for all $y \in \mathbb{F}_2^n$, the decoder always finds the closest codeword $x \in \mathbb{C}$, provided $d(x, y) \leq t$. Formally, for each positive integer t, we define the following problem.

Problem: BDD(t)
Instance: A binary $m \times n$ matrix H, a target vector $s \in \mathbb{F}_2^m$, and an integer $k \leq t$.
Question: Is there a set of at most k columns of H that sum to s?

Notice that BDD(t) is trivially in P for all t, since it can be solved in time $O(n^t)$ by simply computing Hx^t for every vector x in a Hamming sphere of radius t. Hence, the complexity of bounded-distance decoding has to be studied in a different framework. In particular, we would like to have an algorithm that solves BDD(t) in time $O(n^c)$, where c is a constant independent of t. That is, the multiplicative constant in $O(\cdot)$ may depend on t, but not the exponent.

The following corollary to $W[1]$-hardness of Maximum-Likelihood Decoding shows that such an algorithm does not exist unless the W complexity hierarchy collapses with $W[1] = \text{FPT}$.

Theorem 2.7. *Suppose that* $W[1] \neq \text{FPT}$. *Then for any positive constant* c, *there exists an integer* t_0 *such that* BDD(t) *is not solvable in time* $O(n^c)$ *for all* $t \geq t_0$ *even if the multiplicative constant in* $O(n^c)$ *may depend on* t.

Proof. First observe that the claim of the theorem is equivalent to the following: for any positive constant c, there exists an integer t_0 such that BDD(t_0) is not solvable in time $O(n^c)$. This is so because if $t \geq t_0$, then the set of possible instances of BDD(t_0) is a subset of the set of possible instances of BDD(t). Hence, if BDD(t_0) cannot be solved in time $O(n^c)$, then neither can BDD(t) for all $t \geq t_0$.

Now, assume to the contrary that for some c we can solve BDD(t) in time $O(n^c)$ for all t. Given an instance of Maximum-Likelihood Decoding, we set $t = k$ and query an oracle for BDD(t), which provides an answer to the question of Maximum-Likelihood Decoding in time $O(n^c)$. The constant in $O(n^c)$ may depend on $t = k$; let us denote this constant by a_k. Setting $f(k) = a_k$, we see that Maximum-Likelihood Decoding is solvable in time $f(k)n^c$. Hence, it is fixed-parameter tractable, which in view of Theorem 2.4 is possible only if $W[1] = \text{FPT}$. ☐

Parametrized complexity of problems concerning integer lattices. The combinatorial transformation in Theorem 2.2 can also be used to show both NP-completeness and $W[1]$-hardness for Nearest Vector and Theta Series. We start with the Nearest Vector problem.

Theorem 2.8. Nearest Vector *is hard for* $W[1]$.

Proof. Given an instance G and k of Perfect Code, we again construct $G' = (\mathcal{R}, \mathcal{B}, E)$ and k' as in Theorem 2.2. W.l.o.g., let $\mathcal{R} = \{1, 2, \ldots, r\}$ and

$\mathcal{B} = \{1, 2, \ldots, b\}$, and let H be the $r \times b$ blue/red adjacency matrix of G', which is the transpose of the matrix constructed in Theorem 2.4. That is, the *rows* of H are now indicators of the neighborhoods of the vertices in $\mathcal{R}$. If c is a real constant and $A = [a_{ij}]$ is an integer matrix, we let $cA = [ca_{ij}]$ denote the matrix obtained by multiplying each entry of A by c. We choose c to be a large integer, so that $c > k'$, and construct an instance of NEAREST VECTOR as follows. First, we define a $(b+r) \times (b+r)$ integer matrix

$$M = \left[\begin{array}{c|c} cH & I_r \\ \hline 2cI_b & \mathbf{0} \end{array}\right], \tag{2.1}$$

where I_r is the $r \times r$ identity matrix and $\mathbf{0}$ stands for the $b \times r$ all-zero matrix. It is easy to see that M has $n = r + b$ linearly independent rows, which constitute a basis for a sublattice Λ of $\mathbb{Z}^n$. We take the target vector as $s = (c, c, \ldots, c,\ 0, 0, \ldots, 0) \in \mathbb{Z}^n$ so that the first b entries of s are equal to c while the last r entries are equal to zero. Then Λ, s, and k' is the instance of NEAREST VECTOR to which the instance G and k of PERFECT CODE is transformed.

If there is a perfect code of size k in G, then by property P3 of Theorem 2.2 there is a perfect code $R \subseteq \mathcal{R}$ of size k' in G'. For convenience, we let M' denote the $r \times (b+r)$ matrix consisting of the first r rows of M. Thus each row of M' is naturally indexed by a unique vertex of $\mathcal{R}$. Let $x = (x_1, x_2, \ldots, x_n) \in \Lambda$ be the sum of those k' rows of M' that are indexed by the vertices of the perfect code R. Then it is easy to see that $x_i = c$ in the first b positions, and $\|x - s\|^2 = k'$.

In the other direction, suppose that there is a vector $x = (x_1, x_2, \ldots, x_n) \in \Lambda$ with $\|x - s\|^2 \le k'$, and denote $y = (y_1, y_2, \ldots, y_n) = x - s$. First observe that $y_i \equiv 0 \bmod c$ for $i = 1, 2, \ldots, b$ by the construction of M and s. Since $c > k'$ while $\|y\|^2 \le k'$, it follows that $y_i = 0$ in the first b positions. This further implies that $x_i = c$ for $i = 1, 2, \ldots, b$. Now let R be the subset of $\mathcal{R}$ consisting of all the vertices that index those rows of M' that are included in the linear combination comprising x (with a nonzero coefficient). We claim that R is a dominating set in G'. Indeed, if some vertex $i \in \mathcal{B}$ does not have a neighbor in R, then $x_i \equiv 0 \bmod 2c$ in the corresponding position, contrary to the fact that $x_i = c$, which we have already established. Hence, by property P1 of Theorem 2.2, we have that $|R| \ge k'$. Together with $\|x - s\|^2 \le k'$ this implies that $\|x - s\|^2 = |R| = k'$, and R is a perfect code in G' by property P2 of Theorem 2.2. Property P3 of Theorem 2.2 now implies that G has a perfect code of size k. $\square$

By modifying the argument of Theorem 2.8 only slightly, we can prove that THETA SERIES is also hard for $W[1]$. The idea is to incorporate the target vector s into the generator matrix M for Λ. The same argument also shows, for the first time, that THETA SERIES is NP-complete.

THEOREM 2.9. THETA SERIES *is hard for* $W[1]$ *and* NP-*complete.*

Proof. We proceed as in the proof of Theorem 2.8, except that we now choose c so that $c > 4k' + 1$ and replace the matrix in (2.1) by the slightly more elaborate $(b+r+1) \times (b+r+1)$ matrix

$$M = \left[\begin{array}{c|c|c} cH & \mathbf{0} & 2I_r \\ \hline c^2 I_b & \mathbf{0} & \mathbf{0} \\ \hline c\,c \cdots c & 1 & 0\,0 \cdots 0 \end{array}\right], \tag{2.2}$$

where $\mathbf{0}$ is used to denote both the all-zero column and the $b \times r$ all-zero matrix. We again think of the $n = r + b + 1$ rows of M as a basis for a sublattice Λ of $\mathbb{Z}^n$. Thus an instance G and k of Perfect Code is transformed by Theorem 2.2 into the red/blue graph $G' = (\mathcal{R}, \mathcal{B}, E)$ and k', which is further transformed into the instance Λ and $4k' + 1$ of Theta Series.

Suppose that G contains a k-element perfect code, and let $R \subseteq \mathcal{R}$ be the corresponding k'-element perfect code in G'. We again let M' denote the $r \times (b + r + 1)$ submatrix of M consisting of the first r rows that are naturally indexed by the vertices of $\mathcal{R}$. Let $x = (x_1, x_2, \ldots, x_n) \in \Lambda$ be the following linear combination of $k' + 1$ rows of M: k' rows of M' indexed by the k' vertices of the perfect code R with coefficient $+1$ and the last row of M with coefficient -1. Then it is easy to see that $x_i = 0$ in the first b positions and $\|x\|^2 = 4k' + 1$.

In the other direction, suppose that there exists $x = (x_1, x_2, \ldots, x_n) \in \Lambda$ with $\|x\|^2 = 4k' + 1$. Write $x = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n$, where $a_1, a_2, \ldots, a_n$ are integers, not all zero, and $v_1, v_2, \ldots, v_n$ are the rows of M, listed in a top-to-bottom order. First, we again observe that $x_i \equiv 0 \bmod c$ for $i = 1, 2, \ldots, b$ by the construction of M. Since $c > 4k' + 1$, it follows that $x_i = 0$ in the first b positions. Further observe that $x_i \equiv 0 \bmod 2$ in the last r positions. Together with $\|x\|^2 \equiv 1 \bmod 4$, this implies that $x_{b+1} = a_n$ must be nonzero. Notice that

$$a_n^2 \; = \; x_{b+1}^2 \; \le \; \|x\|^2 \; = \; 4k' + 1 \; < \; c. \tag{2.3}$$

As in the proof of Theorem 2.8, let R be the subset of $\mathcal{R}$ consisting of the vertices that index those rows of M' which have a nonzero coefficient in the linear combination comprising x. We again claim that R is a dominating set in G'. Otherwise, let $i \in \mathcal{B}$ be a vertex which does not have a neighbor in R, and let $y = x - a_n v_n$. We have already shown that $x_i = 0$. Hence $y_i = -a_n c \neq 0$ and $|y_i| < c^2$ in view of (2.3). This is a contradiction, since if i does not have neighbors in R, then $y_i \equiv 0 \bmod c^2$ by the construction of M. From this point on, the proof is exactly the same as in Theorem 2.8. Referring to Theorem 2.2, we conclude that $|R| \ge k'$, which together with $\|x\|^2 = 4k' + 1$ implies that $|R| = k'$ and R is necessarily a perfect code in G'. This, in turn, further implies the existence of a k-element perfect code in G. ∎

As a final remark in this section, we observe that a theorem of Cai and Chen [11] states that if the optimization problem naturally associated to an integer-parameter problem has a fully polynomial-time approximation scheme (see [22] for an exposition of this concept), then the corresponding parametrized problem is fixed-parameter tractable. We can draw from this the following corollary.

Corollary 2.10. *There is no fully polynomial-time approximation scheme for any of the problems discussed in this section, unless* $W[1] = \mathrm{FPT}$.

In the next section, we prove our principal combinatorial transformation (Theorem 2.2), which served us so well in this section.

3. The combinatorial engine.

Our proof of Theorem 2.1 has many similarities to the proofs of the main theorems of [14] and [15], to which the reader may wish to refer. Notice that an implicit assumption in Theorem 2.2 is that the constant k' may depend on k but not on n. This assumption is essential for all the parametrized transformations in the previous section. (For more on this, see the appendix.) We now restate Theorem 2.1, while specifying a precise value for k' in terms of k.

THEOREM 2.1. *Let $G = (V, E)$ be a graph on n vertices, and let k be a positive integer. In time polynomial in n and k we can produce a red/blue bipartite graph $G' = (\mathcal{R}, \mathcal{B}, E')$ and the positive integer*

$$k' = (2k+1) + 3\left(2k(k+1) + (k+1)^2\right)$$

such that:

P1. *Every dominating set in G' has size at least k';*
P2. *Every dominating set in G' of size k' is a perfect code in G';*
P3. *There is a perfect code of size k in G if and only if there is a perfect code of size k' in G'.*

We start the proof with some notation. In describing the construction of G' from G it is convenient to identify V with the set of integers $\{1, \dots, n\}$. An *interval* of vertices is defined to be a subset of V consisting of consecutive integers, for example, $\{3, 4, 5, 6\}$, and may be empty. Let $\mathcal{J}$ denote the set of all nonempty intervals having a size of at most $n - k$; that is,

$$\mathcal{J} = \{\ J\ :\ J \text{ is an interval in } V \text{ and } 1 \le |J| \le n - k\ \}.$$

For an interval $J \in \mathcal{J}$, define the *initial boundary* $\partial(J)$ of J to be the largest nonnegative integer strictly less than the smallest element of J. For example, $\partial(\{2, 3, 4\}) = 1$, $\partial(\{6\}) = 5$, $\partial(\{1, 2\}) = 0$. Define the *terminal boundary* $\partial'(J)$ to be the smallest positive integer strictly greater than the largest element of J. Thus $\partial'(\{2, 3, 4\}) = 5$, for example.

We will also need to refer to empty intervals, but we will still need to indicate where these empty intervals begin and end. For this purpose, we introduce the special symbols $\epsilon_0, \epsilon_1, \dots, \epsilon_n$ and extend the definitions of ∂ and ∂' as follows. For $u = 0, 1, \dots, n$, we define $\partial(\epsilon_u) = u$ and $\partial'(\epsilon_u) = u + 1$. Thus ϵ_u represents the "interval" of vertices in V that is empty but "located" between u and $u + 1$. We let $\mathcal{J}^*$ denote the set of nonempty intervals $\mathcal{J}$ augmented with these empty intervals; that is, $\mathcal{J}^* = \mathcal{J} \cup \{\epsilon_u\ :\ 0 \le u \le n\}$.

If $u, v \in V$ with $u < v$, we define the interval *between* u and v as $J(u, v) = \{u + 1, \dots, v - 1\}$, provided $v - u \ge 2$. If $v = u + 1$, then $J(u, v) = \epsilon_u$. Similarly, for $u \in V$ we define the interval *preceding* u to be $J(0, u) = \{1, \dots, u - 1\}$ if $u \ge 2$, and if $u = 1$, then $J(0, u) = J(0, 1) = \epsilon_0$. The interval *succeeding* $u \in V$ is defined as $J(u, \infty) = \{u + 1, \dots, n\}$ if $u < n$, and $J(n, \infty) = \epsilon_n$.

Let $[k]$ denote the set of integers $\{1, 2, \dots, k\}$, and let $[k]^*$ denote the set of integers $\{0, 1, \dots, k\}$. Part of our construction of $G' = (\mathcal{R}, \mathcal{B}, E')$ will be quantified over the set $[k]^* \times [k]^*$ of ordered pairs of elements of $[k]^*$, while other parts of the construction will be quantified over subsets of this set, namely, $[k] \times [k]^*$ and $[k]^* \times [k]$. This distinction between $[k]$ and $[k]^*$ is basically a technicality, which is needed to account for the boundary cases in the construction.

The red foundation. Our description of G' starts with five sets of red vertices $R_1, R_2, \dots, R_5$. We will refer to the vertices of $R_1, R_2, \dots, R_5$ as *basic* red vertices. All the blue vertices, as well as some additional red vertices, will be added to G' as the construction progresses. The five sets $R_1, R_2, \dots, R_5$ are employed in our construction to represent the structure of each possible choice of a k-element subset of the set of vertices of G. Each one of the five sets is, in a sense, a gadget designed to capture a different aspect of this structure. We now describe these sets along with their roles in the construction of G' and the notation used to refer to their vertices:

- Gadgets that indicate the k chosen vertices:
  $R_1 \;=\; \{\, a(i,u) \;:\; i \in [k],\; u \in V \,\}$.
- Gadgets that indicate the intervals between chosen vertices:
  $R_2 \;=\; \{\, b(i,J) \;:\; i \in [k]^*,\; J \in \mathcal{J}^* \,\}$.
- Gadgets that indicate pairs of intervals:
  $R_3 \;=\; \{\, c(i,i',J,J') \;:\; i,i' \in [k]^*,\; J,J' \in \mathcal{J}^* \,\}$.
- Gadgets that indicate choice/interval pairs:
  $R_4 \;=\; \{\, c'(i,i',u,J) \;:\; i \in [k],\; i' \in [k]^*,\; u \in V,\; J \in \mathcal{J}^* \,\}$.
- Gadgets that indicate interval/choice pairs:
  $R_5 \;=\; \{\, c''(i,i',J,u) \;:\; i \in [k]^*,\; i' \in [k],\; J \in \mathcal{J}^*,\; u \in V \,\}$.

It will be convenient to organize the basic red vertices into *blocks*. These blocks will constitute a partition of the basic red vertices, and there will be altogether

$$k'' \;=\; 1 + 2k + 2k(k+1) + (k+1)^2 \tag{3.1}$$

blocks. Specifically, we partition R_1 into k blocks, R_2 into $k+1$ blocks, R_3 into $(k+1)^2$ blocks, and R_4, R_5 into $k(k+1)$ blocks each. These blocks are defined as follows:

$$\mathcal{A}(i) = \{a(i,u) : i \in [k],\; u \in V\} \qquad \text{for } i = 1,2,\ldots,k,$$

$$\mathcal{B}(i) = \{b(i,J) : i \in [k]^*,\; J \in \mathcal{J}^*\} \qquad \text{for } i = 0,1,\ldots,k,$$

$$\mathcal{C}(i,i') = \{c(i,i',J,J') : J,J' \in \mathcal{J}^*\} \qquad \text{for } i \in [k]^* \text{ and } i' \in [k]^*,$$

$$\mathcal{C}'(i,i') = \{c'(i,i',u,J) : u \in V,\; J \in \mathcal{J}^*\} \qquad \text{for } i \in [k] \text{ and } i' \in [k]^*,$$

$$\mathcal{C}''(i,i') = \{c''(i,i',J,u) : J \in \mathcal{J}^*,\; u \in V\} \qquad \text{for } i \in [k]^* \text{ and } i' \in [k].$$

We let $\mathfrak{R}_0$ denote the set of k'' red blocks defined above. These will be referred to as the *basic blocks*. Additional blocks of blue and red vertices will be added to G' later in the construction.

The semantics of the reduction. Our semantic intentions in the design of $G' = (\mathcal{R}, \mathcal{B}, E')$ can be summarized as follows. The k blocks of R_1 represent the choice of k vertices of V that may form a k-element perfect code in G. The $k+1$ blocks of R_2 represent the intervals between the consecutive choices of vertices of V represented in R_1. The blocks of R_3, R_4, R_5 do not carry any meaning with respect to G; these blocks are needed to impose an internal structure on G'.

Although we have only started to describe G', enough is already visible to enable us to describe, at least in part, the solution translations of the reduction. This description may serve to motivate and clarify some of the forthcoming details of the proof. There are two different solution translations, namely, translations in the

forward direction: the manner in which a k-element perfect code in G translates into a k'-element perfect code in G', and the

backward direction: the manner in which a k'-element perfect code (or dominating set) in G' translates into a k-element perfect code in G.

The forward solution translation. In the forward direction, suppose that the elements of a perfect code of size k in $G = (V,E)$ are given by $u_1, u_2, \ldots, u_k$ and w.l.o.g. assume that

$$u_1 \ < \ u_2 \ < \ \cdots \ < \ u_k.$$

Let $J_0 = J(0, u_1)$. In other words, J_0 is the interval (set) of elements of V that precede u_1. For $i = 1, 2, \ldots, k-1$, let $J_i = J(u_i, u_{i+1})$. Thus J_i is the interval of elements of V that are properly between u_i and u_{i+1}. Finally, let $J_k = J(u_k, \infty)$ be the interval of elements of V following u_k. Now consider the sets

$$S_1 = \{\, a(i, u_i) \ : \ i \in [k] \,\}, \tag{3.2}$$

$$S_2 = \{\, b(i, J_i) \ : \ i \in [k]^* \}, \tag{3.3}$$

$$S_3 = \{\, c(i, i', J_i, J_{i'}) \ : \ i \in [k]^*, i' \in [k]^* \}, \tag{3.4}$$

$$S_4 = \{\, c'(i, i', u_i, J_{i'}) \ : \ i \in [k], \ i' \in [k]^* \}, \tag{3.5}$$

$$S_5 = \{\, c''(i, i', J_i, u_{i'}) \ : \ i \in [k]^*, \ i' \in [k] \,\}, \tag{3.6}$$

and let $S = S_1 \cup S_2 \cup \cdots \cup S_5$. Notice that S contains precisely one vertex from each of the k'' basic red blocks, where k'' is given by (3.1). Our construction of $G' = (\mathcal{R}, \mathcal{B}, E')$ will ensure that the set S may be extended to a perfect code in G'. This extension is accomplished by adding two more vertices for each vertex in S_3, S_4, and S_5, as specified later in our construction.

The backward solution translation. Our construction of G' will also ensure that any k'-element dominating set S in G' must be distributed so that there is exactly one element of S in each of the k'' basic blocks of $\mathfrak{R}_0$. Furthermore, S will be forced to have the restricted form of a perfect code in G'. The construction will ensure that for such a set S, the u-indices of the k elements of $S \cap R_1$ are distinct and that these indices correspond to a k-element perfect code in G.

The construction. We will describe $G' = (\mathcal{R}, \mathcal{B}, E')$ by starting with the k'' basic red blocks of $\mathfrak{R}_0$ and applying various *operators* to these blocks. Each application of an operator results in further blocks of red and blue vertices being created, along with various edges. A high-level blueprint for $G' = (\mathcal{R}, \mathcal{B}, E')$ in terms of these operators is shown in Figure 3.1.

In constructing $G' = (\mathcal{R}, \mathcal{B}, E')$, operators are applied only to argument sets of red blocks. Thus each blue vertex $\beta \in \mathcal{B}$ is created by a single specific application of an operator, and the neighborhood of β is completely established by this application. This allows us to argue a series of claims concerning properties of G' as we continue to describe the steps of the construction.

The block guard operator $\mathbf{\Gamma_1}$. The operator Γ_1 takes a single red block $\mathcal{X}$ as an argument and adds one blue vertex connected to every red vertex in $\mathcal{X}$.

The last step of the construction of $G' = (\mathcal{R}, \mathcal{B}, E')$ will be to apply this operator to each of the red blocks of the construction. When we get to the last step, there will be k' red blocks, the collection of which will be denoted $\mathfrak{R}$.

The last step. *Apply the block guard operator* Γ_1 *to every red block.*

We cannot make this the first step of the construction because some of the k' red blocks to which Γ_1 is to be applied have yet to be created by applications of operators in earlier steps of the construction. However, Γ_1 does provide a simple initial example of an operator (the action of the block guard operator is illustrated in Figure 3.2), and we can easily prove certain important properties of $G' = (\mathcal{R}, \mathcal{B}, E')$ having only this much information about the construction.

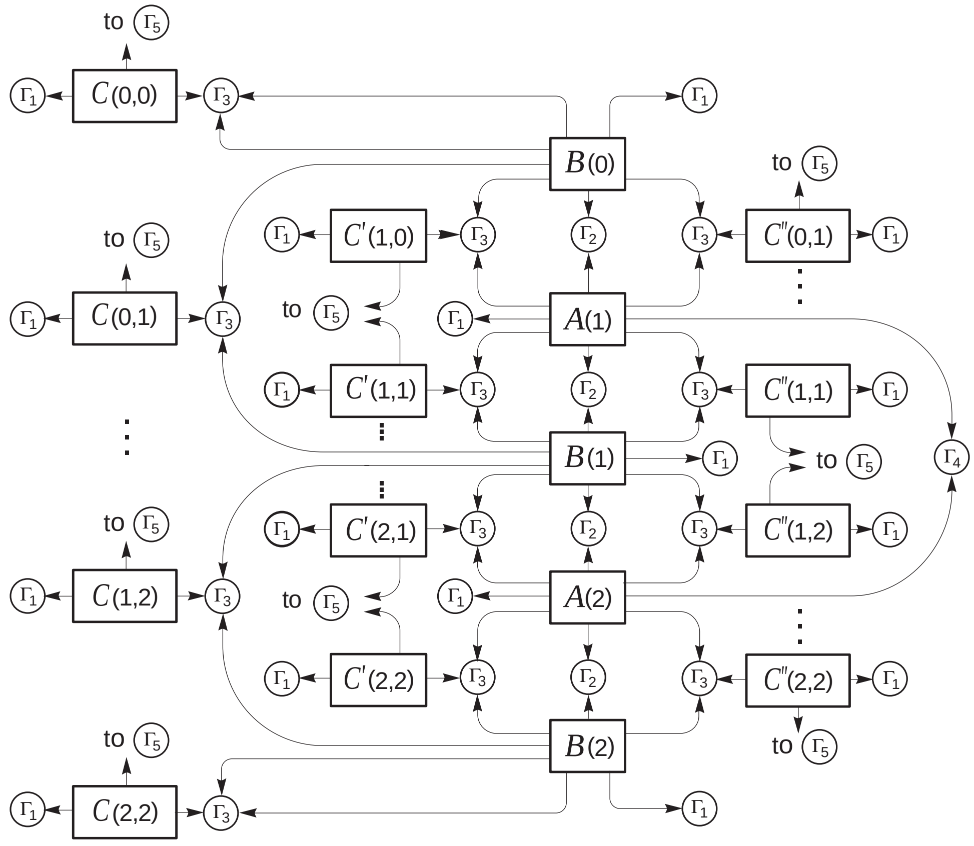


FIG. 3.1. *The blueprint for G' in terms of basic blocks and operators (example for $k = 2$).*

For example, the following lemma follows directly from the definition of Γ_1. This simple lemma already establishes property P1 of Theorem 2.2.

LEMMA 3.1. *Every dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$ has size at least k' and contains at least one vertex from each of the k' red blocks of $\mathfrak{R}$.*

Proof. Let $S \subseteq \mathcal{R}$ be a dominating set in G', and let $\mathcal{X} \in \mathfrak{R}$ be a red block. If $S \cap \mathcal{X} = \varnothing$, then the blue vertex $\beta \in \mathcal{B}$ created by the application of Γ_1 to $\mathcal{X}$ does not have a neighbor in S. ◻

Next, we need a definition. Let Γ be an operator; let A denote the union of the red blocks that either provide the arguments for the application of Γ or are created by the application of Γ; and let B denote the set of blue vertices created by the application of Γ.

DEFINITION 3.2. *We say that the operator Γ is* locally perfect *if the following condition is satisfied for every subset S of A that contains exactly one vertex from each red block contained in A: if every vertex in B has at least one neighbor in S, then every vertex in B has a unique neighbor in S. In other words, if S contains one vertex from each red block and is "locally" a dominating set, then S is necessarily a "local" perfect code.*

The above definition is motivated by the observation that if every application of an operator in the construction of $G' = (\mathcal{R}, \mathcal{B}, E')$ is locally perfect, then property P2 of Theorem 2.2 will hold. This observation is a corollary to the following lemma.

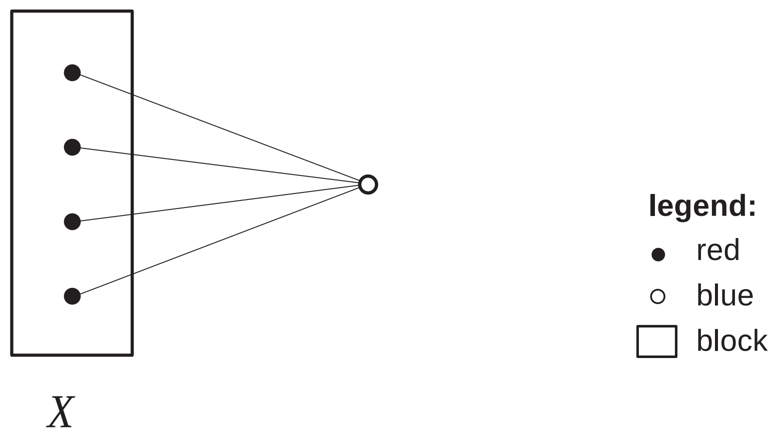


FIG. 3.2. *The block guard operator* Γ_1.

LEMMA 3.3. *Every k'-element dominating set S in G' must contain exactly one vertex from each of the k' red blocks of $\mathfrak{R}$, and the last step in the construction of G' is locally perfect.*

Proof. Since each of the k' blocks of $\mathfrak{R}$ must contain at least one element of S by Lemma 3.1, each block must contain exactly one element of S. Local perfection of Γ_1 is trivial. □

The branch operator $\mathbf{\Gamma_2}$. The arguments to the Γ_2 operator are two red blocks $\mathcal{X}$ and $\mathcal{X}'$, given together with a partition of $\mathcal{X}'$ into $|\mathcal{X}|$ or more nonempty classes, and an injective assignment to each element of $\mathcal{X}$ of a different class in the partition of $\mathcal{X}'$; that is,

$$x \in \mathcal{X} \ \mapsto \ \mathcal{S}_x \subset \mathcal{X}'$$

so that $\mathcal{S}_x \cap \mathcal{S}_y = \varnothing$ for $x \neq y$. The result of applying Γ_2 to $\mathcal{X}$ and $\mathcal{X}'$ with this assignment is

a. the creation of the set of $|\mathcal{X}|$ blue vertices

$$\{\, \beta(x) \ : \ x \in \mathcal{X} \,\},$$

b. for each blue vertex $\beta(x)$ in this set, the creation of edges connecting $\beta(x)$ to $x \in \mathcal{X}$ and to all the red vertices in $\mathcal{X}'$ that belong to $\mathcal{S}_y$ for some $y \neq x$ in $\mathcal{X}$.

LEMMA 3.4. *The branch operator Γ_2 is locally perfect. In particular, suppose that this operator is applied to the red blocks $\mathcal{X}$ and $\mathcal{X}'$, and let S be a k'-element dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$. Then $S \cap \mathcal{X} = \{x\}$ and $S \cap \mathcal{X}' = \{x'\}$ for some $x' \in \mathcal{S}_x$.*

Proof. The fact that S intersects each of the blocks $\mathcal{X}$ and $\mathcal{X}'$ at a single vertex, say, x and x', follows immediately from Lemma 3.3. The vertex $x \in \mathcal{X}$ is adjacent to the single blue vertex $\beta(x)$. If $x' \in \mathcal{X}'$ belongs to a partition class that is not assigned to a vertex of $\mathcal{X}$, then x' is not adjacent to any of the blue vertices created by Γ_2 and the set S dominates only the single vertex $\beta(x)$, a contradiction. Similarly, if $x' \in \mathcal{S}_y$ for some vertex $y \neq x$ in $\mathcal{X}$, then the blue vertex $\beta(y)$ is not adjacent to either x or x'. Hence $\beta(y)$ is not dominated by S, again a contradiction. Thus $x' \in \mathcal{S}_x$, which is the only remaining case. It now follows that every blue vertex created by Γ_2 has a

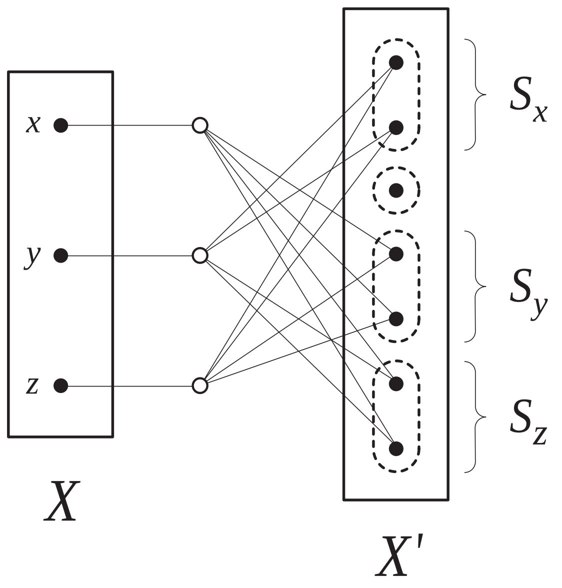


FIG. 3.3. *The branch operator* Γ_2.

unique neighbor in S. The unique neighbor of $\beta(x)$ is x and x' is the unique neighbor of every other blue vertex. Thus, the branch operator Γ_2 is locally perfect. $\square$

A useful special case of the application of Γ_2 is as "equality" enforcer. For this, we assume that the arguments to Γ_2 are isomorphic sets with the isomorphism given by the correspondence

$$x \in \mathcal{X} \quad\longleftrightarrow\quad e(x) \in \mathcal{X}',$$

and in applying the operator Γ_2, we use the natural assignment $x \mapsto \mathcal{S}_x = \{e(x)\}$. In this situation, assuming that S is a k'-element dominating set in G', we observe that $S \cap \mathcal{X} = \{x\}$ and $S \cap \mathcal{X}' = \{y\}$ necessarily imply that $y = e(x)$ in view of Lemma 3.4. With Γ_2 at hand, we are finally ready to describe the first two steps of our construction.

Step 1. *For* $i = 1, 2, \ldots, k-1$, *apply the operator* Γ_2 *to the arguments* $\mathcal{X} = \mathcal{A}(i)$ *and* $\mathcal{X}' = \mathcal{B}(i)$ *with the assignment*

$$a(i,u) \;\mapsto\; \{\, b(i,J) \;:\; J \in \mathcal{J}^*,\ \partial(J) = u \,\}. \tag{3.7}$$

Also apply Γ_2 *to the arguments* $\mathcal{X} = \mathcal{A}(k)$ *and* $\mathcal{X}' = \mathcal{B}(k)$ *with the assignment*

$$a(k,u) \;\mapsto\; \{\, b(k,J) \;:\; J \in \mathcal{J}^*,\ \partial(J) = u \ \text{ and } \ \partial'(J) = \infty \,\}. \tag{3.8}$$

Step 2. *For* $i = 2, 3, \ldots, k$, *apply the operator* Γ_2 *to the arguments* $\mathcal{X} = \mathcal{A}(i)$ *and* $\mathcal{X}' = \mathcal{B}(i-1)$ *with the assignment*

$$a(i,u) \;\mapsto\; \{\, b(i-1,J) \;:\; J \in \mathcal{J}^*,\ \partial'(J) = u \,\}. \tag{3.9}$$

Also apply Γ_2 *to the arguments* $\mathcal{X} = \mathcal{A}(1)$ *and* $\mathcal{X}' = \mathcal{B}(0)$ *with the assignment*

$$a(1,u) \;\mapsto\; \{\, b(0,J) \;:\; J \in \mathcal{J}^*,\ \partial'(J) = u \ \ \textit{and}\ \ \partial(J) = 0 \,\}. \tag{3.10}$$

The first two steps of the construction operate on the vertices in R_1 and R_2 that represent, respectively, the choice of k vertices of G and the intervals between these chosen vertices. The purpose of these two steps is reflected in the following lemma.

LEMMA 3.5. *Suppose that S is a k'-element dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$. Then the $k + (k+1)$ vertices of $S \cap R_1$ and $S \cap R_2$ consistently represent k vertices of $G = (V, E)$ and the intervals between these vertices in the following way:*

a. *If $S \cap \mathcal{A}(1) = \{a(1,u)\}$, then $S \cap \mathcal{B}(0) = \{b(0, J(0,u))\}$.*
b. *For $i = 1, 2, \ldots, k-1$, if $S \cap \mathcal{A}(i) = \{a(i,u)\}$ and $S \cap \mathcal{A}(i+1) = \{a(i+1,v)\}$, then $S \cap \mathcal{B}(i) = \{b(i, J(u,v))\}$.*
c. *If $S \cap \mathcal{A}(k) = \{a(k,u)\}$, then $S \cap \mathcal{B}(k) = \{b(k, J(u,\infty))\}$.*
d. *For all $i < i'$, if $S \cap \mathcal{A}(i) = \{a(i,u)\}$ and $S \cap \mathcal{A}(i') = \{a(i',u')\}$, then $u < u'$.*

Proof. Referring to the last application of the operator Γ_2 in Step 2, it follows from Lemma 3.4 that if $S \cap \mathcal{A}(1) = \{a(1,u)\}$, then $S \cap \mathcal{B}(0)$ consists of a vertex x' which belongs to the set assigned to $a(1,u)$ on the right-hand side of (3.10). But this set consists of the single element $b(0, J(0,u))$, which establishes part (a). Part (c) follows in a similar fashion from Lemma 3.4 and (3.8). Part (b) follows from Lemma 3.4 along with the combination of (3.7) and (3.9). Indeed, assuming the condition of part (b), it follows from Lemma 3.4 that $S \cap \mathcal{B}(i) = \{x'\}$, where x' belongs to the intersection of the set assigned to $a(i,u)$ in (3.7) with the set assigned to $a(i+1,v)$ in (3.9). Thus $x' = b(i,J)$, where $\partial(J) = u$ and $\partial'(J) = v$, that is, $J = J(u,v)$. Part (d) follows immediately from part (b). □

The product operator $\mathbf{\Gamma_3}$. The arguments for the operator are three red blocks $\mathcal{X}_1$, $\mathcal{X}_2$, and $\mathcal{Z}$, where $\mathcal{Z}$ is isomorphic to the product $\mathcal{X}_1 \times \mathcal{X}_2$, together with a one-to-one correspondence:

$$z \in \mathcal{Z} \;\longleftrightarrow\; (x_1, x_2) \in \mathcal{X}_1 \times \mathcal{X}_2. \tag{3.11}$$

We write $z = z(x_1, x_2)$ to denote the element of $\mathcal{Z}$ that corresponds to the pair $(x_1, x_2) \in \mathcal{X}_1 \times \mathcal{X}_2$ in (3.11). An application of Γ_3 augments $G' = (\mathcal{R}, \mathcal{B}, E')$ in the following ways:

a. Two auxiliary blocks of red vertices are created:

$$\mathcal{P}_1 = \{\, \rho_1(x_1,x_2) \;:\; x_1 \in \mathcal{X}_1,\ x_2 \in \mathcal{X}_2 \,\},$$
$$\mathcal{P}_2 = \{\, \rho_2(x_1,x_2) \;:\; x_1 \in \mathcal{X}_1,\ x_2 \in \mathcal{X}_2 \,\}.$$

b1. The operator Γ_2 is applied to the arguments $\mathcal{X} = \mathcal{X}_1$ and $\mathcal{X}' = \mathcal{P}_1$ with the assignment

$$x \in \mathcal{X}_1 \;\mapsto\; \{\, \rho_1(x,y) \;:\; y \in \mathcal{X}_2 \,\}. \tag{3.12}$$

b2. The operator Γ_2 is applied to the arguments $\mathcal{X} = \mathcal{X}_2$ and $\mathcal{X}' = \mathcal{P}_2$ with the assignment

$$y \in \mathcal{X}_2 \;\mapsto\; \{\, \rho_2(x,y) \;:\; x \in \mathcal{X}_1 \,\}. \tag{3.13}$$

c. The operator Γ_2 is applied as an equality enforcer to the arguments $\mathcal{X} = \mathcal{P}_1$ and $\mathcal{X}' = \mathcal{P}_2$ with the assignment

$$\rho_1(x_1,x_2) \;\mapsto\; \{\, \rho_2(x_1,x_2) \,\}. \tag{3.14}$$

d1. The operator Γ_2 is applied as an equality enforcer to the arguments $\mathcal{X} = \mathcal{P}_2$ and $\mathcal{X}' = \mathcal{Z}$ with the assignment

$$\rho_2(x_1, x_2) \mapsto \{\, z(x_1, x_2) \,\}. \tag{3.15}$$

d2. The operator Γ_2 is applied as an equality enforcer to the arguments $\mathcal{X} = \mathcal{Z}$ and $\mathcal{X}' = \mathcal{P}_1$ with the assignment

$$z(x_1, x_2) \mapsto \{\, \rho_1(x_1, x_2) \,\}. \tag{3.16}$$

Lemma 3.6. *The product operator Γ_3 is locally perfect. In particular, suppose that this operator is applied to the three red blocks $\mathcal{X}_1, \mathcal{X}_2$, and $\mathcal{Z}$, and let S be a k'-element dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$ with $S \cap \mathcal{X}_1 = \{x\}$ and $S \cap \mathcal{X}_2 = \{y\}$. Then $S \cap \mathcal{Z} = \{z(x, y)\}$.*

Proof. It follows from Lemma 3.3 that S intersects each of the red blocks $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{Z}$ at a single vertex, say, $\rho_1(x', y') \in \mathcal{P}_1$, $\rho_2(x'', y'') \in \mathcal{P}_2$, and $z \in \mathcal{Z}$, respectively. We can now argue a series of simple observations regarding $\rho_1(x', y'), \rho_2(x'', y'')$, and z, which follow from Lemma 3.4 in conjunction with the assignments in (3.12)–(3.16). In view of (3.12), respectively, (3.13), we have that $x' = x$, respectively, $y'' = y$. By the equality enforcing assignment in (3.14), we have $x' = x''$ and $y' = y''$. Thus $x'' = x' = x$ and $y'' = y' = y$. It now follows from either (3.15) or (3.16) that $z = z(x, y)$.

Since the product operator Γ_3 is composed from five applications of the branch operator Γ_2, the local perfection of Γ_3 follows from the local perfection of Γ_2 established in Lemma 3.4. Alternatively, this can be proved directly by verifying that each of the $|\mathcal{X}_1| + |\mathcal{X}_2| + 3\,|\mathcal{X}_1||\mathcal{X}_2|$ blue vertices created by Γ_3 is adjacent to one and only one of the five red vertices $x \in \mathcal{X}_1$, $y \in \mathcal{X}_2$, $\rho_1(x, y) \in \mathcal{P}_1$, $\rho_2(x, y) \in \mathcal{P}_2$, and $z(x, y) \in \mathcal{Z}$, regardless of the choice of $x \in \mathcal{X}_1$ and $y \in \mathcal{X}_2$. ∎

With the product operator Γ_3 at hand, we can now describe the next three steps in our construction of $G' = (\mathcal{R}, \mathcal{B}, E')$. These three steps establish a relation between the blocks in R_3, R_4, R_5 and the blocks of R_1, R_2 which represent the choice of some k vertices of G according to Lemma 3.5.

Step 3. *For each pair (i, i') with $i, i' \in [k]^*$, apply the operator Γ_3 to the arguments $\mathcal{X}_1 = \mathcal{B}(i)$, $\mathcal{X}_2 = \mathcal{B}(i')$, and $\mathcal{Z} = \mathcal{C}(i, i')$ with the product correspondence*

$$c(i, i', J, J') \in \mathcal{C}(i, i') \;\longleftrightarrow\; (b(i, J), b(i', J')) \in \mathcal{B}(i) \times \mathcal{B}(i').$$

Step 4. *For each pair (i, i') with $i \in [k]$ and $i' \in [k]^*$, apply the operator Γ_3 to the arguments $\mathcal{X}_1 = \mathcal{A}(i)$, $\mathcal{X}_2 = \mathcal{B}(i')$, and $\mathcal{Z} = \mathcal{C}'(i, i')$ with the product correspondence*

$$c'(i, i', u, J) \in \mathcal{C}'(i, i') \;\longleftrightarrow\; (a(i, u), b(i', J)) \in \mathcal{A}(i) \times \mathcal{B}(i').$$

Step 5. *For each pair (i, i') with $i \in [k]^*$ and $i' \in [k]$, apply the operator Γ_3 to the arguments $\mathcal{X}_1 = \mathcal{B}(i)$, $\mathcal{X}_2 = \mathcal{A}(i')$, and $\mathcal{Z} = \mathcal{C}''(i, i')$ with the product correspondence*

$$c''(i, i', J, u) \in \mathcal{C}''(i, i') \;\longleftrightarrow\; (b(i, J), a(i', u)) \in \mathcal{B}(i) \times \mathcal{A}(i').$$

At this stage of the construction, the red/blue graph $G' = (\mathcal{R}, \mathcal{B}, E')$ is highly structured, but the adjacency structure of G' is *independent* of the adjacency structure of $G = (V, E)$. The following operator will finally establish the connection between $G = (V, E)$ and $G' = (\mathcal{R}, \mathcal{B}, E')$.

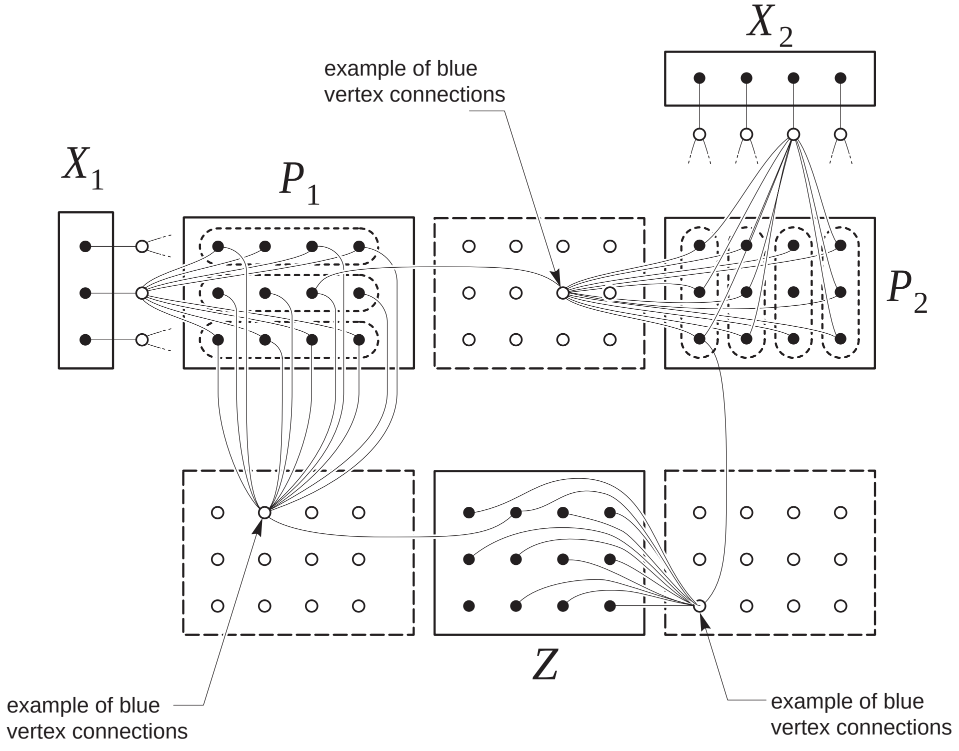

X_2
example of blue vertex connections
X_1
P_1
P_2
Z
example of blue vertex connections
example of blue vertex connections

FIG. 3.4. *The product operator* Γ_3.

The G-adjacency operator Γ_4. This operator takes all the vertices of R_1 as arguments, that is, all the $\mathcal{A}$-blocks $\mathcal{A}(1), \mathcal{A}(2), \ldots, \mathcal{A}(k)$. A set of $n = |V|$ blue vertices

$$\{\, \beta(u) \ : \ u \in V \,\}$$

is created. These vertices are connected as follows. For each $i = 1, 2, \ldots, k$, the blue vertex $\beta(u)$ is connected to $a(i, u)$ and to $a(i, v)$ for all $v \in V$ that are adjacent to u in G.

It is easy to see that an application of Γ_4 essentially amounts to replicating k times the original graph $G = (V, E)$ as a red/blue bipartite graph with the set of red vertices $\mathcal{A}(i)$ and the set of blue vertices $\{\beta(u) : u \in V\}$, both isomorphic to V. This is precisely what we do next.

Step 6. *Apply the operator* Γ_4 *to all of the* $\mathcal{A}$*-blocks* $\mathcal{A}(1), \mathcal{A}(2), \ldots, \mathcal{A}(k)$.

Let S be a k'-element dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$. We know from Lemma 3.3 that S intersects each of the red blocks, in particular each of the blocks $\mathcal{A}(1), \mathcal{A}(2), \ldots, \mathcal{A}(k)$, at a single vertex. Thus for each $i = 1, 2, \ldots, k$, we can define v_i to be the unique vertex of V such that $S \cap \mathcal{A}(i) = \{a(i, v_i)\}$. With this notation, let $V(S) = \{v_1, v_2, \ldots, v_k\}$.

LEMMA 3.7. *The set* $V(S)$ *is a* k*-element dominating set in* G*. Furthermore, if* S *is a perfect code in* $G' = (\mathcal{R}, \mathcal{B}, E')$*, then* $V(S)$ *is a* k*-element perfect code in* G*.*

Proof. It follows from Lemma 3.5 that $v_1, v_2, \ldots, v_k$ are all distinct (in fact $v_1 < v_2 < \cdots < v_k$). Hence $|V(S)| = k$. The rest is immediate from the definition of Γ_4: a vertex $u \in V$ has a (unique) neighbor in $V(S)$ if and only if the vertex $\beta(u)$ created by Γ_4 has a (unique) neighbor in S. ◻

Notice that the G-adjacency operator Γ_4 is not locally perfect unless $V(S)$ is actually a perfect code in $G = (V, E)$ for every k'-element dominating set S in G'. However, the following operator ensures that the latter condition is always satisfied.

We say that two distinct vertices u and u' in $G = (V, E)$ are *close* if there is a path length of 1 or 2 between them. (Notice that a vertex u is not considered close to itself.) It is easy to see that a dominating set in G is also a perfect code if and only if it does not contain close vertices.

The perfection operator Γ_5. This operator takes as arguments all of the $\mathcal{C}, \mathcal{C}'$, and $\mathcal{C}''$ blocks. An application of the operator results in the following.

a. The creation of a set of blue vertices with one vertex for each ordered pair (u, u') of close vertices in G, that is,

$$\{\, \beta(u, u') \ :\ u \text{ is close to } u' \text{ in } G \,\}. \tag{3.17}$$

b. The creation of edges connecting each blue vertex $\beta(u, u')$ in this set to all the red vertices contained in one of the following three sets:

$$\begin{aligned}
\mathbb{C}(u, u') &= \{\, c(i, i', J, J') \ :\ i, i' \in [k]^*,\ u \in J,\ u' \in J' \,\},\\
\mathbb{C}'(u, u') &= \{\, c'(i, i', u, J) \ :\ i \in [k],\ i' \in [k]^*,\ u' \in J \,\},\\
\mathbb{C}''(u, u') &= \{\, c''(i, i', J, u') \ :\ i \in [k]^*,\ i' \in [k],\ u \in J \,\}.
\end{aligned}$$

The idea of the perfection operator Γ_5 is to ensure that if $V(S)$ contains close vertices, then S cannot be a dominating set in $G' = (\mathcal{R}, \mathcal{B}, E')$. This is accomplished in the following step.

Step 7. *Apply the perfection operator* Γ_5 *to all of the* $\mathcal{C}, \mathcal{C}'$*, and* $\mathcal{C}''$ *blocks.*

Next, we need some more notation. We again let S denote a k'-element dominating set in G' and define the sets $J_0, J_1, \ldots, J_k$ as follows:

$$J_i \quad = \quad J(u, v) \subset V \qquad \text{so that } S \cap \mathcal{B}(i) = \{b(i, J(u, v))\} \qquad \text{for } i = 0, 1, \ldots, k.$$

It follows from Lemma 3.5 that the sets $J_0, J_1, \ldots, J_k$ are well defined, and the sequence of sets $J_0, \{v_1\}, J_1, \{v_2\}, J_2, \ldots, J_{k-1}, \{v_k\}, J_k$ constitutes a partition of the vertex set V of G.

LEMMA 3.8. *The perfection operator* Γ_5 *is locally perfect, and the set* $V(S) = \{v_1, v_2, \ldots, v_k\}$ *does not contain close vertices.*

Proof. With the notation just defined, it follows from Lemma 3.6 along with Steps 3, 4, and 5 of our construction that

$$\begin{aligned}
S \cap \mathcal{C}(i, i') &= \{\, c(i, i', J_i, J_{i'}) \,\} &&\text{for all } i \in [k]^* \text{ and } i' \in [k]^*,\\
S \cap \mathcal{C}'(i, i') &= \{\, c'(i, i', v_i, J_{i'}) \,\} &&\text{for all } i \in [k] \text{ and } i' \in [k]^*,\\
S \cap \mathcal{C}''(i, i') &= \{\, c''(i, i', J_i, v_{i'}) \,\} &&\text{for all } i \in [k]^* \text{ and } i' \in [k].
\end{aligned}$$

Now consider a pair (u, u') of close vertices in G. Since $J_0, \{v_1\}, J_1, \{v_2\}, J_2, \ldots, J_{k-1}, \{v_k\}, J_k$ is a partition of the vertex set V of G, exactly one statement is true on the following list describing where u is to be found and where u' is to be found.

Case 1: $u \in J_i$ and $u' \in J_{i'}$.
In this case $\beta(u, u')$ in (3.17) is adjacent to a single red vertex in S. Specifically, $\beta(u, u')$ is connected to $c(i, i', J_i, J_{i'}) \in \mathbb{C}(u, u')$, where $\{\, c(i, i', J_i, J_{i'}) \,\} = S \cap \mathcal{C}(i, i')$.

Case 2: $u = v_i$ and $u' \in J_{i'}$.
In this case $\beta(u, u')$ in (3.17) is adjacent to a single red vertex in S. Specifically, $\beta(u, u')$ is connected to $c'(i, i', v_i, J_{i'}) \in \mathbb{C}'(u, u')$, where $\{\, c'(i, i', v_i, J_{i'}) \,\} = S \cap \mathcal{C}'(i, i')$.

Case 3: $u \in J_i$ and $u' = v_{i'}$.
In this case $\beta(u, u')$ in (3.17) is adjacent to a single red vertex in S. Specifically, $\beta(u, u')$ is connected to $c''(i, i', J_i, v_{i'}) \in \mathbb{C}''(u, u')$, where $\{\, c''(i, i', J_i, v_{i'}) \,\} = S \cap \mathcal{C}''(i, i')$.

Case 4: $u = v_i$ and $u' = v_{i'}$.
In this case $\beta(u, u')$ is *not* adjacent to any of the vertices of S.

In each case, we see that a blue vertex $\beta(u, u')$ created by Γ_5 is adjacent to at most one vertex of S, which implies that Γ_5 is locally perfect. Furthermore, since S is a dominating set by assumption, Case 4 above cannot happen. In other words, there is no pair of close vertices in the set $V(S)$ "chosen" by a k'-element dominating set S in $G' = (\mathcal{R}, \mathcal{B}, E')$. ◻

LEMMA 3.9. *The set $V(S)$ is a k-element perfect code in $G = (V, E)$.*

Proof. By Lemma 3.7, the set $V(S)$ is a k-element dominating set in G, and by Lemma 3.8 it does not contain close vertices. Hence $V(S)$ is a perfect code. ◻

We can now complete our construction and complete the proof of Theorem 2.1. Recall that the last step in the construction is as follows.

Step 8. *Apply the block guard operator Γ_1 to every red block constructed thus far.*

Theorem 2.2 now follows from a series of easy observations. Property P1 of Theorem 2.2 is established in Lemma 3.1. The fact that $V(S)$ is necessarily a perfect code in G, established in Lemma 3.9, further implies that the G-adjacency operator Γ_4 is locally perfect. Thus *all* the operators used in our construction are locally perfect, and property P2 of Theorem 2.2 holds. The "if" part of property P3 then follows directly from Lemma 3.9. The reader can now verify the few remaining details to see that the forward translation of a k-element perfect code in $G = (V, E)$ to a k'-element perfect code in $G' = (\mathcal{R}, \mathcal{B}, E')$, outlined in (3.2)–(3.6), works correctly.

4. Membership in the parametrized complexity class *W*[2].

All of the hardness results for parametrized complexity that we derive in this paper are by reduction from the PERFECT CODE problem. This particular problem has eluded exact classification in the $W[t]$ hierarchy for a number of years. What is known [15] is that PERFECT CODE is hard for $W[1]$ and belongs to $W[2]$. It may be a representative of a natural parametrized complexity degree, which is intermediate between the $W[1]$ and $W[2]$ complexity classes. This remains an interesting open problem in the structure of the parametric complexity classes, especially in view of connections to one-per-clause satisfiability problems.

In this section we indicate how some of the problems considered in this paper, namely, MAXIMUM-LIKELIHOOD DECODING, WEIGHT DISTRIBUTION, and MINIMUM DISTANCE, can be shown to belong to the parametrized complexity class $W[2]$. A wide variety of problems are known to be complete or hard for $W[2]$, including DOMINATING SET, BANDWIDTH, LENGTH-k FACTORIZATION OF MONOIDS, LONGEST COMMON SUBSEQUENCE FOR k-SEQUENCES, and k-PROCESSOR PRECEDENCE CONSTRAINED SCHEDULING (see [8, 12, 6, 7]).

To establish the necessary background for Theorem 4.4 below, we now briefly recall the definition of the classes of the $W[t]$ hierarchy based on problems about bounded-depth circuits. We will generally follow the exposition in [14]. We first define circuits in which some logic gates have bounded fan-in and some have unrestricted fan-in. It is assumed that fan-out is never restricted.

Definition 4.1. *A Boolean circuit is of* mixed type *if it consists of circuits having gates of the following kinds:*

small gates*: ¬ gates, ∧ gates, and ∨ gates with bounded fan-in; we will usually assume that the bound on fan-in is* 2 *for ∧ gates and ∨ gates and* 1 *for ¬ gates.*

large gates*: ∧ gates and ∨ gates with unrestricted fan-in.*

The *depth* of a circuit $\mathcal{C}$ is defined to be the maximum number of gates (small or large) on an input-output path in $\mathcal{C}$. The *weft* of a circuit $\mathcal{C}$ is the maximum number of large gates on an input-output path in $\mathcal{C}$. We say that a family of decision circuits F has *bounded depth* if there is a constant h such that every circuit in the family F has depth at most h. We say that F has *bounded weft* if there is constant t such that every circuit in the family F has weft at most t.

Let F be a family of mixed-type decision circuits. We allow that F may have many different circuits with a given number of inputs. To F we associate the following parametrized circuit problem $L_F = \{ (\mathcal{C}, k) : \mathcal{C} \in F$ accepts some input vector of weight $k\}$, where the (Hamming) weight of a Boolean vector x is the number of 1's in the vector.

Definition 4.2. *A parametrized language L belongs to $W[t]$ if L reduces to the parametrized circuit problem $L_{F(t,h)}$ for the family $F(t,h)$ of mixed-type decision circuits of weft at most t and depth at most h for some constant h.*

Definition 4.3. *A parametrized language L belongs to $W^*[t]$ if it belongs to $W[t]$ with the definition of a* small gate *being revised to allow fan-in bounded by a fixed arbitrary function of k and where the depth of a circuit is allowed to be a function of k as well.*

Whether $W^*[t] = W[t]$ for all t is an important open problem. The significance of this question is that for purposes of establishing membership in the $W[t]$ hierarchy, we would like to have the most generous possible characterization of $W[t]$ available to work with. It is shown that $W^*[1] = W[1]$ and $W^*[2] = W[2]$ in [21] and [18], respectively. For $t \geq 3$ the question is still open. Our argument here makes essential use of the result of [18] that $W^*[2] = W[2]$.

Theorem 4.4. Weight Distribution *belongs to* $W[2]$.

Proof. Given an instance H and k of Weight Distribution, we describe how to compute a pair (E, k'), consisting of a Boolean expression E and a positive integer k', such that the circuit corresponding to E has a form allowed by the definition of $W^*[2] = W[2]$ and such that H, k is a yes-instance of Weight Distribution if and only if E is satisfied by a weight k' truth assignment. In order for this to be a parametric reduction, we must have k' computed purely as a function of k. Our reduction is simple in this regard: we take $k' = k$.

Suppose that H is an $m \times n$ binary matrix, and let $h_1, h_2, \ldots, h_n$ denote the columns of this matrix. For $j = 1, 2, \ldots, m$, we will write $h_i[j]$ to denote the jth component of h_i. The set V of Boolean variables for E is

$$V = \{ v[b,i] : b = 1, 2, \ldots, k \text{ and } i = 1, 2, \ldots, n \}.$$

Intuition can be served by viewing V as consisting of k "choice blocks," each of size n. The expression E is constructed so that any satisfying truth assignment must make exactly one variable in each of these blocks `true`, and will in this way indicate a set of k columns of H.

Let $\mathcal{E}$ denote the set of all subsets of $\{1, 2, \ldots, k\}$ of even cardinality. For each $j \in \{1, 2, \ldots, m\}$ and $\sigma \in \mathcal{E}$, define the Boolean expressions:

$$E^+(\sigma,j) = \bigwedge_{b\in\sigma} \quad \bigwedge_{i:\, h_i[j]=0} \neg v[b,i],$$

$$E^-(\sigma,j) = \bigwedge_{b\notin\sigma} \quad \bigwedge_{i:\, h_i[j]=1} \neg v[b,i],$$

$$E(\sigma,j) = E^+(\sigma,j) \ \wedge\ E^-(\sigma,j).$$

Then the expression E has the form

$$E \ = \ E_1 \wedge E_2 \ \wedge\ E_3,$$

where

$$E_1 = \bigwedge_{1\le b<b'\le k} \ \bigwedge_{i=1}^{n} (\neg v[b,i] \ \vee\ \neg v[b',i]),$$

$$E_2 = \bigwedge_{b=1}^{k} \left(\bigvee_{i=1}^{n} v[b,i] \right),$$

$$E_3 = \bigwedge_{j=1}^{m} \ \bigvee_{\sigma\in\mathcal{E}} E(\sigma,j).$$

It is easy to see that if the definition of a small gate allows fan-in bounded by a function of k, then each of the subexpressions E_1 and E_2 has weft one, while the subexpression E_3 has weft two. The depth of E_1, E_2, and E_3 is 4, 2, and 6, respectively, so that the depth of E itself is 7. Thus E belongs to a family of weft-two circuits allowed by the definition of $W^*[2]$. We next argue the correctness of the reduction. Note the validity of the following easy claims.

CLAIM 1. *The subexpression E_2 is satisfied by a weight k truth assignment to the variables of V if and only if exactly one variable in each of the k blocks is assigned the value* `true`.

CLAIM 2. *The subexpression $E' = E_1 \wedge E_2$ is satisfied by a weight k truth assignment to the variables of V if and only if exactly one variable in each of the k blocks is assigned the value* `true` *in such a way that the second indices of the* `true` *variables are all distinct.*

Let τ be a weight k truth assignment that satisfies E'. It follows from Claim 1 that for each $b \in \{1,2,\ldots,k\}$, there is a unique $i \in \{1,2,\ldots,n\}$ such that $v[b,i]$ is assigned the value `true` by τ. Thus τ and b together specify the unique ith column of H. For $j = 1,2,\ldots,m$, we let $\alpha(b,j,\tau)$ denote the binary value to be found in the jth row of this column, that is, $\alpha(b,j,\tau) = h_i[j]$.

CLAIM 3. *The expression E_3 is satisfied by a weight k truth assignment $\tau : V \to \{$*`true`*, *`false`*$\}$ that also satisfies $E' = E_1 \wedge E_2$ if and only if for each $j \in \{1,2,\ldots,m\}$ there is some $\sigma \in \mathcal{E}$ such that the following equality holds: $\sigma = \{b : \alpha(b,j,\tau) = 1\}$.*

Now suppose that H, k is a yes-instance of WEIGHT DISTRIBUTION, and let $h_{i_1}, h_{i_2}, \ldots, h_{i_k}$ be the k columns of H that sum to the all-zero vector. Let τ be the truth assignment that assigns the variables $v[1,i_1], v[2,i_2], \ldots, v[k,i_k]$ to be `true` and assigns all the other variables in V the value `false`. Clearly, τ has weight k. It follows from Claim 2 that τ satisfies $E' = E_1 \wedge E_2$. For $j = 1, 2, \dots, m$, let σ_j be the subset of $\{1, 2, \dots, k\}$ consisting of all b such that $h_{i_b}[j] = 1$. Since

$$h_{i_1}[j] + h_{i_2}[j] + \cdots + h_{i_k}[j] \;=\; 0 \pmod 2$$

for all j, it follows that the number of such indices b must be even. Hence $\sigma_j \in \mathcal{E}$ for all j. It is not difficult to see that for the truth assignment τ specified above, σ_j is precisely the set $\{b : \alpha(b, j, \tau) = 1\}$. Hence by Claim 3, τ also satisfies E_3 and therefore E itself.

Conversely, suppose that E has a truth assignment τ of weight k. By Claim 2, there are k distinct indices $i_1, i_2, \dots, i_k$ such that τ assigns to the k variables $v[1, i_1]$, $v[2, i_2], \dots, v[k, i_k]$ the value `true` and assigns all other variables in V the value `false`. By Claim 3, the number of elements in the set $\{b : \alpha(b, j, \tau) = 1\} = \{b : h_{i_b}[j] = 1\}$ must be even for each $j = 1, 2, \dots, m$. Hence the k columns $h_{i_1}, h_{i_2}, \dots, h_{i_k}$ of H sum to the all-zero vector. $\square$

The above argument can be easily modified to show that Maximum-Likelihood Decoding and Minimum Distance also belong to $W^*[2] = W[2]$. We conjecture that all of these problems are equivalent to the Perfect Code problem.

5. Concluding discussion and open problems.

We have shown that four of six fundamental computational problems in the domains of linear codes and integer lattices are NP-complete and hard for the parametrized complexity class $W[1]$. The obvious outstanding open problems are the following.

- Is the Minimum Distance problem, recently proved to be NP-complete in [32], also hard for $W[1]$?
- Is the Shortest Vector problem hard for NP and $W[1]$?

A consequence of the proof in [32] that the Minimum Distance problem is NP-hard is that Even Set is NP-hard. This leaves us in the curious situation that the only known proof of this seemingly quite combinatorial result is by means of sophisticated algebraic techniques deeply rooted in coding theory. Is there a direct combinatorial proof? Understanding this issue may shed some light on whether the combinatorial methods used here to show NP and $W[1]$ hardness for Theta Series can be extended to the Shortest Vector problem.

6. Appendix: Short survey of parametrized complexity.

Over the past several years, it has become increasingly clear that classical complexity frameworks such as NP-completeness and PSPACE-completeness are not adequate to address intractability questions for problems that are naturally parametrized and for which the important applications are covered by parameter values of, say, $k \leq 50$.

For example, consider the Vertex Cover problem, which asks whether a graph G on n vertices has a vertex cover of size at most k. This is one of the six NP-complete problems singled out for attention by Garey and Johnson [22]. The best known result, presently, is that Vertex Cover can be solved in time $O(kn + (4/3)^k k^2)$ with a very small hidden constant [3]. This means that although the problem is NP-complete, it is well solved for input graphs of *unlimited* size, as long as k is at most 70 or so. Strong tractability results such as this seem to be not uncommon when problems are qualitatively classifiable as fixed-parameter tractable, meaning that they belong to the parametrized complexity class FPT, formally defined below. Three of the six basic NP-complete problems considered by Garey and Johnson in [22, Chapter 3] are fixed-parameter tractable.

In general, a variety of metrics can be applied to the input of a computational problem. The total length of the input is one basic measurement, but it is by no means the only important one. It is natural to try to understand how different input measurements interact in determining problem complexity. Furthermore, it is essential to understand such interactions in order to exploit the opportunities for designing algorithms that are sensitive to natural input distributions.

A generic example of a parameterization is provided by the many well-known decision problems concerning graphs that take as input a graph G and a positive integer k. The parameter k appears to contribute to the complexity of such problems in two qualitatively different ways. GRAPH GENUS, MIN-CUT LINEAR ARRANGEMENT, VERTEX COVER, and FEEDBACK VERTEX SET FOR UNDIRECTED GRAPHS (see, for example, [22] for definitions) can all be solved in time $O(f(k)n^c)$, where c is a constant independent of k and $f(\cdot)$ is some (arbitrary) function. This "good behavior" is termed *fixed-parameter tractability* in the theory introduced in [14]. As is the case with polynomial-time complexity, the exponent c is typically small. One can equivalently define fixed-parameter tractability to mean solvability in time $O(f(k) + n^c)$, that is, with only an *additive* contribution from the parameter [13]. There is a rich collection of distinctive techniques for devising FPT algorithms (see [16, 18, 23, 10, 27]).

Contrasting complexity behavior is exhibited by the naturally parametrized problems such as CLIQUE, DOMINATING SET, and BANDWIDTH for which the best known algorithms have running times $O(n^{ck})$. These problems have been shown to be complete or hard for the various levels of the W hierarchy of parametrized complexity

$$W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P] \subseteq \cdots \subseteq XP,$$

which can be taken as evidence that they are unlikely to be fixed-parameter tractable [8]. With these problems, we seem to hit a natural "wall" requiring brute force effort, much as is typically the case with NP-complete problems. For example, essentially no better algorithm is known for the k-DOMINATING SET problem than checking all k-subsets.

As in the theory of NP-completeness, there are roughly two kinds of arguments that can be offered for believing that parametrized problems that are complete or hard for $W[1]$ are not likely to be fixed-parameter tractable. The first kind of argument is, roughly speaking, sociological. So many different kinds of problems stand or fall together that the combination of efforts expended unsuccessfully from the various vantages compels a belief in inherent intractability. The second kind of argument is some form of direct intuition concerning the nature of the computations that define the issue, e.g., nondeterministic as opposed to deterministic polynomial time.

For parametrized complexity, both kinds of arguments can be made. Although the amount of unsuccessful effort that has been expended in an attempt to show fixed-parameter tractability for $W[1]$-hard problems is much less than the total effort expended to date in attempting to develop polynomial-time algorithms for NP-complete problems, it is still considerable and accumulating. The parametrized complexity class $W[1]$ is particularly interesting and important for several reasons: there is a substantial list of natural and useful computational problems that are precisely $W[1]$-complete. This list includes CLIQUE, INDEPENDENT SET, VAPNIK–CHERVONENKIS DIMENSION, MONOTONE DATA COMPLEXITY FOR RELATIONAL DATABASES, SQUARE TILING, k-STEP DERIVATION FOR CONTEXT SENSITIVE GRAMMARS, m-LENGTH COMMON SUBSEQUENCE FOR k-SEQUENCES, and k-LENGTH POST CORRESPONDENCE, among other problems.

Direct intuition about $W[1]$ is also available. It is shown in [19, 12] that the k-step halting problem for nondeterministic Turing machines is $W[1]$-complete. This problem is formally defined as follows.

Problem: SHORT TURING MACHINE ACCEPTANCE
Instance: A nondeterministic Turing machine M and a positive integer k.
Question: Does M have a computation path accepting the empty string in at most k steps?
Parameter: k

This is a problem so generic and opaque that it is hard to imagine that there is any algorithm for it that radically improves on simply exploring the n-branching depth-k tree of allowed transitions exhaustively. This is essentially the same intuition as the belief that Cook's theorem provides a basis for the intractability of NP-complete problems.

For a definition of the $W[t]$ complexity classes and the fundamentals of parametrized complexity, we refer the reader to section 4 and [14, 15, 16, 18]. Here, we will briefly review the basic definitions of a parametrized problem and fixed-parameter tractability.

DEFINITION 6.1. *A* parametrized problem *is a set $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a fixed alphabet. For convenience, we can think of a parametrized problem as a subset L of $\Sigma^* \times N$, where N is the set of nonnegative integers.*

DEFINITION 6.2. *We say that parametrized problem L is (uniformly)* fixed-parameter tractable *if there is a constant α and an algorithm Φ such that Φ decides if $(x, k) \in L$ in time $f(k)|x|^\alpha$, where $f : N \to N$ is an arbitrary function.*

Let A and B be parametrized problems. We say that A is (uniformly many to 1) *reducible* to B if there is an algorithm Φ which transforms $(x, k) \in \Sigma^* \times N$ into $(x', g(k))$ in time $f(k)|x|^\alpha$, where $f, g : N \to N$ are arbitrary functions and α is a constant independent of k so that $(x, k) \in A$ if and only if $(x', g(k)) \in B$. Such an algorithm Φ may be called a parametric transformation. It is easy to see that if A reduces to B and B is fixed-parameter tractable, then so too is A.

REFERENCES

[1] M. AJTAI, *private communication*, May 1997.
[2] S. ARORA, L. BABAI, J. STERN, and Z. SWEEDYK, *The hardness of approximate optima in lattices, codes, and systems of linear equations*, in Proc. 34th Symposium on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 724–733.
[3] R. BALASUBRAMANIAN, M. R. FELLOWS, and V. RAMAN, *An Improved Fixed-Parameter Algorithm for Vertex Cover*, preprint, 1996.
[4] A. BARG, *Some new NP-complete coding problems,* Problemy Peredachi Informatsii, 30 (1994), pp. 23–28 (in Russian).
[5] E. R. BERLEKAMP, R. J. MCELIECE, and H. C. A. VAN TILBORG, *On the inherent intractability of certain coding problems*, IEEE Trans. Inform. Theory, 24 (1978), pp. 384–386.
[6] H. BODLAENDER, R. G. DOWNEY, M. R. FELLOWS, and H. T. WAREHAM, *The parametrized complexity of the longest common subsequence problem*, Theoret. Comput. Sci., 147 (1995), pp. 31–54.
[7] H. BODLAENDER AND M. R. FELLOWS, *On the complexity of k-processor scheduling*, Oper. Res. Lett., 18 (1995), pp. 93–98.
[8] H. BODLAENDER, M. R. FELLOWS, AND M. T. HALLETT, *Beyond NP-completeness for problems of bounded width: Hardness for the W hierarchy*, in Proc. 26th ACM Symposium on Theory of Computing, 1994, pp. 449–458.
[9] J. BRUCK AND M. NAOR, *The hardness of decoding linear codes with preprocessing,* IEEE Trans. Inform. Theory, 36 (1990), pp. 381–385.
[10] L. CAI, *Fixed-parameter tractability of graph modification problems for hereditary properties*, Inform. Process. Lett., to appear.
[11] L. CAI AND J. CHEN, *Fixed parameter tractability and approximability of NP-hard optimization problems*, in Proc. 2nd Israel Symposium on Theory of Computing Systems, 1993, pp. 118–126.

[12] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows, *On the parametrized complexity of short computation and factorization*, Arch. Math. Logic, to appear.
[13] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows, *Advice classes of parametrized tractability*, Ann. Pure Appl. Logic, 84 (1997), pp. 119–138.
[14] R. G. Downey and M. R. Fellows, *Fixed-parameter tractability and completeness* I*: Basic results*, SIAM J. Comput., 24 (1995), pp. 873–921.
[15] R. G. Downey and M. R. Fellows, *Fixed parameter tractability and completeness* II*: Completeness for $W[1]$*, Theoret. Comput. Sci., 141 (1995), pp. 109–131.
[16] R. G. Downey and M. R. Fellows, *parametrized computational feasibility*, in Feasible Mathematics, P. Clote and J. Remmel, eds., Birkhäuser, Cambridge, MA, 1995, pp. 219–244.
[17] R. G. Downey and M. R. Fellows, *Threshold dominating sets and an improved characterization of $W[2]$*, Theoret. Comput. Sci., 189 (1997), to appear.
[18] R. G. Downey and M. R. Fellows, *parametrized Complexity*, Springer-Verlag, Berlin, 1997, to appear.
[19] R. G. Downey, M. R. Fellows, B. M. Kapron, M. T. Hallett, and H. T. Wareham, *The parametrized complexity of some problems in logic and linguistics*, in Lecture Notes in Comput. Sci. 813, Springer-Verlag, Berlin, New York, 1994, pp. 89–100.
[20] R. G. Downey, M. R. Fellows, and K. R. Regan, *parametrized circuit complexity and the W hierarchy*, Theoret. Comput. Sci., 189 (1997), to appear.
[21] R. G. Downey, M. R. Fellows, and U. Taylor, *The parametrized complexity of relational database queries and an improved characterization of $W[1]$*, in Combinatorics, Complexity, and Logic, D. Bridges, C. Calude, J. Gibbons, S. Reeves, and I. Witten, eds., Springer-Verlag, Berlin, 1996, pp. 194–213.
[22] M. R. Garey and D. S. Johnson, *Computers and Intractability: Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
[23] H. Kaplan, R. Shamir, and R. E. Tarjan, *Tractability of parametrized completion problems on chordal and interval graphs: Minimum fill-in and DNA physical mapping*, in Proc. 35th Symposium on Found. Computer Science, 1994, pp. 780–891.
[24] J. Kratochvíl, *Regular codes in regular graphs are difficult*, Discrete Math., 133 (1994), pp. 191–205.
[25] J. Kratochvíl and M. Křivánek, *On the computational complexity of codes in graphs*, in Proceedings MFCS Karlovy Vary 1988, Lecture Notes in Comput. Sci. 324, Springer-Verlag, New York, Berlin, 1988, pp. 396–404.
[26] M. Sipser and D. A. Spielman, *Expander codes*, IEEE Trans. Inform. Theory, 42 (1996), pp. 1710–1722.
[27] M. Steel, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.
[28] J. Stern, *Approximating the number of error locations within a constant ratio is NP-complete*, in Lecture Notes in Comput. Sci., Springer-Verlag 673, New York, Berlin, 1993, pp. 325–331.
[29] R. M. Tanner, *A recursive approach to low-complexity codes*, IEEE Trans. Inform. Theory, 27 (1981), pp. 533–547.
[30] P. van Emde Boas, *Another NP-Complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice*, Tech. Report 81–04, Department of Mathematics, University of Amsterdam, Amsterdam, The Netherlands, 1980.
[31] A. Vardy, *Algorithmic complexity in coding theory and the minimum distance problem*, in Proc. 29th ACM Symposium on Theory of Computing, El Paso, TX, 1997, pp. 92–109.
[32] A. Vardy, *The intractability of computing the minimum distance of a code*, IEEE Trans. Inform. Theory, 43 (1997), pp. 1757–1766.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 571–589

TWO- AND HIGHER-DIMENSIONAL PATTERN MATCHING IN OPTIMAL EXPECTED TIME[*]

JUHA KÄRKKÄINEN[†] AND ESKO UKKONEN[†]

Abstract. Algorithms with optimal expected running time are presented for searching the occurrences of a two-dimensional $m \times m$ pattern P in a two-dimensional $n \times n$ text T over an alphabet of size c. The algorithms are based on placing in the text a static grid of test points, determined only by n, m, and c (not dynamically by earlier test results). Using test strings read from the test points the algorithms eliminate as many potential occurrences of P as possible. The remaining potential occurrences are separately checked for actual occurrences. A suitable choice of the test point set leads to algorithms with expected running time $O(n^2 \log_c m^2/m^2)$ using the uniform Bernoulli model of randomness. This is shown to be optimal by a generalization of a one-dimensional lower bound result by Yao. Experimental results show that the algorithms are efficient in practice, too. The method is also generalized for the k mismatches problem. The resulting algorithm has expected running time $O(kn^2 \log_c m^2/m^2)$, provided that $k \leq (m\lfloor m/\lceil \log_c m^2 \rceil \rfloor - 1)/2$. All algorithms need preprocessing of P which takes time and space $O(m^2)$. The text processing can be done on-line, using a rather small window. The algorithms easily generalize to d-dimensional matching for any d.

Key words. multidimensional matching, average case analysis, approximate matching

AMS subject classifications. 68P05, 68P10, 68Q20, 68Q25, 68U10

PII. S0097539794275872

1. Introduction.

The classical *pattern matching problem* for strings is to find all (exact or approximate) occurrences of a *pattern* P in a *text* T. In a one-dimensional case, P and T are strings over some alphabet Σ of size c. In a d-dimensional case, P and T are d-dimensional arrays over Σ. The applications are numerous.

The exact one-dimensional pattern matching is well understood. For two- and higher-dimensional versions, several algorithms have been proposed, too, mainly aiming at a good worst-case running time; see, e.g., [7, 6, 16, 26, 11, 2, 13, 22]. Recently, the two-dimensional problem has been solved in linear worst-case time with algorithms that are alphabet-independent and work in small additional space [10].

In the applications of one-dimensional matching, the Boyer–Moore algorithm [8], whose expected running time is "sublinear," is clearly the most successful; see, e.g., [15]. Also in higher dimensions, one should look at algorithms with good expected running time, not only the worst case. The first significant step in this direction was the algorithm of Baeza–Yates and Régnier [5] that finds the occurrences of an $m \times m$ pattern in an $n \times n$ text in expected time $O(n^2/m)$.

Results. In this paper, we give better algorithms that find exact occurrences of an $m \times m$ pattern in an $n \times n$ text in expected time $O(n^2 \log_c m^2/m^2)$ under the uniform Bernoulli model of random strings over Σ. Algorithms with the same expected performance have independently appeared in [17, 12, 23].

Our algorithms are not dimensionality specific. For a d-dimensional pattern of size m^d and text of size n^d, they run in expected time $O(n^d \log_c m^d/m^d)$.

[*]Received by the editors October 19, 1994; accepted for publication (in revised form) July 10, 1998; published electronically November 10, 1999. This work was supported by the Academy of Finland. A preliminary version of this paper appeared in *Proceedings of the Fifth Annual ACM–SIAM Symposium on Discrete Algorithms*, SIAM, Philadelphia, PA, 1994, pp. 715–723.

http://www.siam.org/journals/sicomp/29-2/27587.html

[†]Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland (Juha.Karkkainen@cs.Helsinki.FI, Esko.Ukkonen@cs.Helsinki.FI).

In [25], A. Yao confirmed a conjecture of Knuth [18] by proving a lower bound of $O(n \log_c m/m)$ for the expected time complexity of one-dimensional exact pattern matching. Yao's argument generalizes for d-dimensional matching and gives a lower bound $O(n^d \log_c m^d/m^d)$. Hence, our algorithms are the first algorithms for multidimensional matching with optimal expected running time. Experiments show that the algorithms are also fast in practice.

We also generalize our algorithm for approximate pattern matching. We consider the two-dimensional k *mismatches problem* (find all occurrences of P in T with $\leq k$ mismatching symbols) and give an algorithm with expected running time $O(kn^2 \log_c m^2/m^2)$. The algorithm requires that $k \leq (m\lfloor m/\lceil \log_c m^2 \rceil \rfloor - 1)/2$. Under this condition the expected running time is at most linear in $|T|$; previously Chang and Lawler [9] (see also [24]) have given an algorithm with similar properties for the k differences problem of one-dimensional approximate matching. The two-dimensional k mismatches problem has also been studied in [21, 19, 4, 3].

Overview. We next informally describe the very simple elimination idea behind our algorithms. Note first that the naive algorithm for pattern matching and its improvements such as the Knuth–Morris–Pratt algorithm [18] are based on a view of the problem that could be called *positive*: the algorithm tries to prove at each text location that there is an occurrence of P at that location.

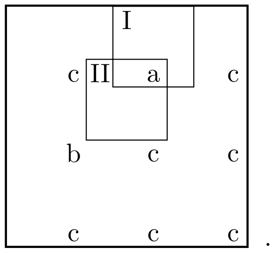
In applications, T often contains relatively few occurrences of P. Hence the *negative* view can fit the situation better than the positive one. A negative view algorithm tries to prove for each text location that there can *not* be an occurrence of P at that location. This forms the *elimination phase* of the algorithm. Whenever the elimination proof fails, the algorithm has found a potential occurrence of P that needs further consideration. A separate *checking phase* then examines whether or not a potential occurrence really matches P. This leads to fast algorithms because the elimination phase needs, on average, to examine only a small fraction of T.

The well-known Boyer–Moore algorithm [8] has this flavor. In this paper, we develop algorithms based on the negative view for two- and higher-dimensional pattern matching.

We explain the idea of our algorithms using two-dimensional P and T. Let

$$P: \boxed{\begin{array}{ccc} \mathrm{b} & \mathrm{b} & \mathrm{a} \\ \mathrm{b} & \mathrm{b} & \mathrm{b} \\ \mathrm{b} & \mathrm{a} & \mathrm{b} \end{array}} \qquad T: \boxed{\begin{array}{ccccccccc} \mathrm{a} & \mathrm{a} & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{b} \\ \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{b} \\ \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{c} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{c} \\ \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} \\ \mathrm{a} & \mathrm{a} & \mathrm{c} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{a} & \mathrm{a} \\ \mathrm{b} & \mathrm{c} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{c} & \mathrm{b} & \mathrm{b} & \mathrm{c} \\ \mathrm{a} & \mathrm{c} & \mathrm{c} & \mathrm{b} & \mathrm{c} & \mathrm{c} & \mathrm{a} & \mathrm{c} & \mathrm{c} \\ \mathrm{c} & \mathrm{c} & \mathrm{c} & \mathrm{b} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{a} & \mathrm{a} \\ \mathrm{c} & \mathrm{c} & \mathrm{c} & \mathrm{c} & \mathrm{a} & \mathrm{c} & \mathrm{b} & \mathrm{b} & \mathrm{c} \end{array}}\ .$$

We have to find all (I, J) such that $T[I + k - 1, J + h - 1] = P[k, h]$ for $1 \leq k, h \leq 3$. If we are lucky, we can eliminate all possible occurrences of P in T by examining only the entries $T[i, j]$ where $i = 3, 6, 9$; $j = 3, 6, 9$, because any possible occurrence of P in T must overlap at least one of these entries of T. The test entries in T are as follows:



As the first entry $T[3,3]$ equals c, and c does not occur in P at all, no occurrence of P in T can overlap $T[3,3]$. This proves that there can not be an occurrence of P inside subarray $T[1:5,1:5]$. Test entry $T[3,6]=a$ eliminates all other occurrences of P that overlap $T[3,6]$ except those that have the overlapping symbol a. The upper left-hand corners of these potential occurrences of P are $T[1,5]$ (marked I in the figure) and $T[3,4]$ (marked II), because $P[1,3]=P[3,2]=a$. The checking phase then finds that there is an occurrence of P at $T[1,5]$ but not at $T[3,4]$.

The fourth test entry $T[6,3]=b$ is least useful for elimination because b occurs 7 times in P. Hence, we are left with 7 potential occurrences of P that overlap $T[6,3]$. However, there is no reason to use only one entry of T in each elimination step. Taking $T[6,2]=c$ as an additional test entry would eliminate all possible occurrences of P that overlap $T[6,2]$ and $T[6,3]$ because substring cb does not occur in P.

Increasing the number of test entries used in each elimination step obviously leads to more extensive elimination but possibly takes more time. Therefore, we determine q such that, when each elimination step is based on q entries of T, the expected total running time of elimination and (naive) checking is minimized. It turns out that the minimum is given by $q \approx \log_{|\Sigma|}|P|$; this simply has the effect of using a superalphabet of size $\Theta(|P|)$.

The test entries will be distributed over T such that every potential occurrence of P in T covers at least q entries. We use a *static distribution* of test entries; this is a major difference to virtually all other pattern matching algorithms proposed in the literature. The grid of test entries is determined using only the size (and shape) of P, T and the underlying alphabet. The elimination phase becomes *oblivious*: the results of earlier elimination tests do not cause dynamic changes on later tests. This property helps in writing fast code for the algorithms, for both sequential and parallel environments.

The idea of using superalphabet of size $\Theta(|P|)$ in two-dimensional matching appears independently in [12, 17, 23]. All achieve the same asymptotic running time as our algorithms. Crochemore and Rytter [12, Sect. 12.7] sketch an algorithm which comes close to one of our algorithms, while the algorithms in [17, 23] differ in that they determine the test entries dynamically.

The above idea is formulated in section 2 as a generic pattern matching algorithm. The elimination of potential occurrences of P is based on reading test strings of length q from T. The strings are picked up according to a fixed sampling scheme. In section 3, we give an example scheme such that each test entry belongs to only one test string. To get the minimum total number of test entries, we use test strings whose shape has the smallest possible diameter. In section 4, we give a bit more efficient scheme in which each test entry belongs to several test strings. Section 5 reformulates the lower bound of Yao [25] for d-dimensional pattern matching. Section 6 describes the results of experiments comparing the running times of the algorithms of sections 3 and 4, the trivial algorithm, and an algorithm by Tarhio [23]. In section 7, we generalize the algorithm for the k mismatches problem. The elimination is based on $k + p$ nonoverlapping test strings; here p is a parameter used for optimizing the performance of the method.

2. General algorithm.

To keep the technical exposition simple we restrict ourselves to the two-dimensional case with square patterns and texts. The generalization of all our results to the d-dimensional pattern matching with rectangular patterns and texts is straightforward.

Let a two-dimensional $m \times m$ array $P = (p_{ij})$, $1 \le i, j \le m$, be the *pattern* and a two-dimensional $n \times n$ array $T = (t_{ij})$, $1 \le i, j \le n$, be the *text* over some alphabet Σ. Let c denote the size of the alphabet Σ.

We let integer q denote the *sample size*. Selecting a suitable q will minimize the expected running time of our algorithms. A sequence $Q = ((0,0), (i_1, j_1), (i_2, j_2), \ldots, (i_{q-1}, j_{q-1}))$ of q *disjoint* pairs of indexes is called a (two-dimensional) *template* of size q. The sequence always begins with the pair $(0,0)$ which is called the *origin* of the template. A template Q and a pair of indexes (i,j) define a *sample* $Q(i,j) = ((i,j), (i+i_1, j+j_1), \ldots, (i+i_{q-1}, j+j_{q-1}))$. Here Q gives the shape of the sample and (i,j) the location of the sample.

Pattern P is said to *contain* sample $Q(i,j)$ if all the pairs listed in $Q(i,j)$ refer to entries of P. Such sample $Q(i,j)$ determines a *test string* $s(P, Q(i,j))$ in P as $s(P, Q(i,j)) = p_{i,j} p_{i+i_1, j+j_1} \cdots p_{i+i_{q-1}, j+j_{q-1}}$.

A *sampling scheme* $\mathcal{Q}$ for a template Q, a pattern P, and a text T is a pair $(\mathcal{Q}_P, \mathcal{Q}_T)$ where *pattern sampling scheme* $\mathcal{Q}_P$ and *text sampling scheme* $\mathcal{Q}_T$ are subsets of all possible Q-shaped samples of P and T, respectively, i.e.,

$$\mathcal{Q}_P \subseteq \{Q(i,j) \mid P \text{ contains } Q(i,j)\} \text{ and}$$
$$\mathcal{Q}_T \subseteq \{Q(i,j) \mid T \text{ contains } Q(i,j)\}.$$

Let R be a potential occurrence of P in T (i.e., an $m \times m$ subarray of T) with upper left-hand corner at (i,j). A *witness* of R is a pair $(Q(i_P, j_P), Q(i_T, j_T)) \in \mathcal{Q}_P \times \mathcal{Q}_T$ such that $Q(i_T, j_T)$ is in R in the same position as $Q(i_P, j_P)$ is in P, i.e., $i_T - i + 1 = i_P$ and $j_T - j + 1 = j_P$. Witness $(Q(i_P, j_P), Q(i_T, j_T))$ is positive if $s(P, Q(i_P, j_P)) = s(T, Q(i_T, j_T))$; otherwise the witness is negative. A potential occurrence with a negative witness can not be a real occurrence of P in T.

Example 1. Let

$$Q: \begin{array}{|cc|}\hline 0 & 2 \\ \hline & \boxed{1} \end{array} \qquad P: \begin{array}{|ccc|}\hline \mathrm{b} & \mathrm{b} & \mathrm{c} \\ \mathrm{b} & \mathrm{c} & \mathrm{a} \\ \mathrm{a} & \mathrm{a} & \mathrm{a} \\ \hline \end{array} \qquad T: \begin{array}{|ccc:cc|}\hline \mathrm{c} & \mathrm{b} & \mathrm{a} & \mathrm{a} & \mathrm{b} \\ \hdashline \mathrm{b} & \mathrm{b} & \mathrm{c} & \mathrm{b} & \mathrm{c} \\ \mathrm{b} & \mathrm{c} & \mathrm{a} & \mathrm{c} & \mathrm{a} \\ \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} \\ \hdashline \mathrm{c} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{b} \\ \hline \end{array}\ .$$

The numbers in the template $Q = ((0,0), (1,1), (0,1))$ show the order of the entries. Let R be a potential occurrence of P in T with the upper left-hand corner at $(2,1)$ (marked with dashed boundaries above). If $Q(2,2) \in \mathcal{Q}_P$ and $Q(3,2) \in \mathcal{Q}_T$, then (Q(2,2),Q(3,2)) is a negative witness of R as $s(P, Q(2,2)) = caa \ne cba = s(T, Q(3,2))$. This proves that R is not an occurrence of P. All other possible witnesses of R would be positive.

We are now ready to give a general description of our algorithms. The idea is to eliminate potential occurrences with negative witnesses and check the remaining potential occurrences using, e.g., naive checking. We select a sampling scheme $\mathcal{Q}$ such that every potential occurrence has exactly one witness in $\mathcal{Q}$. If that one witness is positive, then the potential occurrence has no negative witnesses and must be checked. In other words, the potential occurrences worth checking are determined by finding all positive witnesses. Uninteresting potential occurrences are implicitly eliminated by ignoring negative witnesses.

ALGORITHM [Generic].

1. [Selection of q, Q and $\mathcal{Q}$]. Select suitable sample size q, template Q and sampling scheme $\mathcal{Q} = (\mathcal{Q}_P, \mathcal{Q}_T)$ for a given pattern P and text T over alphabet Σ of size c. Scheme $\mathcal{Q}$ must provide exactly one witness for each potential occurrence of P in T.
2. [Preprocessing of P]. For each pattern sample $Q(i, j) \in \mathcal{Q}_P$, evaluate the test string $s(P, Q(i, j))$. Construct a suitable data structure to quickly find all pattern samples with a given test string.
3. [Elimination]. For each text sample $Q(i_T, j_T) \in \mathcal{Q}_T$, scan T to obtain the string $s(T, Q(i_T, j_T))$ and find, using the data structure constructed in the preprocessing step, all pattern samples $Q(i_P, j_P)$ such that $s(P, Q(i_P, j_P)) = s(T, Q(i_T, j_T))$. For each matching pair $(Q(i_P, j_P), Q(i_T, j_T))$, execute the checking phase for the potential occurrence R with the upper left-hand corner at $(i_R, j_R) = (i_T - i_P + 1, j_T - j_P + 1)$.
4. [Checking]. Given a potential occurrence R with the upper left-hand corner at (i_R, j_R), check whether or not there really is an occurrence, that is, test whether or not $t_{i_R+I-1, j_R+J-1} = p_{I,J}$ for $1 \leq I, J \leq m$.

Selecting different sampling schemes in the generic algorithm gives different concrete algorithms. The simplest alternative is developed in section 3 where we use templates with small diameter to maximize the number of potential occurrences each text sample is contained in. Utilizing overlaps of text samples leads to an algorithm with faster pattern preprocessing and a more sophisticated (not necessarily slower) elimination phase. Such methods are described and analyzed in section 4.

The sampling schemes of both of these algorithms are designed to minimize the number of text positions inspected during elimination as the text is typically much larger than the pattern. For the same reason, our algorithms preprocess the pattern samples. In an application, where multiple patterns are to be searched in the same static text, it might be useful to reverse the handling of text and patterns. However, we will not follow this direction further in this paper.

3. Square templates.

The requirement of one witness per potential occurrence means that the number of different witnesses in $\mathcal{Q}$, $|\mathcal{Q}_P||\mathcal{Q}_T|$, approximately equals the number of potential occurrences of P in T.[1] This implies that one can minimize the number of text samples by maximizing the number of pattern samples. This is the main idea of the algorithm presented in this section. The idea is realized by using what we call *square templates*.

DEFINITION 3.1. *Template* $Q = ((0, 0), (i_1, j_1), \ldots, (i_{q-1}, j_{q-1}))$ *of size* q *is a square template iff* $0 \leq i_k, j_k \leq \lceil \sqrt{q} \rceil - 1$ *for* $1 \leq k \leq q - 1$.

A square sample (sample whose shape is defined by a square template) of size q is contained in a $\lceil \sqrt{q} \rceil \times \lceil \sqrt{q} \rceil$ subarray whose upper left-hand corner is the origin of the sample. It is easy to see that the square shape maximizes the number of possible pattern samples.[2] Optimal sample size q will be determined in the analysis of the

[1] The nonexactness is caused by witnesses near the text boundaries.

[2] Actually, for some q, we could fit the template in a $\lceil \sqrt{q} \rceil \times \lfloor \sqrt{q} \rfloor$ subarray.

algorithm.

Given a square template Q, the pattern sampling scheme $\mathcal{Q}_P$ now contains all possible Q-shaped pattern samples, that is,

$$\mathcal{Q}_P = \{Q(i,j) \mid 1 \leq i,j \leq m - \lceil\sqrt{q}\rceil + 1\}.$$

To get one witness per potential occurrence the text samples must be placed at the intersections of every $(m - \lceil\sqrt{q}\rceil + 1)$th row and column. This gives

$$\mathcal{Q}_T = \left\{ Q\left(i(m - \lceil\sqrt{q}\rceil + 1), j(m - \lceil\sqrt{q}\rceil + 1)\right) \,\middle|\, 1 \leq i,j \leq \left\lfloor \frac{n - \lceil\sqrt{q}\rceil + 1}{m - \lceil\sqrt{q}\rceil + 1} \right\rfloor \right\}.$$

Now each text sample is a component of the witness of $|\mathcal{Q}_P| = (m - \lceil\sqrt{q}\rceil + 1)^2$ different potential occurrences. For each text sample, we need to find all matching pattern samples. To do this as quickly as possible, we preprocess the pattern to get a data structure which we call *test dictionary*.

Let w be a string of length q over alphabet Σ. Let $A(w)$ be the set of origins of pattern samples whose test string is w. That is,

$$A(w) = \{(i,j) \mid w = s(P, Q(i,j)),\ Q(i,j) \in \mathcal{Q}_P\}.$$

$A(w)$ is called the set of *addresses* of w. The test dictionary $D(P, \mathcal{Q}_P)$ for pattern P and sampling scheme $\mathcal{Q}_P$ of P is a data structure supporting queries that for a given string $w \in \Sigma^q$ yield the set of addresses of w.

There are two obvious implementations of test dictionary $D(P, \mathcal{Q}_P)$. First, one can use a trie representing the test strings, with the addresses $A(w)$ associated with the leaf that corresponds to string w. The test strings and their addresses can be constructed by scanning P for each pattern sample. There are $\leq m^2$ samples, hence the total length of the test strings is $\leq m^2 q$, and the trie representing them can be built in time $O(m^2qc)$ or $O(m^2 q \log c)$, depending on whether direct indexing or balanced search trees are used for implementing the branches of the trie. The queries for $A(w)$ can be answered in time $O(q)$ or $O(q \log c)$, respectively (plus the size of $A(w)$).

The second possibility to implement $D(P, \mathcal{Q}_P)$ is to convert the test strings into (c-ary) integers, and use these integers as indices to an array of size c^q representing the addresses $A(w)$ for each w. The optimal value of q turns out to be small enough to keep c^q acceptable. The array initialization takes $O(c^q)$ time and the trivial method of pattern processing requires $O(m^2q)$. The queries can be answered in time $O(q)$.

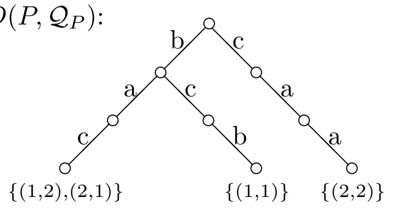
With a suitable ordering of the template entries and a little bookkeeping the pattern processing time can be reduced to $O(m^2)$. The template is ordered like text: row by row from top to bottom with each row ordered left to right. Individual rows of the samples can then be thought of as c-ary integers of $\lceil\sqrt{q}\rceil$ digits and the whole sample as a $c^{\lceil\sqrt{q}\rceil}$-ary integer of $\lceil\sqrt{q}\rceil$ digits (rows). Each sample row is a substring of a pattern row. The sample row values coming from a pattern row can be calculated with a single scan of the row in time $O(m)$ in the obvious way. Similarly, given all the sample row values the sample values can then be calculated by scanning the row values column by column. The operation is further complicated by the fact that, to have the sample size be exactly q, one of the sample rows may have to be shorter than $\lceil\sqrt{q}\rceil$. Despite the complexity of this method, it turned out to be faster, in practice, than the simpler $O(m^2q)$ method.

Example 2. Let Q, P, T and R be as in Example 1. Then Q is a square template and, using the sampling scheme of this section, we have

$$\mathcal{Q}_P = \{Q(1,1), Q(1,2), Q(2,1), Q(2,2)\},$$
$$\mathcal{Q}_T = \{Q(2,2), Q(2,4), Q(4,2), Q(4,4)\}.$$

The witness of R in $\mathcal{Q} = (\mathcal{Q}_P, \mathcal{Q}_T)$ is $(Q(1,2), Q(2,2))$. The addresses of strings $w \in \Sigma^3$ are now $A(bac) = \{(1,2),(2,1)\}$, $A(bcb) = \{(1,1)\}$, $A(caa) = \{(2,2)\}$, and $A(w) = \emptyset$ for all other w. The trie implementation of $D(P, \mathcal{Q}_P)$ would be as follows.

$D(P, \mathcal{Q}_P)$:



Analysis. We determine q such that the expected processing time of T (i.e., elimination + checking) is minimized under the assumption that T is a random text in the uniform Bernoulli model (each symbol of Σ can occur in a given position of T with probability $1/c$ independently of the content of other positions). We will next analyze the expected time for the elimination and checking of a fixed potential occurrence R of P in T.

Let $Q(i_T, j_T) \in \mathcal{Q}_T$ be the text sample contained in R. The elimination phase for R consists of evaluating string $w = s(T, Q(i_T, j_T))$ and finding $A(w)$ from $D(P, \mathcal{Q}_P)$. This takes time $O(q)$, i.e., is proportional to the number q of entries of T that have to be examined to get w. As $Q(i_T, j_T)$ is contained in $(m - \lceil\sqrt{q}\rceil + 1)^2$ potential occurrences of P, the elimination time amortized for R is proportional to $q/(m - \lceil\sqrt{q}\rceil + 1)^2$.

The analysis of the checking phase is based on the naive checking that compares the corresponding entries of P and R until the first mismatch or an entire occurrence of P is found. The expected number of comparisons is less than $\alpha = c/(c-1)$. However, the checking is activated only if the witness of R is positive. The probability of this event is $1/c^q$, and the expected number of comparisons in the checking phase is therefore less than α/c^q.

Now we choose q to minimize the total expected number of comparisons allocated for R,

$$\frac{q}{(m - \lceil\sqrt{q}\rceil + 1)^2} + \frac{\alpha}{c^q}. \tag{3.1}$$

Application of elementary calculus on (3.1) without the ceiling function gives the equation

$$q = \log_c m^2 - \log_c \frac{m^2(m+1)}{(m - \sqrt{q} + 1)^3} + \log_c \frac{c \ln c}{c-1}$$

for the optimal (real) value of q. This further implies that

$$\log_c m^2 - 1 < q < \log_c m^2 + \frac{1}{2},$$

when $m \geq 2$ and $c \geq 2$. Based on this we choose $q = \lceil \log_c m^2 \rceil$ (when $m = 1$, define $q = 1$).

Substituting this into (3.1) and noting that there are $\leq n^2$ potential occurrences gives the following bound for the expected number of symbols examined during the elimination and checking:

$$n^2 \left(\frac{\lceil \log_c m^2 \rceil}{\left(m - \left\lceil \sqrt{\lceil \log_c m^2 \rceil} \right\rceil + 1 \right)^2} + \frac{\alpha}{m^2} \right) \leq n^2 \frac{4 \lceil \log_c m^2 \rceil + \alpha}{m^2}$$
$$= O\left(\frac{n^2 \log_c m^2}{m^2} \right) = O\left(\frac{|T| \log_c |P|}{|P|} \right),$$

where we have used the fact that $m - \lceil \sqrt{\lceil \log_c m^2 \rceil} \rceil + 1 \geq m/2$ if $c \geq 2$ and $m \geq 2$.

We have shown the following result.

THEOREM 3.2. *The occurrences of an $m \times m$ pattern P in an $n \times n$ text T can be found using square templates of size $\lceil \log_c m^2 \rceil$ in expected time $O(n^2 \log_c m^2/m^2)$. The preprocessing of P takes time and space $O(m^2)$.*

When the pattern is large ($|P| = \omega(n\sqrt{\log_c n})$), the pattern preprocessing dominates the matching time. If desirable, the preprocessing time can be reduced by using only a part of the pattern in the preprocessing and elimination phases. The full pattern is used only in the checking phase. When the area of the pattern used in the preprocessing is $\Theta(n\sqrt{\log_c n})$, the expected total time becomes $O(n\sqrt{\log_c n})$.

It should be clear that the algorithm of this section generalizes to d-dimensional pattern matching for rectangular patterns P and texts T such that the expected running time is $O(|T| \log_c |P|/|P|)$ for processing T. Preprocessing P needs time $O(|P|)$.

4. Linear templates. In this section, we relax the requirement of a minimum number of text samples. Instead, we minimize the number of text entries belonging to samples by letting the samples share entries. This, together with an efficient method for reading the text samples and comparing them to pattern samples, leads to an algorithm with performance similar to the algorithm of previous section.

To make the reading of the text samples simple in the case of overlapping templates, we use *linear templates*. A linear template Q of size q defines the shape of samples to be an evenly spaced subsequence of q entries on a row of a two-dimensional array. Hence Q is of the form $Q = ((0,0), (0,h), (0,2h), \ldots, (0,(q-1)h))$ for some $h \geq 1$. Note that samples $Q(i,j)$ and $Q(i,j+h)$ share $q-1$ entries.

Text sampling scheme $\mathcal{Q}_T$ will consist of linear samples placed on every mth row, called a *test row*, such that they start from every hth entry. Hence the samples overlap each other as $Q(i,j)$ and $Q(i,j+h)$ above. This gives

$$\mathcal{Q}_T = \left\{ Q(im, jh) \,\middle|\, 1 \leq i \leq \left\lfloor \frac{n}{m} \right\rfloor, 1 \leq j \leq \left\lfloor \frac{n-m}{h} \right\rfloor + 1 \right\}.$$

The number of text entries belonging to samples in $\mathcal{Q}_T$ is now $\lfloor n/m \rfloor (\lfloor (n-m)/h \rfloor + q)$. To minimize this, we should maximize h. The upper bound for h is given by the fact that every potential occurrence must contain at least one text sample. This gives $h = \lfloor m/q \rfloor$. The pattern sampling scheme is

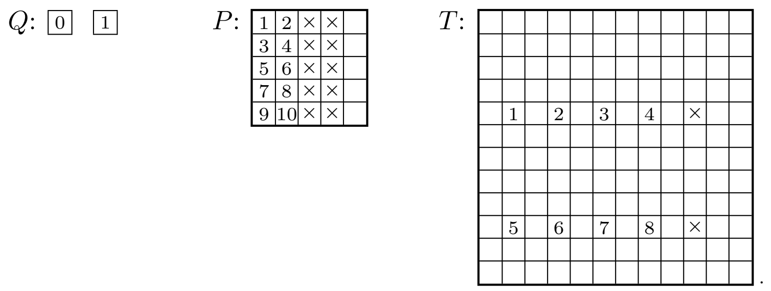
$$\mathcal{Q}_P = \{ Q(i,j) \mid 1 \leq i \leq m,\ 1 \leq j \leq h \}.$$

Note that there are no overlaps between pattern samples.

Example 3. Let P be a 5×5 pattern, T a 12×12 text and $Q = ((0,0),(0,2))$ a linear template of size 2. Using the above method, we get

$$\mathcal{Q}_P = \{Q(1,1), Q(1,2), Q(2,1), Q(2,2), Q(3,1), Q(3,2), Q(4,1), Q(4,2), Q(5,1), Q(5,2)\},$$
$$\mathcal{Q}_T = \{Q(5,2), Q(5,4), Q(5,6), Q(5,8), Q(10,2), Q(10,4), Q(10,6), Q(10,8)\}.$$

This is illustrated below:



The numbered entries in the pattern and the text are the origins of the samples. The entries marked with "×" are the other sample entries.

Utilizing the overlaps of text samples makes it possible to read each test string $s(T, Q(i,j))$, $Q(i,j) \in \mathcal{Q}_T$, in constant amortized time. For each test row, we read the sample entries from left to right and maintain a window of size q to hold the current string.

We also need to find the matching pattern samples in constant time. To achieve this we replace the trie of section 3 with an Aho–Corasick multipattern matching automaton $AC(P, \mathcal{Q}_P)$ (see [1]). The automaton can be built in the same asymptotic time as the trie; in fact $AC(P, \mathcal{Q}_P)$ is simply the trie augmented with the *failure*-transitions. The addresses of the test strings are attached to the states of $AC(P, \mathcal{Q}_P)$ that correspond to the test strings. As there are $m\lfloor m/q \rfloor$ samples of size q in $\mathcal{Q}_P$, the preprocessing time becomes $O(m^2)$; with c shown explicitly the bound becomes $O(m^2 c)$ or $O(m^2 \log c)$. This is smaller, by a factor of q, than the preprocessing time of the previous section, as there are now fewer pattern samples.

The method of treating strings as integers can be modified to work with the linear templates, too. During the scanning of T, we treat the characters as digits and maintain a window of q digits. The pattern samples can be read one at a time in $O(m^2)$ time. Otherwise, the method works as with square templates.

Analysis. As already mentioned, the number of text entries read during elimination is

$$\left\lfloor \frac{n}{m} \right\rfloor \left(\left\lfloor \frac{n-m}{h} \right\rfloor + q \right) = \left\lfloor \frac{n}{m} \right\rfloor \left\lfloor \frac{n-m+qh}{h} \right\rfloor \le \left\lfloor \frac{n}{m} \right\rfloor \left\lfloor \frac{n}{h} \right\rfloor \le \frac{n^2}{mh} = \frac{n^2}{m\lfloor m/q \rfloor},$$

which is also the running time of the elimination phase, as each entry can be processed in constant time. The expected running time of the checking phase is, as before, at most $n^2\alpha/c^q$. Thus, the total expected text processing time is now bounded by

$$n^2 \left(\frac{1}{m \left\lfloor \frac{m}{q} \right\rfloor} + \frac{\alpha}{c^q} \right).$$

Omitting the floor function, we can again apply calculus to find that the optimal (real) value of q is

$$q = \log_c m^2 + \log_c \frac{c \ln c}{c-1},$$

where $0 < \log_c \frac{c \ln c}{c-1} < \frac{1}{2}$ for $c \geq 2$. Thus we again choose $q = \lceil \log_c m^2 \rceil$ which leads to the bound[3]

$$n^2 \left(\frac{2 \log_c m^2 + \alpha}{m^2} \right) = O\left(\frac{n^2 \log_c m^2}{m^2} \right) = O\left(\frac{|T| \log_c |P|}{|P|} \right)$$

for the total expected number of examined symbols during elimination and checking.

THEOREM 4.1. *The occurrences of an $m \times m$ pattern P in an $n \times n$ text T can be found using overlapping linear templates of size $\lceil \log_c m^2 \rceil$ in expected time $O(n^2 \log_c m^2 / m^2)$. The preprocessing of P takes time and space $O(m^2)$.*

Again, the pattern preprocessing time dominates when $|P| = \omega(n\sqrt{\log_c n})$. As with the square template algorithm, the total expected time in such a case can be limited to $O(n\sqrt{\log_c n})$ by using only an area of size $\Theta(n\sqrt{\log_c n})$ of the pattern in the preprocessing.

The algorithm again easily generalizes to d-dimensional P and T. Note that, for any d, the generalized algorithm uses linear (one-dimensional) samples scanned with the Aho–Corasick automaton. In this way, the elimination phase of d-dimensional matching reduces directly to one-dimensional techniques.

For larger d, it may happen that $q = \lceil \log_c m^d \rceil > m$ in which case the straightforward generalization does not work any more.[4] An easy correction is to use superrows, consisting of two or more adjacent one-dimensional rows. As $2^{d-1}m \geq \log_c m^d$ for $d \geq 1$, $c \geq 2$, $m \geq 2$, a superrow with diameter 2 is always sufficient. Therefore this does not change the asymptotic behavior of the algorithm.

The algorithm of this section is quite similar to the (suboptimal) algorithm by Baeza–Yates and Régnier [5], the main difference being that we use test samples with fixed size and location.

5. Lower bound.

In [25], Yao proved a lower bound of $\Omega(n \log_c m/m)$ for the expected running time of one-dimensional pattern matching. In this section, we generalize Yao's result for d-dimensional hypercubic strings.

Let P be a pattern of size m^d and T a text of size n^d over alphabet Σ of size c. There exists some minimum number of entries of T that must be examined before any algorithm can be sure that all occurrences of P in T have been found. We define $g(P, n)$ to be the minimum of such numbers for pattern P over all texts of size n^d. Thus, $g(P, n)$ gives the best case lower bound of pattern matching for pattern P.

There exists patterns that are easy to match in the best case (e.g., patterns that do not contain all characters of the alphabet). However, the following theorem shows that such patterns are rare.

DEFINITION 5.1. *For $n \geq m > 0$, let*

$$f_1(m, n) = \left\lceil \frac{1}{d} \log_c \left(\frac{(n-m)^d}{\ln(m^d + 1)} + 2 \right) \right\rceil \quad \textit{and}$$

[3]Taking the floor function into account, $q' = \lfloor m / \lfloor m/q \rfloor \rfloor$ is always at least as good a choice as q, because $q' \geq q$ and $\lfloor m/q' \rfloor = \lfloor m/q \rfloor$.

[4]For $d = 2$ there is one such case, namely $m = 3$, $c = 2$ with $\log_2 3^2 \approx 3.17$. In this special case, we can simply set $q = 3$.

$$f_2(m,n) = \left\lfloor \frac{n}{2m} \right\rfloor^d \left\lceil \frac{1}{d} \log_c(m^d + 1) \right\rceil .$$

Define

$$f(m,n) = \begin{cases} f_1(m,n) & \textit{if } m \le n \le 2m, \\ f_2(m,n) & \textit{if } n > 2m. \end{cases}$$

THEOREM 5.2. *There exists a positive number b such that, for any $|\Sigma| = c \ge 2$ and $m > 0$, there exists a set of patterns $L \subseteq \Sigma^{m^d}$ satisfying*

$$|L| \ge \left(1 - \frac{1}{m^{9d}}\right) c^{m^d},$$

and

$$\textit{for each } P \in L,\ g(P,n) \ge bf(m,n) \textit{ for all } n \ge m.$$

Proof. The proof is a simple modification of Yao's proof in [25]. As the proof is quite long we will give here only a brief guide for the modification. Almost all that is required is to remember that we are now dealing with d-dimensional strings. For example, in Yao's Lemma 5.2 the text has to be divided into hypercubes of size $(2m)^d$ instead of strings of length $2m$. Similarly, we need to replace the occurrences of m, n, $n - m$ (denoted by d in Yao's paper), $2m$ and $\lfloor n/(2m) \rfloor$ throughout the proof with m^d, n^d, $(n-m)^d$, $(2m)^d$ and $\lfloor n/(2m) \rfloor^d$, respectively. Additionally, the terms of the form $xq^4 \ldots$ and $xq^{20} \ldots$ must have an extra factor of 2^d. This is balanced by the factor $1/d$ in the definitions of f_1 and f_2. $\square$

The above theorem shows that for the majority of patterns (i.e., for the patterns in L) even the best case complexity of pattern matching is

$$\Omega\left(\frac{n^d \log_c m^d}{m^d}\right),$$

for a fixed d. This implies that for a fixed d and T and random P, the expected running time of pattern matching is $\Omega\left(n^d \log_c m^d / m^d\right)$. Thus our algorithms for fixed-dimensional strings are optimal.

It is possible to have a single version of our algorithms work for all dimensions d. In such a case, the lower bound given by Theorem 5.2, with dependency on d shown explicitly, is

$$\Omega\left(\frac{1}{d2^d} \frac{n^d \log_c m^d}{m^d}\right),$$

leaving the upper and lower bounds separated by factor $d2^d$. Our conjecture is that the lower bound can be improved to $\Omega\left(n^d \log_c m^d / m^d\right)$ in this case, too, but that remains an open question.

6. Experimental results. We have conducted some experiments to find out how these algorithms work in practice and how they compare to each other and some other algorithms. The other algorithms in the experiments are the trivial algorithm (check naively each potential occurrence of P in T) and an asymptotically optimal algorithm by Tarhio [23]. Tarhio's algorithm represents the dynamic sampling approach as opposed to the static sampling schemes of our algorithms.

Table 6.1
Running times of the algorithms in milliseconds for $n = 1000$.

		Preprocessing			Text scanning			Total			
c	m	ST	LT	TDBM	ST	LT	TDBM	ST	LT	TDBM	TRIV
2	2	0.16	**0.10**	0.13	1593	**1254**	1539	1593	1254	1539	**1156**
2	4	0.27	**0.14**	0.15	531	**446**	565	531	**446**	565	1101
2	8	0.64	0.42	**0.24**	147	**140**	188	148	**140**	189	1064
2	16	2.30	**0.58**	0.67	46.1	**41.7**	49.6	48.4	**42.3**	50.3	1044
2	32	10.1	**1.64**	2.24	12.0	13.0	**11.8**	22.1	14.6	**14.1**	1014
2	64	44.9	**6.34**	8.69	3.01	4.59	**2.90**	47.9	**10.9**	11.6	945
2	128	202	**26.7**	42.0	0.77	1.19	**0.74**	203	**27.9**	42.7	820
2	256	1081	**107**	180	0.22	0.27	**0.20**	1082	**107**	180	594
256	2	0.61	**0.38**	0.50	300	**270**	374	300	**270**	374	524
256	4	0.78	**0.46**	0.53	84.6	**77.7**	110	85.4	**78.1**	111	522
256	8	1.23	0.57	**0.56**	28.9	**27.0**	32.8	30.1	**27.6**	33.4	519
256	16	3.19	0.94	**0.80**	13.1	12.6	**11.2**	16.3	13.6	**12.0**	511
256	32	96.9	**54.0**	83.3	2.21	4.34	**2.35**	99.1	**58.3**	85.7	496
256	64	144	**59.4**	88.5	0.53	0.72	**0.50**	144	**60.1**	89.0	462
256	128	331	**80.9**	107	0.17	0.17	**0.12**	331	**81.1**	107	402
256	256	1367	**166**	182	0.10	0.06	**0.04**	1367	**166**	182	293

Both the square template algorithm (ST) and the linear template algorithm (LT) use the method of converting strings to integers. Also Tarhio's algorithm (TDBM = two-dimensional Boyer–Moore) uses a similar method; in this sense, these algorithms are very comparable. The pattern preprocessing of ST is done using the more complicated (but faster) $O(m^2)$ method. All of these algorithms try to choose the parameters in the pattern preprocessing phase so that the running time of the text scanning phase is minimized. The algorithms were written and carefully optimized using C and were run on a Sun 4 workstation.

Table 6.1 shows the results of experiments using independently generated random texts and patterns. Text size is 1000×1000 in all of the tests, and pattern and alphabet sizes vary. The times (given in milliseconds) are averages of seven tests with different patterns.

The table gives the pattern preprocessing and text scanning times separately; the total time is the sum of these. The total time for the trivial algorithm (TRIV) is also its scanning time, as it doesn't have preprocessing. The times do not include the reading of the text and the pattern from disk.

The trivial algorithm is clearly inferior to other algorithms for all except the smallest patterns. The ST algorithm has poor preprocessing times due to the complicated preprocessing method. Otherwise the algorithms perform similarly.

The scanning times are the most interesting here as that is what the algorithms try to minimize. Figure 6.1 compares the scanning times of the algorithms. The times are shown relative to $\log_c m^2/m^2$ to bring the times with different pattern sizes to the same scale.

As the pattern size increases, the total time decreases at first but starts to increase when the pattern preprocessing starts to dominate. Both in theory and in practice, the minimum is at $m^2 \approx n\sqrt{\log_c n}$. As suggested earlier, for large patterns the total time can be reduced by using only a part of the pattern in the preprocessing phase. The algorithms LT, ST, and TDBM can all be modified in this way. Table 6.2 shows the running times for the modified LT algorithm for alphabet size 2. Figure 6.2 compares

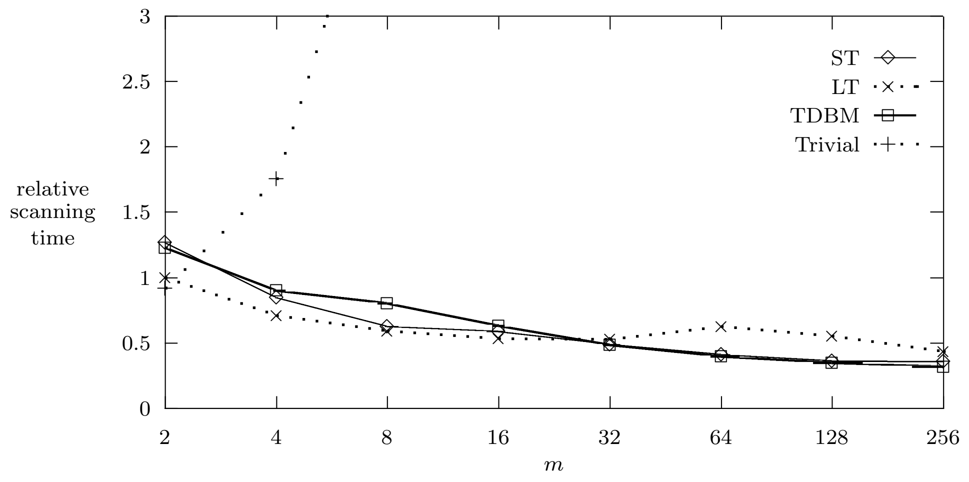


FIG. 6.1. *Scanning times relative to $\log_c m^2/m^2$ scaled so that the scanning time of LT is 1 when $m = 2$. The alphabet size c is 2.*

TABLE 6.2
Running times of LT with total time minimization in milliseconds for $n = 1000$ and $c = 2$.

m	Prepr	Scan	Total
2	0.10	1253	1253
4	0.14	446	447
8	0.42	139	140
16	0.58	41.8	42.4
32	1.63	13.0	14.6
64	2.83	5.95	8.78
128	2.95	5.31	8.25
256	3.00	3.49	6.49

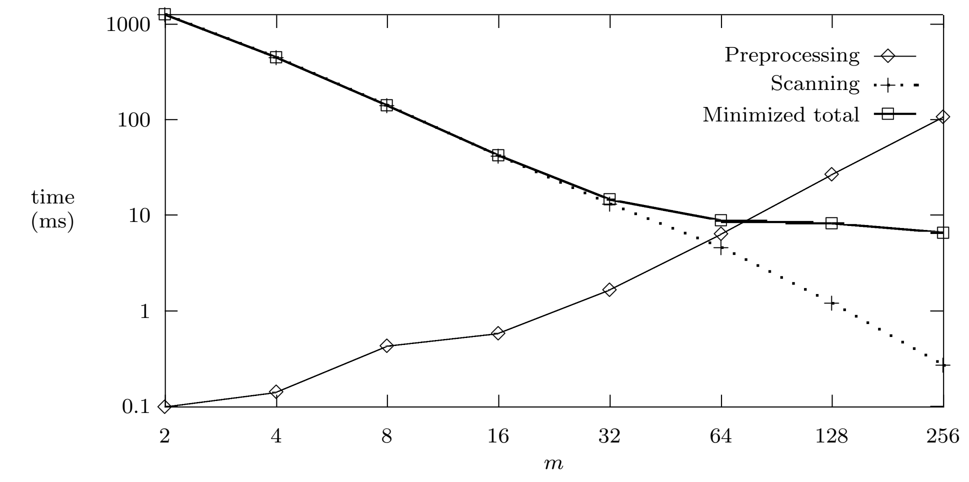


FIG. 6.2. *The minimized total running time of LT compared to the preprocessing and scanning times of basic LT.*

the minimized total running time to the preprocessing and scanning times of the basic algorithm that tries to minimize scanning time.

7. The k mismatches case.

The *k mismatches problem* asks for finding approximate occurrences of P with at most k mismatches, that is, occurrences that match with P except possibly for $\leq k$ symbols. In this section, we generalize the algorithms of sections 3 and 4 to the k mismatches problem. After the publication of an earlier version of this paper, Park [20] presented some improvements to the approximate matching algorithms and their analysis. This section includes some small changes to the earlier version inspired by Park's improvements.

One of the requirements of the algorithms in the previous sections was that every potential occurrence have exactly one witness. The k mismatches case, $k \geq 1$, can be handled by increasing the number of witnesses per potential occurrence as shown by the following lemma.

LEMMA 7.1. *Let potential occurrence R have $k+p$ disjoint witnesses. If $< p$ of them are positive, then R can not be an occurrence of P with $\leq k$ mismatches.*

Suggested by the lemma, the sampling scheme $\mathcal{Q}$ has to be such that every potential occurrence has $k+p$ nonoverlapping witnesses of size q. Parameters q and p will be determined in the analysis; note that to get optimal performance, we cannot simply choose $p = 1$.

The elimination phase now maintains a counter, initially equal to 0, for every potential occurrence R. When a positive witness of R is found, the counter of R is increased by one. When the counter achieves value p, R is transmitted to the checking phase. The checking phase naively compares R and P until $k+1$ mismatches are found or the whole pattern has been compared.

To avoid initializing all the $\Theta(n^2)$ counters, we do not create a counter until its first incrementation. The counters can be accessed in constant expected time using, e.g., hashing.

We will now show how to modify the sampling schemes of sections 3 and 4 to have $k+p$ witnesses for each potential occurrence.

Square templates. The text sampling scheme described in section 3 consists of square samples placed in the form of a regular grid such that every potential occurrence contains exactly one whole sample. If we take $k+p$ copies of such a sample grid, each translated such that there are no overlaps, we have a text sampling scheme $\mathcal{Q}_T$ where each potential occurrence contains exactly $k+p$ disjoint text samples. When we define the pattern sampling scheme $\mathcal{Q}_P$ to be the same as in section 3, we have the desired sampling scheme $\mathcal{Q} = (\mathcal{Q}_P, Q_T)$ with $k+p$ witnesses per potential occurrence.

The method works as long as we can keep the text samples disjoint. This means that we must have

$$k + p \leq \left\lfloor \frac{m - \lceil\sqrt{q}\rceil + 1}{\lceil\sqrt{q}\rceil} \right\rfloor^2 . \tag{7.1}$$

Linear templates. Using the linear templates of section 4, there are two ways to increase the number of witnesses. First, we can increase the number of test rows in $\mathcal{Q}_T$ so that every potential occurrence contains u test rows. If $k+p \leq m$, we can select $u = k+p$ and we are done. In the case $k+p > m$, every segment of length m of every test row in $\mathcal{Q}_T$ will contain more than one, say v, disjoint samples. This is achieved by selecting a step size $h = \lfloor m/vq \rfloor$. The text sampling scheme can now be defined as

$$\mathcal{Q}_T = \Big\{ Q(im-u+r, jh) \;\Big|\; 1 \le i \le \left\lfloor \frac{n-u+1}{m} \right\rfloor, \\ 1 \le j \le \left\lfloor \frac{n-m}{h} \right\rfloor + 1 + (v-1)q,\ 1 \le r \le \min\{u, n-im+u\} \Big\}.$$

The corresponding pattern sampling scheme is

$$\mathcal{Q}_P = \{ Q(i, j+rqh) \mid 0 \le r \le v-1, 1 \le i \le m,\ 1 \le j \le h \}.$$

This sampling scheme $\mathcal{Q} = (\mathcal{Q}_P, \mathcal{Q}_T)$ gives uv disjoint witnesses per potential occurrence.

For some (large) values of q and $k+p$ it may not be possible to select u and v such that the optimum value $k+p$ of uv is achieved. However, by selecting

$$v = \left\lceil \frac{k+p}{m} \right\rceil \text{ and } u = \left\lceil \frac{k+p}{v} \right\rceil,$$

we can get close, as then

$$k+p \;\le\; uv = \left\lceil \frac{k+p}{v} \right\rceil v \le k+p+v-1 = k+p+\left\lceil \frac{k+p}{m} \right\rceil - 1 < (k+p)\left(1+\frac{1}{m}\right).$$

The upper bounds for u and v are given by $u \le m$ and $vq \le m$. Thus, the above selections can be done as long as

$$k+p \le m \left\lfloor \frac{m}{q} \right\rfloor. \tag{7.2}$$

Analysis. The running time (without pattern preprocessing) is the sum of three components:
$A =$ the number of entries of T examined during the elimination,
$B =$ the number of updates of the counters for potential occurrences, and
$C =$ the number of symbols compared during the checking.
For both of the above sampling schemes we have

$$A \approx \frac{n^2(k+p)q}{m^2}. \tag{7.3}$$

The expected value $\bar{B}$ of B is

$$\bar{B} \le \frac{n^2(k+p)}{c^q} \tag{7.4}$$

because there are at most n^2 potential occurrences with $k+p$ witnesses each, and a witness is positive with probability $1/c^q$.

The third component C needs a more careful analysis. First, we denote by C' the number of character comparisons during naive checking of a randomly chosen potential occurrence R. As C' is a Pascal distributed random variable with parameters $k+1$ and $\frac{c-1}{c}$, we get

$$\bar{C'} = (k+1)\frac{c}{c-1} \le 2(k+1).$$

However, a candidate reaching the checking phase is not random: it is known to contain p positive witnesses. This increases the number of comparisons by at most pq.

Next we analyze the probability, denoted by H, that the checking is started for a fixed R. As H is the probability that at most k of the $k+p$ witnesses of R are negative, we have

$$H = \sum_{i=0}^{k} \binom{k+p}{i} \left(\frac{1}{c^q}\right)^{k+p-i} \left(1 - \frac{1}{c^q}\right)^{i}.$$

Noting that

$$\binom{k+p}{i} = \frac{\binom{k+p}{k}\binom{k}{i}}{\binom{k+p-i}{k-i}},$$

we can rewrite H as follows

$$\begin{aligned} H &= \frac{\binom{k+p}{k}}{c^{qp}} \sum_{i=0}^{k} \frac{\binom{k}{i}}{\binom{k+p-i}{k-i}} \left(\frac{1}{c^q}\right)^{k-i} \left(1 - \frac{1}{c^q}\right)^{i} \\ &\le \frac{\binom{k+p}{k}}{c^{qp}} \sum_{i=0}^{k} \binom{k}{i} \left(\frac{1}{c^q}\right)^{k-i} \left(1 - \frac{1}{c^q}\right)^{i} \\ &= \frac{\binom{k+p}{k}}{c^{qp}}. \end{aligned}$$

As there are at most n^2 potential occurrences, we now have

$$\bar{C} \le n^2 H(\bar{C}' + pq) \le \frac{n^2\binom{k+p}{k}}{c^{qp}}(2k+2+pq). \tag{7.5}$$

Select finally the parameters q and p satisfying

$$q = \lceil \log_c m^2 \rceil \quad \text{and} \tag{7.6}$$

$$p = 1 + \left\lceil \log_{c^q} \binom{k+p}{k} \right\rceil. \tag{7.7}$$

The value of p needs a little more analysis. We first note that

$$1 \le 1 + \left\lceil \log_{c^q} \binom{k+1}{k} \right\rceil \quad \text{and} \quad k+1 \ge 1 + \left\lceil \log_{c^q} \binom{2k+1}{k} \right\rceil.$$

The latter inequality holds because

$$\log_{c^q} \binom{2k+1}{k} = \log_{c^q} \left(\frac{2k+1}{k} \frac{2k}{k-1} \cdots \frac{k+1}{1} \right) \le \log_{c^q}(k+1)^k = k \log_{c^q}(k+1) \le k,$$

with the reasonable assumption that $k+1 \le m^2 \le c^q$. Given the above and the fact that the right-hand side of (7.7) is an integer valued nondecreasing function of p, we are guaranteed that there exists an integer value of p satisfying (7.7) in the range $[1, k+1]$. If there are many such values, we select the smallest of them. The value can, in practice, be found by trying with $p = 1, 2, \ldots$ until (7.7) is satisfied. Note that if $k = 0$, then $p = 1$, and the algorithm reduces to the exact matching algorithm.

Substituting (7.6), (7.7), and the fact that $p \leq k+1$ into (7.3), (7.4), and (7.5) gives

$$A, \bar{C} = O\left(\frac{n^2(k+1)\log_c m^2}{m^2}\right) \text{ and}$$
$$\bar{B} = O\left(\frac{n^2(k+1)}{m^2}\right).$$

Hence, the expected running time of the algorithm is

$$A + \bar{B} + \bar{C} = O\left(\frac{n^2(k+1)\log_c m^2}{m^2}\right).$$

As A is the most significant component in the running time, the only way to improve the selection of parameters q and p would be to decrease q. However, any asymptotically significant decrease in q would increase $\bar{B}$ past A. Thus, the above selection of the parameters is asymptotically optimal.

The maximum value of k that the algorithm can handle is limited by (7.1) with square templates and by (7.2) with linear templates. The linear template algorithm is less restrictive in most cases. Substituting $q = \lceil \log_c m^2 \rceil$ and $p \leq k+1$ into (7.2) gives the bound

$$k \leq \frac{m\left\lfloor \frac{m}{\lceil \log_c m^2 \rceil} \right\rfloor - 1}{2}$$

for the maximum value of k. For larger patterns, the actual upper bound for k in the linear template algorithm is close to $m^2/\lceil \log_c m^2 \rceil$.

If the counters are destroyed as soon as they can no longer be increased, the space taken by the counters is $O(kn/m)$ on average and $O(nm)$ in the worst case. The space requirement can be reduced with the following technique. Divide the text into slices of $2m-2$ rows that overlap each other by $m-1$ rows, and search each slice separately. All potential occurrences belong to exactly one slice, so no occurrence will be missed or found twice. The space requirement is now reduced to $O(k)$ on average and $O(m^2)$ in the worst case. The number of text samples read at most doubles, so the asymptotic time requirement remains the same. The number of samples can still be reduced by using wider slices, which increases the space requirement, and/or by optimizing the sampling scheme for the slices, which has been done by Park [20].

We have shown the following theorem.

THEOREM 7.2. *Let $k \leq (m\lfloor m/\lceil \log_c m^2 \rceil \rfloor - 1)/2$. Then the k mismatches problem for two-dimensional $m \times m$ pattern P and $n \times n$ text T can be solved in expected time $O((k+1)n^2 \log_c m^2/m^2)$ and additional space $O(m^2)$. The preprocessing of P takes time and space $O(m^2)$.*

The algorithm requires that $k = O(m^2/\log_c m^2)$. Substituting this into the bound of the above theorem shows that whenever the algorithm works, it is at most linear in $|T|$. Chang and Lawler [9] have obtained similar results for the one-dimensional k differences problem.

8. Concluding remarks. There are many possibilities for practical fine-tuning of our algorithms. For example, in the algorithm of section 3 it is possible to combine the elimination and checking phases using the spiral-shaped test strings of Gonnet [14]. The text processing in all algorithms can be performed with a window of size $O(m^2)$ into the text.

We ignored worst-case considerations in this paper. It is possible to use some linear worst-case algorithm for the checking phase of our algorithms such that the total time stays $O(n^2)$. However, it is not clear that this would be useful in practice. More important is that the obliviousness of the elimination phase makes parallel implementations of our algorithms simple.

The expected time $O(|T|\log_c |P|/|P|)$ of our exact matching algorithms was shown to be optimal for d-dimensional strings when d is considered constant, but not when d is variable. It remains an open problem to either remove the dependence on d from the lower bound or develop an algorithm that is better than ours for variable d. Another open problem is whether our approximate matching algorithms are optimal in the expected case.

Acknowledgment. We would like to thank Jorma Tarhio for his useful comments and his help in implementing the TDBM-algorithm.

REFERENCES

[1] A. V. Aho and M. J. Corasick, *Efficient string matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[2] A. Amir, G. Benson, and M. Farach, *An alphabet independent approach to two-dimensional pattern matching*, SIAM J. Comput., 23 (1994), pp. 313–323.

[3] A. Amir and M. Farach, *Efficient 2-dimensional approximate matching of non-rectangular figures*, in Proceedings of the 2nd ACM–SIAM Symposium on Discrete Algorithms, ACM–SIAM, 1991, pp. 212–223.

[4] A. Amir and G. M. Landau, *Fast parallel and serial multidimensional approximate pattern matching*, Theoretical Comput. Sci., 81 (1991), pp. 97–115.

[5] R. Baeza-Yates and M. Régnier, *Fast two-dimensional pattern matching*, Inform. Process. Lett., 45 (1993), pp. 51–57.

[6] T. P. Baker, *A technique for extending rapid exact-match string matching to arrays of more than one dimension*, SIAM J. Comput., 7 (1978), pp. 533–541.

[7] R. S. Bird, *Two dimensional pattern matching*, Inform. Process. Lett., 6 (1977), pp. 168–170.

[8] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[9] W. I. Chang and E. L. Lawler, *Approximate string matching in sublinear expected time*, in Proceedings of the 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1990, pp. 116–124.

[10] M. Crochemore, L. Gąsieniec, W. Plandowski, and W. Rytter, *Two-dimensional pattern matching in linear time and small space*, in Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science, vol. 900 of LNCS, Springer, New York, 1995, pp. 181–192.

[11] M. Crochemore, L. Gąsieniec, and W. Rytter, *Two-dimensional pattern matching by sampling*, Inform. Process. Lett., 46 (1993), pp. 159–162.

[12] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, 1994.

[13] Z. Galil and K. Park, *Alphabet-independent two-dimensional witness computation*, SIAM J. Comput., 25 (1996), pp. 907–935.

[14] G. H. Gonnet, *Efficient searching of text and pictures* (*extended abstract*), Technical Report OED-88-02, Centre for the New OED, University of Waterloo, Waterloo, Ontario, Canada, 1988.

[15] A. Hume and D. M. Sunday, *Fast string searching*, Software—Practice and Experience, 21 (1991), pp. 1221–1248.

[16] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Dev., 31 (1987), pp. 249–260.

[17] J. Y. Kim and J. Shawe-Taylor, *Fast expected two dimensional pattern matching*, in Proceedings of the 1st South American Workshop on String Processing, 1993, pp. 77–92.

[18] D. E. Knuth, J. H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[19] K. Krithivasan and R. Sitalakshmi, *Efficient two-dimensional pattern matching in the presence of errors*, Inform. Sci., 43 (1987), pp. 169–184.

[20] K. Park, *Analysis of two-dimensional approximate pattern matching algorithms* (*Note*), Theoret. Comput. Sci., 201 (1998), pp. 263–273.

[21] S. Ranka and T. Heywood, *Two-dimensional pattern matching with k mismatches*, Pattern Recognition, 24 (1991), pp. 31–40.

[22] M. Régnier and L. Rostami, *A simple (but optimal) 2d-witness algorithm*, in Proceedings of the 3rd South American Workshop on String Processing, Carleton University Press, 1996, pp. 243–256.

[23] J. Tarhio, *A sublinear algorithm for two-dimensional string matching*, Pattern Recognition Letters, 17 (1996), pp. 833–838.

[24] E. Ukkonen, *Approximate string-matching with q-grams and maximal matches*, Theoret. Comput. Sci., 92 (1992), pp. 191–211.

[25] A. C. Yao, *The complexity of pattern matching for a random string*, SIAM J. Comput., 8 (1979), pp. 368–387.

[26] R. F. Zhu and T. Takaoka, *A technique for two-dimensional pattern matching*, Comm. ACM, 32 (1989), pp. 1110–1120.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 590–599

KOLMOGOROV RANDOM GRAPHS AND THE INCOMPRESSIBILITY METHOD*

HARRY BUHRMAN†, MING LI‡, JOHN TROMP§, AND PAUL VITÁNYI¶

Abstract. We investigate topological, combinatorial, statistical, and enumeration properties of finite graphs with high Kolmogorov complexity (almost all graphs) using the novel incompressibility method. Example results are (i) the mean and variance of the number of (possibly overlapping) ordered labeled subgraphs of a labeled graph as a function of its randomness deficiency (how far it falls short of the maximum possible Kolmogorov complexity) and (ii) a new elementary proof for the number of unlabeled graphs.

Key words. Kolmogorov complexity, incompressibility method, random graphs, enumeration of graphs, algorithmic information theory

AMS subject classifications. 68Q30, 05C80, 05C35, 05C30

PII. S0097539797327805

1. Introduction.

The incompressibility of individual random objects yields a simple but powerful proof technique. The incompressibility method [9] is a new general purpose tool and should be compared with the pigeon hole principle or the probabilistic method. Here we apply the incompressibility method to randomly generated graphs and "individually random" graphs—graphs with high Kolmogorov complexity.

In a typical proof using the incompressibility method, one first chooses an individually random object from the class under discussion. This object is effectively incompressible. The argument invariably says that if a desired property does not hold, then the object can be compressed. This yields the required contradiction. Since a randomly generated object is *with overwhelming probability* individually random and hence incompressible, one usually obtains the property with high probability.

Results. We apply the incompressibility method to obtain combinatorial properties of graphs with high Kolmogorov complexity. These properties are parametrized in terms of a "randomness deficiency" function.[1] This can be considered as a parametrized version of the incompressibility method. In section 2 we show that for every labeled graph on n nodes with high Kolmogorov complexity (also called "Kolmogorov random graph" or "high-complexity graph"), the node degree of every vertex is about $n/2$ and there are about $n/4$ node-disjoint paths of length 2 between every

*Received by the editors September 23, 1997; accepted for publication (in revised form) October 9, 1998; published electronically November 10, 1999. A partial preliminary version appeared in *Proceedings of the Conference on Compression and Complexity of Sequences*, Salerno, Italy, 1997, IEEE Press, Piscataway, NJ.

http://www.siam.org/journals/sicomp/29-2/32780.html

†CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (buhrman@cwi.nl). The work of this author was partially supported by the European Union through NeuroCOLT ESPRIT Working Group 8556, and by NWO through NFI Project ALADDIN NF 62-376 and SION grant 612-34-002.

‡Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (mli@wh.math.uwaterloo.ca). The work of this author was supported in part by the NSERC Operating grant OGP0046506, CITO, a CGAT grant, and the Steacie Fellowship.

§CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (tromp@cwi.nl).

¶CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (paulv@cwi.nl). The work of this author was partially supported by the European Union through NeuroCOLT ESPRIT Working Group 8556 and by NWO through NFI Project ALADDIN NF 62-376.

[1]Randomness deficiency measures how far the object falls short of the maximum possible Kolmogorov complexity. It is formally defined in Definition 4.

pair of nodes. In section 2.2, we analyze "normality" properties of Kolmogorov random graphs. In analogy with infinite sequences one can call an infinite labeled graph normal if each finite ordered labeled subgraph of size k occurs in the appropriate sense (possibly overlapping) with limiting frequency $2^{-\binom{k}{2}}$. It follows from the Martin–Löf theory of effective tests for randomness [14] that individually random (high complexity) infinite labeled graphs are normal. Such properties cannot hold precisely for finite graphs, where randomness is necessarily a matter of degree: We determine close quantitative bounds on the normality (frequency of subgraphs) of high-complexity finite graphs in terms of their randomness deficiency.

Denote the number of unlabeled graphs on n nodes by g_n. In section 2.3 we demonstrate the use of the incompressibility method and Kolmogorov random graphs by providing a new elementary proof that $g_n \sim 2^{\binom{n}{2}}/n!$. This has previously been obtained by more advanced methods [12]. Moreover, we give a good estimate of the error term. Part of the proof involves estimating the order (number of automorphisms) $s(G)$ of graphs G as a function of the randomness deficiency of G. For example, we show that labeled graphs with randomness deficiency appropriately less than n are rigid (have but one automorphism: the identity automorphism).

Related work. Several properties (high degree nodes, diameter 2, rigidity) have also been proven by traditional methods to hold with high probability for randomly generated graphs [5, 4]. We provide new proofs for these results using the incompressibility method. They are actually proved to hold for the definite class of Kolmogorov random graphs—rather than with high probability for randomly generated graphs.

In [10] (also [9]) Li and Vitányi investigated topological properties of labeled graphs with high Kolmogorov complexity and proved them using the incompressibility method to compare ease of such proofs with the probabilistic method [7] and the entropy method.

In [8] it was shown that every labeled tree on n nodes with randomness deficiency $O(\log n)$ has maximum node degree of $O(\log n/\log\log n)$. Analysis of Kolmogorov random graphs was used to establish the total interconnect length of Euclidean (real-world) embeddings of computer network topologies [15] and the size of compact routing tables in computer networks [6]. Infinite binary sequences that asymptotically have equal numbers of 0s and 1s and, more generally, where every block of length k occurs (possibly overlapping) with frequency $1/2^k$ were called "normal" by E. Borel [2]. References [9, 11] investigate the quantitative deviation from normal as a function of the Kolmogorov complexity of a finite binary string. Here we consider an analogous question for Kolmogorov random graphs.[2] Finally, there is a close relation and genuine difference between high-probability properties and properties of incompressible objects; see [9, Section 6.2].

1.1. Kolmogorov complexity.

We use the following notation. Let A be a finite set. By $d(A)$ we denote the *cardinality* of A. In particular, $d(\emptyset) = 0$. Let x be a finite binary string. Then $l(x)$ denotes the *length* (number of bits) of x. In particular, $l(\epsilon) = 0$, where ϵ denotes the *empty word*.

Let $x, y, z \in \mathcal{N}$, where $\mathcal{N}$ denotes the natural numbers. Identify $\mathcal{N}$ and $\{0,1\}^*$

[2]There are some results along these lines related to randomly generated graphs, but as far as the authors could ascertain (consulting Alan Frieze, Svante Janson, and Andrzej Rucinski around June 1996) such properties have not been investigated in the same detail as here. See, for example, [1, pp. 125–140]. But note that pseudorandomness also is different from Kolmogorov randomness.

according to the correspondence

$$(0,\epsilon),(1,0),(2,1),(3,00),(4,01),\ldots.$$

Hence, the length $l(x)$ of x is the number of bits in the binary string or number x. Let $T_0, T_1, \ldots$ be a standard enumeration of all Turing machines. Let $\langle\cdot,\cdot\rangle$ be a standard one-to-one mapping from $\mathcal{N} \times \mathcal{N}$ to $\mathcal{N}$, for technical reasons chosen such that $l(\langle x,y\rangle) = l(y) + O(l(x))$. An example is $\langle x,y\rangle = 1^{l(x)}0xy$. This can be iterated to $\langle\langle\cdot,\cdot\rangle,\cdot\rangle$.

Informally, the Kolmogorov complexity [13] of x is the length of the *shortest* effective description of x. That is, the *Kolmogorov complexity* $C(x)$ of a finite string x is simply the length of the shortest program, say in FORTRAN (or in Turing machine codes) encoded in binary, which prints x without any input. A similar definition holds conditionally in the sense that $C(x|y)$ is the length of the shortest binary program which computes x on input y. Kolmogorov complexity is absolute in the sense of being independent of the programming language up to a fixed additional constant term which depends on the programming language but not on x. We now fix one canonical programming language once and for all as reference and thereby $C()$. For the theory and applications, see [9]. A formal definition is as follows:

Definition 1. *Let U be an appropriate universal Turing machine such that*

$$U(\langle\langle i,p\rangle,y\rangle) = T_i(\langle p,y\rangle)$$

for all i and $\langle p,y\rangle$. The conditional Kolmogorov complexity *of x given y is*

$$C(x|y) = \min_{p\in\{0,1\}^*}\{l(p) : U(\langle p,y\rangle) = x\}.$$

The unconditional Kolmogorov complexity of x is defined as $C(x) := C(x|\epsilon)$.

It is easy to see that there are strings that can be described by programs much shorter than themselves. For instance, the function defined by $f(1) = 2$ and $f(i) = 2^{f(i-1)}$ for $i > 1$ grows very fast, $f(k)$ is a "stack" of k twos. Yet for each k it is clear that $f(k)$ has complexity at most $C(k) + O(1)$. What about incompressibility?

By a simple counting argument one can show that whereas some strings can be enormously compressed, the majority of strings can hardly be compressed at all.

For each n there are 2^n binary strings of length n but only $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ possible shorter descriptions. Therefore, there is at least one binary string x of length n such that $C(x) \geq n$. We call such strings *incompressible*. It also follows that for any length n and any binary string y, there is a binary string x of length n such that $C(x|y) \geq n$. Generally, for every constant c we can say a string x is *c-incompressible* if $C(x) \geq l(x) - c$. Strings that are incompressible (say, c-incompressible with small c) are patternless, since a pattern could be used to reduce the description length. Intuitively, we think of such patternless sequences as being random, and we use "random sequence" synonymously with "incompressible sequence."[3] By the same counting argument as before we find that the number of strings of length n that are c-incompressible is at least $2^n - 2^{n-c} + 1$. Hence there is at least one 0-incompressible string of length n, at least one-half of all strings of length n are 1-incompressible, at least three-fourths of all strings of length n are 2-incompressible, ..., and at least the $(1 - 1/2^c)$th part of all 2^n strings of length n are c-incompressible. This means

[3]It is possible to give a rigorous formalization of the intuitive notion of a random sequence as a sequence that passes all effective tests for randomness; see, for example, [9].

that for each constant $c \geq 1$ the majority of all strings of length n (with $n > c$) is c-incompressible. We generalize this to the following simple but extremely useful lemma.

LEMMA 1. *Let c be a positive integer. For each fixed y, every set A of cardinality m has at least $m(1-2^{-c})+1$ elements x with $C(x|y) \geq \lfloor \log m \rfloor - c$.*

Proof. The proof is by simple counting. ☐

As an example, set $A = \{x : l(x) = n\}$. Then the cardinality of A is $m = 2^n$. Since it is easy to assert that $C(x) \leq n + c$ for some fixed c and all x in A, Lemma 1 demonstrates that this trivial estimate is quite sharp. The deeper reason is that since there are few short programs, there can be only few objects of low complexity. We require another quantity: The prefix Kolmogorov complexity which is defined just as $C(\cdot|\cdot)$ but now with respect to a subset of Turing machines that have the property that the set of programs for which the machine halts is prefix-free; that is, no halting program is a prefix of any other halting program. For details see [9]. Here we require only the quantitative relation below.

DEFINITION 2. *The* prefix *Kolmogorov complexity of x conditional to y is denoted by $K(x|y)$. It satisfies the inequality*

$$C(x|y) \leq K(x|y) \leq C(x|y) + 2\log C(x|y) + O(1).$$

2. Kolmogorov random graphs. Statistical properties of strings with high Kolmogorov complexity have been studied in [11]. The interpretation of strings as more complex combinatorial objects leads to a new set of properties and problems that have no direct counterpart in the "flatter" string world. Here we derive topological, combinatorial, and statistical properties of graphs with high Kolmogorov complexity. Every such graph possesses simultaneously all properties that hold with high probability for randomly generated graphs. They constitute "almost all graphs" and the derived properties a fortiori hold with probability that goes to 1 as the number of nodes grows unboundedly.

DEFINITION 3. *Each labeled graph $G = (V, E)$ on n nodes $V = \{1, 2, \ldots, n\}$ can be represented (up to automorphism) by a binary string $E(G)$ of length $\binom{n}{2}$. We simply assume a fixed ordering of the $\binom{n}{2}$ possible edges in an n-node graph, e.g., lexicographically, and let the ith bit in the string indicate presence* (1) *or absence* (0) *of the ith edge. Conversely, each binary string of length $\binom{n}{2}$ encodes an n-node graph. Hence we can identify each such graph with its binary string representation.*

DEFINITION 4. *A labeled graph G on n nodes has* randomness deficiency *at most $\delta(n)$ and is called $\delta(n)$*-random *if it satisfies*

$$C(E(G)|n) \geq \binom{n}{2} - \delta(n). \tag{2.1}$$

2.1. Some basic properties. Using Lemma 1, with $y = n$, A the set of strings of length $\binom{n}{2}$, and $c = \delta(n)$ gives us the following lemma.

LEMMA 2. *A fraction of at least $1 - 1/2^{\delta(n)}$ of all labeled graphs G on n nodes is $\delta(n)$-random.*

As a consequence, for example, the $c \log n$-random labeled graphs constitute a fraction of at least $(1 - 1/n^c)$ of all graphs on n nodes, where $c > 0$ is an arbitrary constant.

Labeled graphs with high complexity have many specific topological properties, which seem to contradict their randomness. However, these are simply the likely properties, whose absence would be rather unlikely. Thus, randomness enforces strict statistical regularities—for example, to have diameter exactly 2.

We will use the following lemma (Theorem 2.6.1 in [9]).

LEMMA 3. *Let $x = x_1 \dots x_n$ be a binary string of length n, and y a much smaller string of length l. Let $p = 2^{-l}$ and $\#y(x)$ be the number of (possibly overlapping) distinct occurrences of y in x. For convenience, we assume that x "wraps around" so that an occurrence of y starting at the end of x and continuing at the start also counts. Assume that $l \leq \log n$. There is a constant c such that for all n and $x \in \{0,1\}^n$ if $C(x) \geq n - \delta(n)$, then*

$$|\#y(x) - pn| \leq \sqrt{\alpha pn}$$

with $\alpha = [K(y|n) + \log l + \delta(n) + c]3l/\log e$.

LEMMA 4. *All $o(n)$-random labeled graphs have $n/4 + o(n)$ disjoint paths of length 2 between each pair of nodes i, j. In particular, all $o(n)$-random labeled graphs have diameter 2.*

Proof. The only graphs with diameter 1 are the complete graphs that can be described in $O(1)$ bits, given n, and hence are not random. It remains to consider an $o(n)$-random graph $G = (V, E)$ with diameter greater than or equal to 2. Let i, j be a pair of nodes connected by r disjoint paths of length 2. Then we can describe G by modifying the old code for G as follows:

- a program to reconstruct the object from the various parts of the encoding in $O(1)$ bits;
- the identities of $i < j$ in $2 \log n$ bits;
- the old code $E(G)$ of G with the $2(n-2)$ bits representing presence or absence of edges (j, k) and (i, k) for each $k \neq i, j$ deleted;
- a short program for the string $e_{i,j}$ consisting of the (reordered) $n - 2$ pairs of bits deleted above.

From this description we can reconstruct G in

$$O(\log n) + \binom{n}{2} - 2(n-2) + C(e_{i,j}|n)$$

bits, from which we may conclude that $C(e_{i,j}|n) \geq l(e_{i,j}) - o(n)$. As shown in [11] or [9] (here Lemma 3) this implies that the frequency of occurrence in $e_{i,j}$ of the aligned 2-bit block "11"—which by construction equals the number of disjoint paths of length 2 between i and j—is $n/4 + o(n)$. ☐

A graph is *k-connected* if there are at least k node-disjoint paths between every pair of nodes.

COROLLARY 1. *All $o(n)$-random labeled graphs are $(\frac{n}{4} + o(n))$-connected.*

LEMMA 5. *Let $G = (V, E)$ be a graph on n nodes with randomness deficiency $O(\log n)$. Then the largest clique in G has at most $\lfloor 2 \log n \rfloor + O(1)$ nodes.*

Proof. The proof is the same as the largest size transitive subtournament in a high-complexity tournament as in [9]. ☐

With respect to the related property of random graphs, in [1, pp. 86–87], it is shown that a random graph with edge probability 1/2 contains a clique on asymptotically $2 \log n$ nodes with probability at least $1 - e^{-n^2}$.

2.2. Statistics of subgraphs. We start by defining the notion of labeled subgraph of a labeled graph.

DEFINITION 5. *Let $G = (V, E)$ be a labeled graph on n nodes. Consider a labeled graph H on k nodes $\{1, 2, \ldots, k\}$. Each subset of k nodes of G induces a subgraph G_k of G. The subgraph G_k is an ordered labeled* occurrence *of H when we obtain H by relabeling the nodes $i_1 < i_2 < \cdots < i_k$ of G_k as $1, 2, \ldots, k$.*

It is easy to conclude from the statistics of high-complexity strings in Lemma 3 that the frequency of each of the two labeled two-node subgraphs (there are only two different ones: the graph consisting of two isolated nodes and the graph consisting of two connected nodes) in a $\delta(n)$-random graph G is

$$\frac{n(n-1)}{4} \pm \sqrt{\frac{3}{4}(\delta(n) + O(1))n(n-1)/\log e}.$$

This case is easy since the frequency of such subgraphs corresponds to the frequency of 1s or 0s in the $\binom{n}{2}$-length standard encoding $E(G)$ of G. However, to determine the frequencies of labeled subgraphs on k nodes (up to isomorphism) for $k > 2$ is a matter more complicated than the frequencies of substrings of length k. Clearly, there are $\binom{n}{k}$ subsets of k nodes out of n and hence that many occurrences of subgraphs. Such subgraphs may overlap in more complex ways than substrings of a string. Let $\#H(G)$ be *the number of times H occurs* as an ordered labeled subgraph of G (possibly overlapping). Let p be the probability that we obtain H by flipping a fair coin to decide for each pair of nodes whether it is connected by an edge or not:

$$p = 2^{-k(k-1)/2}. \tag{2.2}$$

THEOREM 1. *Assume the terminology above with $G = (V, E)$ a labeled graph on n nodes, k is a positive integer dividing n, and H is a labeled graph on $k \le \sqrt{2 \log n}$ nodes. Let $C(E(G)|n) \ge \binom{n}{2} - \delta(n)$. Then*

$$\left| \#H(G) - \binom{n}{k} p \right| \le \binom{n}{k} \sqrt{\alpha(k/n)p}$$

with $\alpha := (K(H|n) + \delta(n) + \log \binom{n}{k}/(n/k) + O(1))3/\log e$.

Proof. A *cover* of G is a set $C = \{S_1, \ldots, S_N\}$ with $N = n/k$, where the S_i's are pairwise disjoint subsets of V and $\bigcup_{i=1}^{N} S_i = V$. According to [3], we have the following claim.

Claim 1. There is a partition of the $\binom{n}{k}$ different k-node subsets into $h = \binom{n}{k}/N$ distinct covers of G, each cover consisting of $N = n/k$ disjoint subsets. That is, each subset of k nodes of V belongs to precisely one cover.

Enumerate the covers as $C_0, C_2, \ldots, C_{h-1}$. For each $i \in \{0, 1, \ldots, h-1\}$ and k-node labeled graph H, let $\#H(G, i)$ be the number of (now nonoverlapping) occurrences of subgraph H in G occurring in cover C_i.

Now consider an experiment of N trials, each trial with the same set of $2^{k(k-1)/2}$ outcomes. Intuitively, each trial corresponds to an element of a cover, and each outcome corresponds to a k-node subgraph. For every i we can form a string s_i consisting of the N blocks of $\binom{k}{2}$ bits that represent presence or absence of edges within the induced subgraphs of each of the N subsets of C_i. Since G can be reconstructed from n, i, s_i, and the remaining $\binom{n}{2} - N\binom{k}{2}$ bits of $E(G)$, we find that $C(s_i|n) \ge l(s_i) - \delta(n) - \log h$. Again, according to Lemma 3 this implies that the frequency of occurrence of the aligned $\binom{k}{2}$-block $E(H)$, which is $\#H(G, i)$, equals

$$Np \pm \sqrt{Np\alpha}$$

with α as in the statement of Theorem 1. One can do this for each i independently, notwithstanding the dependence between the frequencies of subgraphs in different covers. Namely, the argument depends on the incompressibility of G alone. If the number of occurrences of a certain subgraph in *any* of the covers is too large or too small then we can compress G. Now,

$$\begin{aligned}\left|\#H(G) - p\binom{n}{k}\right| &= \sum_{i=0}^{h-1} |\#H(G,i) - Np| \\ &\leq \binom{n}{k}\sqrt{\alpha(k/n)p}. \qquad \square\end{aligned}$$

In [9, 11] we investigated up to which length l all blocks of length l occurred at least once in each $\delta(n)$-random string of length n.

Theorem 2. *Let $\delta(n) < 2^{\sqrt{\frac{1}{2}\log n}}/4\log n$ and G be a $\delta(n)$-random graph on n nodes. Then for sufficiently large n, the graph G contains all subgraphs on $\sqrt{2\log n}$ nodes.*

Proof. We are sure that H on k nodes occurs at least once in G if $\binom{n}{k}\sqrt{\alpha(k/n)p}$ in Theorem 1 is less than $\binom{n}{k}p$. This is the case if $\alpha < (n/k)p$. This inequality is satisfied for an overestimate of $K(H|n)$ by $\binom{k}{2}+2\log\binom{k}{2}+O(1)$ (since $K(H|n) \leq K(H)+O(1)$), and $p = 2^{-k(k-1)/2}$, with k set at $k = \sqrt{2\log n}$. This proves the theorem. $\square$

2.3. Unlabeled graph counting. An unlabeled graph is a graph with no labels. For convenience we can define this as follows: Call two labeled graphs *equivalent* (up to relabeling) if there is a relabeling that makes them equal. An *unlabeled graph* is an equivalence class of labeled graphs. An *automorphism* of $G = (V, E)$ is a permutation π of V such that $(\pi(u), \pi(v)) \in E$ iff $(u, v) \in E$. Clearly, the set of automorphisms of a graph forms a group with group operation of function composition and the identity permutation as unity. It is easy to verify that π is an automorphism of G iff $\pi(G)$ and G have the *same binary string standard encoding*, that is, $E(G) = E(\pi(G))$. This contrasts with the more general case of permutation relabeling, where the standard encodings may be different. A graph is *rigid* if its only automorphism is the identity automorphism. It turns out that Kolmogorov random graphs are rigid graphs. To obtain an expression for the number of unlabeled graphs we have to estimate the number of automorphisms of a graph in terms of its randomness deficiency.

In [12] an asymptotic expression for the number of unlabeled graphs is derived using sophisticated methods. We give a new elementary proof by incompressibility. Denote by g_n the number of unlabeled graphs on n nodes—that is, the number of isomorphism classes in the set $\mathcal{G}_n$ of undirected graphs on nodes $\{0, 1, \ldots, n-1\}$.

Theorem 3. $g_n \sim \frac{2^{\binom{n}{2}}}{n!}$.

Proof. Clearly,

$$g_n = \sum_{G \in \mathcal{G}_n} \frac{1}{d(\bar{G})},$$

where $\bar{G}$ is the isomorphism class of graph G. By elementary group theory,

$$d(\bar{G}) = \frac{d(S_n)}{d(Aut(G))} = \frac{n!}{d(Aut(G))},$$

where S_n is the group of permutations on n elements and $Aut(G)$ is the automorphism group of G. Let us partition $\mathcal{G}_n$ into $\mathcal{G}_n = \mathcal{G}_n^0 \cup \cdots \cup \mathcal{G}_n^n$, where $\mathcal{G}_n^m$ is the set of graphs for which m is the number of nodes moved (mapped to another node) by any of its automorphisms.

Claim 2. *For $G \in \mathcal{G}_n^m$, $d(Aut(G)) \leq n^m = 2^{m \log n}$.*

Proof. $d(Aut(G)) \leq \binom{n}{m} m! \leq n^m$. □

Consider each graph $G \in \mathcal{G}_n$ having a probability $\text{Prob}(G) = 2^{-\binom{n}{2}}$.

Claim 3. $\text{Prob}(G \in \mathcal{G}_n^m) \leq 2^{-m(\frac{n}{2} - \frac{3m}{8} - \log n)}$.

Proof. By Lemma 2 it suffices to show that if $G \in \mathcal{G}_n^m$ and

$$C(E(G)|n, m) \geq \binom{n}{2} - \delta(n, m)$$

then $\delta(n, m)$ satisfies

$$\delta(n, m) \geq m \left(\frac{n}{2} - \frac{3m}{8} - \log n \right). \tag{2.3}$$

Let $\pi \in Aut(G)$ move m nodes. Suppose π is the product of k disjoint cycles of sizes $c_1, \ldots, c_k$. Spend at most $m \log n$ bits describing π: For example, if the nodes $i_1 < \cdots < i_m$ are moved then list the sequence $\pi(i_1), \ldots, \pi(i_m)$. Writing the nodes of the latter sequence in increasing order we obtain $i_1, \ldots, i_m$ again; that is, we execute permutation π^{-1} and hence we obtain π.

Select one node from each cycle—say, the lowest numbered one. Then for every unselected node on a cycle, we can delete the $n-m$ bits corresponding to the presence or absence of edges to stable nodes, and $m-k$ half-bits corresponding to presence or absence of edges to the other, unselected cycle nodes. In total we delete

$$\sum_{i=1}^{k} (c_i - 1) \left(n - m + \frac{m-k}{2} \right) = (m-k) \left(n - \frac{m+k}{2} \right)$$

bits. Observing that $k = m/2$ is the largest possible value for k, we arrive at the claimed $\delta(n, m)$ of G (difference between savings and spendings is $\frac{m}{2}(n - \frac{3m}{4}) - m \log n$) of (2.3). □

We continue the proof of Theorem 3:

$$g_n = \sum_{G \in \mathcal{G}_n} \frac{1}{d(\bar{G})} = \sum_{G \in \mathcal{G}_n} \frac{d(Aut(g))}{n!} = \frac{2^{\binom{n}{2}}}{n!} E_n,$$

where $E_n := \sum_{G \in \mathcal{G}_n} \text{Prob}(G) d(Aut(G))$ is the expected size of the automorphism group of a graph on n nodes. Clearly, $E_n \geq 1$, yielding the lower bound on g_n. For the upper bound on g_n, noting that $\mathcal{G}_n^1 = \emptyset$ and using the above claims, we find

$$\begin{aligned} E_n &= \sum_{m=0}^{n} \text{Prob}(G \in \mathcal{G}_n^m) Avg_{G \in \mathcal{G}_n^m} d(Aut(G)) \\ &\leq 1 + \sum_{m=2}^{n} 2^{-m(\frac{n}{2} - \frac{3m}{8} - 2 \log n)} \\ &\leq 1 + 2^{-(n - 4 \log n - 2)}, \end{aligned}$$

which proves the theorem. □

The proof of the theorem shows that the error in the asymptotic expression is very small.

COROLLARY 2. $\frac{2^{\binom{n}{2}}}{n!} \leq g_n \leq \frac{2^{\binom{n}{2}}}{n!}(1 + \frac{4n^4}{2^n})$.

The next corollary follows from (2.3) (since $m = 1$ is impossible).

COROLLARY 3. *If a graph G has randomness deficiency slightly less than n (more precisely, $C(E(G)|n) \geq \binom{n}{2} - n - \log n - 2$) then G is rigid.*

The expression for g_n can be used to determine the maximal complexity of an unlabeled graph on n nodes. Namely, we can effectively enumerate all unlabeled graphs as follows:

- Effectively enumerate all labeled graphs on n nodes by enumerating all binary strings of length n and for each labeled graph G do the following:
  If G cannot be obtained by relabeling from any previously enumerated labeled graph then G is added to the set of unlabeled graphs.

This way we obtain each unlabeled graph by precisely one labeled graph representing it. Since we can describe each unlabeled graph by its index in this enumeration, we find by Theorem 3 and Stirling's formula that if G is an unlabeled graph then

$$C(E(G)|n) \leq \binom{n}{2} - n \log n + O(n).$$

THEOREM 4. *Let G be a labeled graph on n nodes and let G_0 be the unlabeled version of G. There exists a graph G' and a label permutation π such that $G' = \pi(G)$ and up to additional constant terms $C(E(G')) = C(E(G_0))$ and $C(E(G)|n) = C(E(G_0), \pi|n)$.*

By Theorem 4, for *every* graph G on n nodes with maximum complexity there is a relabeling (permutation) that causes the complexity to drop by as much as $n \log n$. Our proofs of topological properties by the incompressibility method required the graph G to be Kolmogorov random in the sense of $C(E(G)|n) \geq \binom{n}{2} - O(\log n)$ or for some results $C(E(G)|n) \geq \binom{n}{2} - o(n)$. Hence by relabeling such a graph we can always obtain a labeled graph that has a complexity too low to use our incompressibility proof. Nonetheless, topological properties do not change under relabeling.

REFERENCES

[1] N. ALON, J.H. SPENCER, AND P. ERDÖS, *The Probabilistic Method*, Wiley, 1992,

[2] E. BOREL, *Leçons sur la Théorie des Functions*, 2nd ed., Gauthier-Villars, Paris, 1914, pp. 182–216.

[3] BARANYAI, ZS., *On the factorization of the complete uniform hypergraph,* in Infinite and Finite Sets, Proc. Coll. Keszthely, A. Hajnal, R. Rado, and V.T. Sós, eds., Colloq. Math. Soc. János Bolyai 10, North-Holland, Amsterdam, 1995, pp. 91–108.

[4] B. BOLLOBÁS, *Graph Theory*, Springer-Verlag, New York, 1979.

[5] B. BOLLOBÁS, *Random Graphs*, Academic Press, London, 1985.

[6] H. BUHRMAN, J.H. HOEPMAN, AND P. VITÁNYI, *Space-efficient routing tables for almost all networks and the incompressibility method*, SIAM J. Comput., 28 (1999), pp. 1414–1432.

[7] P. ERDÖS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[8] W.W. KIRCHHERR, *Kolmogorov complexity and random graphs*, Inform. Process. Lett., 41 (1992), pp. 125–130.

[9] M. LI AND P.M.B. VITÁNYI, *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd ed., Springer-Verlag, New York, 1997.

[10] M. LI AND P.M.B. VITÁNYI, *Kolmogorov complexity arguments in Combinatorics*, J. Combin. Theory Ser. A, 66 (1994), pp. 226–236. Errata, J. Combin. Theory Ser. A, 69 (1995), p. 183.

[11] M. LI AND P.M.B. VITÁNYI, *Statistical properties of finite sequences with high Kolmogorov complexity*, Math. Systems Theory, 27 (1994), pp. 365–376.

[12] F. Harary and E.M. Palmer, *Graphical Enumeration*, Academic Press, New York, London, 1973.

[13] A.N. Kolmogorov, *Three approaches to the quantitative definition of information*, Problems Inform. Transmission, 1 (1965), pp. 1–7.

[14] P. Martin-Löf, *On the definition of random sequences*, Inform. and Control, 9 (1966), pp. 602–619.

[15] P.M.B. Vitányi, *Physics and the new computation*, in Proceedings of the 20th International Symposium on Math. Foundations of Computer Science, Prague, 1995, Lecture Notes in Comput. Sci. 969, Springer-Verlag, Heidelberg, 1995, pp. 106–128.

SIAM J. Comput.
Vol. 29, No. 2, pp. 600–614

SELF-STABILIZING ALGORITHMS FOR FINDING CENTERS AND MEDIANS OF TREES[*]

STEVEN C. BRUELL[†], SUKUMAR GHOSH[†], MEHMET HAKAN KARAATA[‡], AND SRIRAM V. PEMMARAJU[§]

Abstract. Locating a center or a median in a graph is a fundamental graph-theoretic problem. Centers and medians are especially important in distributed systems because they are ideal locations for placing resources that need to be shared among different processes in a network. This paper presents simple self-stabilizing algorithms for locating centers and medians of trees. Since these algorithms are self-stabilizing, they can tolerate transient failures. In addition, they can automatically adjust to a dynamically changing tree topology. After the algorithms are presented, their correctness is proven and upper bounds on their time complexity are established. Finally, extensions of our algorithms to trees with arbitrary, positive edge costs are sketched.

Key words. center, distributed algorithm, median, self-stabilization, tree

AMS subject classifications. 05C05, 05C12, 05C85, 68Q22, 68Q25, 68R10

PII. S0097539798427156

1. Introduction.

Let $G = (V, E)$ be a simple, connected graph with vertex set V and edge set E. The *eccentricity* of a vertex $i \in V$ is the largest distance from i to any vertex in V. A vertex with minimum eccentricity is called a *center* of G. The *weight* of a vertex $i \in V$ is the sum of the distances from i to all vertices in V. A vertex with minimum weight is called a *median* of G.

Locating centers and medians of graphs has a wide variety of important applications because placing a common resource at a center or a median of a graph minimizes the costs of sharing the resource with other locations. This is especially important in distributed systems in which information is disseminated from or gathered at a single node (or a small number of nodes). As a result, distributed algorithms for locating centers and medians of graphs are extremely useful. There are several known algorithms [10, 11, 15, 18, 24, 25, 29, 31] for finding centers and medians of graphs, including distributed algorithms by Korach, Rotem, and Santoro [27]. In this paper, we propose two simple, self-stabilizing, distributed algorithms for finding centers and medians of trees. We then prove the correctness of these algorithms and establish upper bounds on their time complexity. We finally show that the algorithms can be slightly modified to locate centers and medians in trees with arbitrary positive edge costs.

Self-stabilization in distributed systems was first introduced by Dijkstra [8] in 1974. Since then the power and simplicity of self-stabilizing algorithms has been amply demonstrated. For example, self-stabilizing mutual exclusion algorithms for various classes of networks have been presented in [5, 8, 19, 22, 28]; self-stabilizing algorithms

[*]Received by the editors July 22, 1994; accepted for publication (in revised form) July 31, 1998; published electronically November 10, 1999.
http://www.siam.org/journals/sicomp/29-2/27156.html

[†]Department of Computer Science, The University of Iowa, Iowa City, IA 52242-1419 (bruell@cs.uiowa.edu, ghosh@cs.uiowa.edu). The research of the second author was supported in part by the National Science Foundation under grant CCR-9402050.

[‡]Electrical and Computer Engineering Department, Kuwait, P.O. Box 5969, Safat 13060, Kuwait (karaata@eng.kuniv.edu.kw).

[§]Department of Mathematics, Indian Institute of Technology, Bombay, Powai, Mumbai 400 076, India (sriram@math.iitb.ernet.in).

for electing a leader appear in [9, 21, 30]; self-stabilizing algorithms for constructing spanning trees have been presented in [1, 6, 7]; self-stabilizing algorithms for problems related to maximum flow and maximum matching appear in [13, 16, 20]; self-stabilizing algorithms for coloring graphs appear in [14, 34]; general techniques for constructing self-stabilizing algorithms are presented in [2, 26]. Not only are self-stabilizing algorithms tolerant to transient failures, but many of them are also capable of handling the addition and deletion of vertices or edges in a transparent manner. These attractive properties give self-stabilizing algorithms a distinct advantage over the more classical distributed algorithms.

By using the general techniques of Katz and Perry [26] and Awerbuch and Varghese [2], it is possible to construct self-stabilizing center-finding or median-finding algorithms. However, the general technique proposed by Katz and Perry [26] requires global state collection at a leader followed by distributed reset (if necessary) and this makes their method unacceptably inefficient. Therefore, we view Katz and Perry more as a demonstration of the feasibility of self-stabilization than as a practical alternative. A similar claim can be made about the compiler of Awerbuch and Varghese [2] that takes any deterministic, synchronous, distributed protocol in a message passing system and produces an equivalent asynchronous, self-stabilizing, distributed protocol. The technique of Awerbuch and Varghese for constructing self-stabilizing protocols is powerful and general. However, it is because of the generality of their technique that often, hand-crafted protocols for a particular problem are simpler and more efficient than the self-stabilizing protocols generated by their compiler. This is certainly true for the center- and median-finding problems for which our solutions are not only extremely simple, but also time and space optimal.

The balance of this paper is organized as follows. In section 2 we discuss the notion of self-stabilization in distributed systems. In section 3 we present two self-stabilizing algorithms—one for finding centers of trees and the other for finding medians of trees. Section 4 contains proofs that establish the partial correctness of the proposed algorithms. In section 5 we present proofs to show that the proposed algorithms terminate. In section 6 we establish upper bounds on the time complexity of the algorithms. Section 7 extends our algorithms so as to locate centers and medians in trees that have arbitrary, positive edge costs. Section 8 contains concluding remarks and a glimpse of possible future work.

2. Self-stabilization in distributed systems.

Following Schneider [32], we view a distributed system S as consisting of two types of components: *processes* and *interconnections* between processes representing communication through local shared memory or through message channels. Each component in the system has a *local state* and the *global state* of the system is simply the union of the local states of all components. Let P be a predicate defined over the set of global states of the system. An algorithm A running on S is said to be *self-stabilizing* with respect to P if it satisfies the following:

Closure: If a global state q satisfies P, then any global state that is reachable from q using algorithm A, also satisfies P.

Convergence: Starting from an arbitrary global state, the distributed system S is guaranteed to reach a global state satisfying P in a finite number of steps of A.

Global states satisfying P are said to be *stable*. To show that an algorithm is self-stabilizing with respect to P we need to show both closure and convergence. In addition, to show that an algorithm solves a certain problem, we need to show *partial*

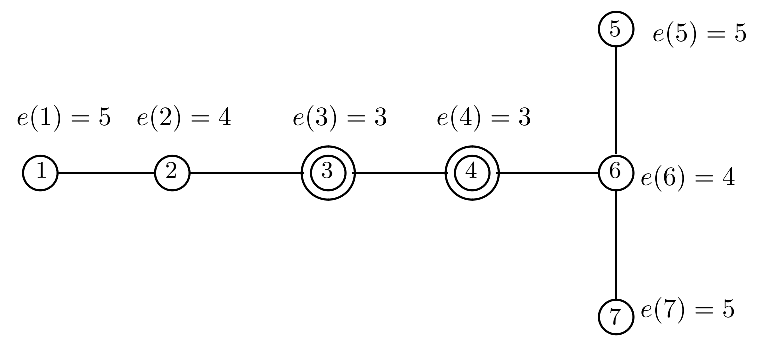


FIG. 1. *Eccentricities of vertices in a tree. The two centers in the tree are marked by double circles.*

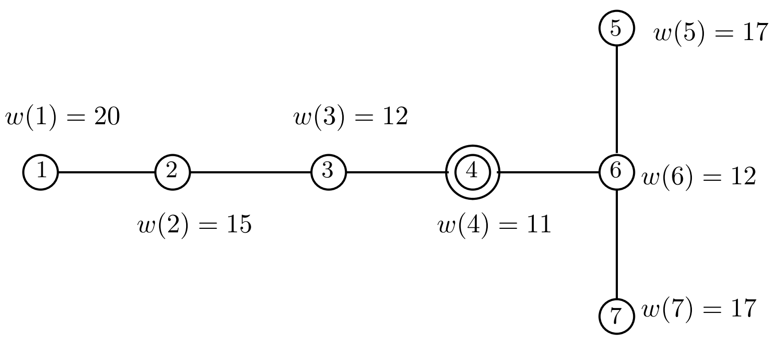


FIG. 2. *Weights of vertices in a tree. The unique median in the tree is marked by a double circle.*

correctness, that is, every stable state of S constitutes a solution to the problem.

3. Center and median-finding algorithms.

3.1. Notation. We begin this section with some notation and terminology.

- $d(i, j)$ denotes the *distance*, the length of a shortest path in G, between vertices i and j.
- $e(i) = \max\{d(i, j) \mid j \in V\}$ denotes the *eccentricity* of a vertex i, the distance between i and a farthest vertex from i in G.
- $\mathit{center}(G) = \{i \in V \mid e(i) \leq e(j) \text{ for all } j \in V\}$ denotes the set of *centers* of G, the set of vertices in V with minimum eccentricity.
- $w(i) = \sum_{j \in V} d(i, j)$ denotes the *weight* of a vertex i, the sum of the distances between i and all vertices in V.
- $\mathit{median}(G) = \{i \in V \mid w(i) \leq w(j) \text{ for all } j \in V\}$ denotes the set of *medians* of G, the set of vertices in V with minimum weight.

Figure 1 shows a tree along with the eccentricity of each vertex. Vertices 3 and 4 have the minimum eccentricity, and are therefore centers of the tree. Figure 2 shows the same tree along with the weight of each vertex. Vertex 4 has the minimum weight, and is therefore the unique median.

For the remainder of the paper we restrict our attention to trees. We demonstrate that for this class of graphs, simple and elegant self-stabilizing algorithms for locating centers and medians exist. Our algorithms have been used as the basis of other self-stabilizing algorithms on trees [12, 33]. We hope that our algorithms can be extended to solve the corresponding problems for arbitrary graphs. Let $T = (V, E)$ be a tree with vertex set V and edge set E. Without loss of generality we assume that $V = \{1, 2, \ldots, n\}$. The following proposition states well-known properties of centers and medians of trees [3, 4, 17].

PROPOSITION 3.1. *A tree has a single center (median) or two adjacent centers (medians).*

3.2. The model of computation. Before presenting our self-stabilizing algorithms, we briefly describe the underlying model of computation. We assume that each vertex in T is a process. Each process i maintains a single local variable whose value can be read by the neighboring processes, but can be written into only by i. The program executed by each process is expressed in the language of guarded commands. We assume that each process repeatedly evaluates its guards and checks if any guard is true. If so, the process executes the corresponding action in a *single atomic step*. Such an atomic step by a process is called a *move*. A guard that evaluates to true in a certain global state is said to be *enabled* in that global state. In this model of computation, the execution of our algorithms can be viewed as a sequence of moves, in which moves by different processes are interleaved. In particular, an *execution sequence* of an algorithm A running on a distributed system S is a sequence: $q_0, m_1, q_1, m_2, q_2, \ldots$ such that

(i) q_i, for each $i \geq 0$, is a global state of the system S,
(ii) m_i, for each $i \geq 1$, is a move by some process executing algorithm A,
(iii) move m_i is a move whose guard is enabled in state q_{i-1}; move m_i takes the system from state q_{i-1} into state q_i,
(iv) the sequence is either infinite or is finite and, if finite, ends in a global state in which no guard is enabled.

Since our algorithms are self-stabilizing, no initialization of the variables is assumed. Furthermore, the underlying network topology is permitted to change under the condition that the topology remain connected and acyclic.

3.3. The algorithms. To facilitate the description of the algorithms, we introduce two functions. Let $h : V \to \mathcal{N}$ be a function from vertex set V to the set of natural numbers $\mathcal{N}$. Similarly, let $s : V \to \mathcal{N} - \{0\}$ be a function from V to the set of positive natural numbers. We will refer to $h(i)$ and $s(i)$ as the h-value and the s-value, respectively, of vertex i. In addition, we will use the following notation:

$N(i) = \{j \mid (i, j) \in E\}$ denotes the set of neighbors of vertex i.

$N_h(i) = \{h(j) \mid (i, j) \in E\}$ denotes the *multiset* of h-values of the neighbors of i.

$N_h^-(i) = N_h(i) - \{\max(N_h(i))\}$ denotes all of $N_h(i)$ with *one* maximum h-value removed. For example, if $N_h(i) = \{2, 3, 3\}$, then $N_h^-(i) = \{2, 3\}$.

$N_s(i) = \{s(j) \mid (i, j) \in E\}$ denotes the *multiset* of s-values of the neighbors of i.

$N_s^-(i) = N_s(i) - \{\max(N_s(i))\}$ denotes all of $N_s(i)$ with *one* maximum s-value removed.

To motivate our algorithms, we first make a few observations. The following condition on the h-value of vertex i is called the *height condition*:

$$h(i) = \begin{cases} 0 & \text{if } i \text{ is a leaf,} \\ 1 + \max(N_h^-(i)) & \text{otherwise.} \end{cases}$$

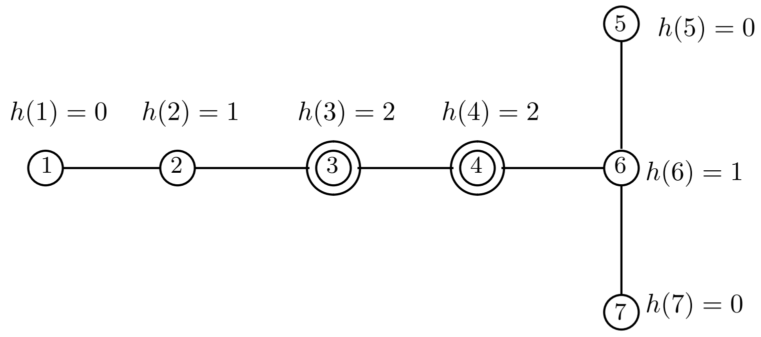


FIG. 3. *h-values of vertices satisfying the height condition in a tree.*

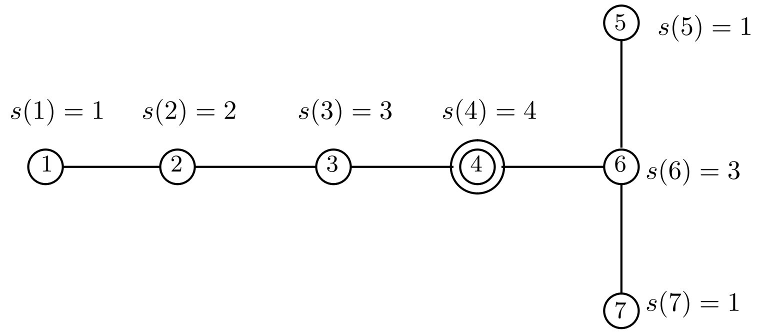


FIG. 4. *s-values of vertices satisfying the size condition in a tree.*

Here we use the notation $\max(N_h^-(i))$ to stand for the maximum value in the multiset $N_h^-(i)$.

The following condition on the s-value of a vertex i is called the *size condition*:

$$s(i) = \begin{cases} 1 & \text{if } i \text{ is a leaf,} \\ 1 + \Sigma(N_s^-(i)) & \text{otherwise.} \end{cases}$$

Here we use the notation $\Sigma(N_s^-(i))$ to represent the sum of all the elements in the multiset $N_s^-(i)$.

We will show that if the h-values (s-values) of all the vertices in T satisfy the height (size) condition, then the centers (medians) of T are the *only* vertices whose h-values (s-values) are greater than or equal to the h-values (s-values) of all neighboring vertices. Figure 3 shows a tree with the h-values of all its vertices satisfying the height condition. The two centers, 3 and 4, are the only vertices whose h-values are greater than or equal to the h-values of all neighbors. Figure 4 shows a tree with the s-values of all its vertices satisfying the size condition. The median, 4, is the only vertex whose s-value is greater than or equal to the s-values of all neighbors. Our center-finding and the median-finding algorithms are shown in Figure 5. It is easy to observe that each of the two statements in the center-finding (median-finding) algorithm is an attempt to ensure that the height (size) condition is satisfied by vertex i.

4. Partial correctness. The stable states of the center-finding algorithm are defined as those that satisfy the height condition. Similarly, the stable states of the

```
{ Center-finding algorithm for vertex $i$ }
$*\Big[(i \text{ is a leaf}) \wedge (h(i) \neq 0)$   $\longrightarrow$   $h(i) := 0$
$\Box\ (i \text{ is not a leaf}) \wedge (h(i) \neq 1 + max(N_h^-(i)))$   $\longrightarrow$   $h(i) := 1 + max(N_h^-(i))]$

{ Median-finding algorithm for vertex $i$ }
$*\Big[(i \text{ is a leaf}) \wedge (s(i) \neq 1)$   $\longrightarrow$   $s(i) := 1$
$\Box\ (i \text{ is not a leaf}) \wedge (s(i) \neq 1 + \Sigma(N_s^-(i)))$   $\longrightarrow$   $s(i) := 1 + \Sigma(N_s^-(i))]$
```

FIG. 5. *The center-finding and the median-finding algorithms.*

median-finding algorithm are those that satisfy the size condition. We prove the partial correctness of the center-finding and median-finding algorithms by showing that the stable states of these algorithms indeed constitute a solution to the center-finding and median-finding problem, respectively.

To prove the partial correctness of the proposed algorithms, we introduce the notion of an ordering function. A function $f : V \to \mathcal{N}$, that associates natural numbers to all vertices in T, is an *ordering function* if for all $i \in V$, there exists at most one neighbor j of i such that $f(i) \leq f(j)$. We shall refer to $f(i)$ as the f-value of vertex i. The following is easily verified (see Figures 3 and 4 for examples).

PROPOSITION 4.1. *If the h-values (s-values) of all the vertices in T satisfy the height condition (size condition), then h (s) is an ordering function.*

Given an ordering function $f : V \to \mathcal{N}$, define a directed graph (digraph) $G(f) = (N(f), A(f))$ whose node[1] set is $N(f) = V$ and whose arc set $A(f)$ is defined as

$$A(f) = \{(i,j) \mid j \in N(i) \text{ and } (f(j), j) \text{ is lexicographically the largest}\}.$$

Thus, from each node $i \in N(f)$ there is an outgoing arc to a neighbor with largest f-value. If there are several neighbors with largest f-value, then the tie is broken by choosing a neighbor with the largest f-value *and* the largest label.

The following lemma states an important property of the digraph $G(f)$.

LEMMA 4.2. *The underlying undirected graph of the digraph $G(f)$ is connected. $G(f)$ contains exactly one cycle and this cycle is of length two.*

Proof. Since T is connected, by definition of the arc set $A(f)$, for all $i \in N(f)$, there exists exactly one neighbor of i, say j, such that $(i,j) \in A(f)$. By definition of an ordering function, for any other neighbor of i, say k, $k \neq j$, we have $f(k) < f(i)$. Since $f(k) < f(i)$, from the definition of an ordering function, we conclude that for all neighbors k' of k, where $k' \neq i$, $f(k') < f(k)$. Therefore $(k,i) \in A(f)$. This implies that for every edge in T there is at least one corresponding arc in $G(f)$ and hence $G(f)$ is connected. Furthermore, every node in $G(f)$ has exactly one outgoing arc implying that $G(f)$ has $|N(f)|$ arcs. Since $G(f)$ is a connected graph with $|N(f)|$ arcs, it contains exactly one cycle. Suppose that $(i_1, i_2, \ldots, i_l)$ for $l \geq 3$ is a cycle in $G(f)$. We know that if (i,j) is an arc in $A(f)$, then (i,j) is an edge in E. This implies that $(i_1, i_2, \ldots, i_l)$ is a cycle in T also. But, T is acyclic. Hence, the single cycle that $G(f)$ contains has to be a 2-cycle (a cycle of length two). $\Box$

Let i and j be the two nodes in $G(f)$ that belong to its unique cycle. Without loss of generality assume that $f(i) \geq f(j)$. On deleting the arcs (i,j) and (j,i) from

[1]We employ the convention of using "vertex" and "edge" for undirected graphs and "node" and "arc" for directed graphs.

$G(f)$ we obtain two connected components, each of which is a directed tree, with arcs directed towards i and j, respectively. Denote the directed tree containing node i by $T_i(f) = (N_i(f), A_i(f))$ and the directed tree containing node j by $T_j(f) = (N_j(f), A_j(f))$. Designate i as the root of $T_i(f)$ and j as the root of $T_j(f)$. Since $T_i(f)$ and $T_j(f)$ are now rooted trees, we can use the standard notion of a parent-child relationship between pairs of adjacent nodes in these trees. The following property of the trees $T_i(f)$ and $T_j(f)$ is easy to verify.

PROPOSITION 4.3. *Each arc in $T_i(f)$ and in $T_j(f)$ is directed from a node to its parent and if k is a nonleaf node in $T_i(f)$ or in $T_j(f)$, then $f(k) > f(\ell)$ for each child ℓ of k.*

We now use Proposition 4.1, Lemma 4.2, and Proposition 4.3 to prove the following theorem that provides a way of identifying the centers in T using the h-values associated with the vertices.

THEOREM 4.4. *Let the h-values of all vertices in T satisfy the height condition. Then*

$$center(T) = \{k \mid h(k) \geq h(\ell) \text{ for all } \ell \in N(k)\}.$$

Proof. Since the h-values satisfy the height condition, by Proposition 4.1, h is an ordering function. Hence by Lemma 4.2, $G(h) = (N(h), A(h))$ is a connected digraph, with a unique cycle that is of length two. Let i and j be the two nodes in the unique cycle in $G(h)$ such that $h(i) \geq h(j)$. Consider the rooted, directed trees $T_i(h) = (N_i(h), A_i(h))$ and $T_j(h) = (N_j(h), A_j(h))$. It is easy to see that for all $k \in N_i(h)$ $(N_j(h))$, $h(k)$ is the height of the subtree of $T_i(h)$ $(T_j(h))$ rooted at k.

The rest of the proof is organized into the following two subcases that are based on the relationship between $h(i)$ and $h(j)$.

Case 1. $h(i) > h(j)$. In this case, it follows from Proposition 4.3 that i is the only vertex in T whose h-value is greater than or equal to the h-values of all of its neighbors. We will now show that $center(T) = \{i\}$. Since h is an ordering function and $(i, j) \in A(h)$ we know that j has the largest h-value among all the neighbors of i. Let ℓ be a child of i in $T_i(h)$ with the largest h-value. Then, $h(j) \geq h(\ell)$. Furthermore, we know that $h(i) > h(j)$ and $h(i) = h(\ell)+1$. This implies that $h(\ell) = h(j)$ and hence $h(i) = h(j) + 1$. Since the height of $T_i(h)$ is equal to the distance between the root i and the farthest node from i in $T_i(h)$, we have that $e(i) = \max\{h(i), h(j)+1\} = h(i)$. Similarly, $e(j) = \max(h(i) + 1, h(j)) = h(i) + 1$. Any node in $N_i(h) - \{i\}$ is at least a distance of $h(j) + 2 = h(i) + 1$ from the farthest node in $T_j(h)$ and any node in $N_j(h) - \{j\}$ is at least a distance of $h(i) + 2$ from the farthest node in $T_i(h)$. This implies that i is the vertex in T with the smallest eccentricity. Hence, $center(T) = \{i\}$ and the claim in the theorem is true for the case when $h(i) > h(j)$.

Case 2. $h(i) = h(j)$. In this case, Proposition 4.3 implies that i and j are the only two vertices in T whose h-values are greater than or equal to the h-values of all of their neighbors. We will now show that $center(T) = \{i, j\}$. Furthermore, it is easy to see that $e(i) = e(j) = h(i) + 1 = h(j) + 1$. As in Case 1, any node in $N_i(h) - \{i\}$ is at least a distance of $h(j)+2$ from the farthest node in $T_j(h)$ and any node in $N_j(h)-\{j\}$ is at least a distance of $h(i) + 2$ from the farthest node in $T_i(h)$. Hence, i and j are the two vertices in T with minimum eccentricity. Therefore, $center(T) = \{i, j\}$. The claim in the theorem is true even in the case when $h(i) = h(j)$. ∎

Theorem 4.4 implies that if all h-values satisfy the height-condition, then each process can determine if it is a center of T, by simply checking if its h-value is greater than or equal to the h-values of all its neighbors. Thus a stable state of the system running our center-finding algorithm constitutes a solution to the center-finding problem.

The following theorem is similar to Theorem 4.4 and provides a way of identifying the medians in T using the s-values of the vertices in T. We omit the proof since it is similar to the proof of Theorem 4.4.

THEOREM 4.5. *Let the s-values of all the vertices in T satisfy the size condition. Then*

$$\mathit{median}(T) = \{k \mid s(k) \geq s(\ell) \text{ for all } \ell \in N(k)\}.$$

Theorem 4.5 implies that if all s-values satisfy the size condition, then each process can determine if it is a median of T or not, by simply checking if its s-value is greater than or equal to the s-values of all its neighbors.

5. Convergence.

Our algorithms trivially satisfy the closure property. This is because in a stable state all guards are false and no moves are possible. This implies that once the system reaches a stable state, it remains in a stable state. So we turn our attention to convergence.

To establish convergence of an algorithm, we need to show that any execution sequence of the algorithm is finite in length. In this section we show that any execution sequence of the center-finding algorithm is finite in length, thereby showing that the center-finding algorithm satisfies the convergence property. A similar proof (omitted, see [23]) suffices to show that any execution sequence of the median-finding algorithm is also finite. We call a move by process i a *decreasing move* if it decreases the h-value of process i; otherwise we call the move an *increasing move*. The convergence proof of the center-finding algorithm is organized in two sections. Section 5.1 shows the total number of decreasing moves that all processes can make is at most n^2. Section 5.2 shows that the total number of increasing moves is also finite. The precise upper bound on the number of increasing moves is established in section 6.

5.1. Decreasing moves.

We start with an arbitrary execution sequence by the center-finding algorithm. Let $M(i, p)$ denote a move by process i that is the pth move in this execution sequence. Occasionally, when the identity of the process making a move is irrelevant to our discussion, we will simply use $M(\_, p)$ to denote the move. In what follows, we show that the processes in T can make at most n^2 decreasing moves during the execution of the center-finding algorithm. Our proof is presented in two sections. In section 5.1.1 we define the notion of the cause of a decreasing move and prove two properties related to the cause of a decreasing move. Section 5.1.2 extends the notion of the cause of a decreasing move to define the source of a decreasing move and then shows that distinct decreasing moves by a process have distinct sources. We then show an upper bound on n on the number of distinct sources and this leads to the upper bound of n on the number of decreasing moves each process can make. The upper bound of n^2 on the total number of decreasing moves by all processes follows.

5.1.1. The cause of a decreasing move.

Let $h(i, p)$ denote the value of the variable $h(i)$ after move $M(\_, p)$ and before move $M(\_, p + 1)$. Accordingly, we use $N_h^-(i, p)$ to denote the multi-set $N_h^-(i)$ after move $M(\_, p)$ and before move $M(\_, p+1)$.

Consider a decreasing move $M(i, p)$ by process i. Let us suppose that $M(i, p)$ is not the initial (first) move by process i. Intuitively, our goal is to identify a unique neighbor of process i, say j, and a unique decreasing move, say $M(j, q)$, by process j to be the "cause" of move $M(i, p)$. Since $M(i, p)$ is not the initial move by process i, we can identify a $p' < p$ such that $M(i,p')$ is the move made by process i just before move $M(i,p)$. Note that $M(i,p')$ may or may not be a decreasing move. Focusing on the situation just after move $M(i,p')$, we notice that since process i has just made a move,

$$h(i,p') = \max(N_h^-(i,p')) + 1. \tag{5.1}$$

Since move $M(i,p)$ is decreasing, we have that

$$h(i,p-1) > \max(N_h^-(i,p-1)) + 1. \tag{5.2}$$

Since $M(i,p')$ and $M(i,p)$ are consecutive moves by process i, the value of the variable $h(i)$ does not change after move $M(i,p')$ and before move $M(i,p)$. Hence, $h(i,p') = h(i,p-1)$. This fact, along with (1) and (2), implies that

$$\max(N_h^-(i,p-1)) < \max(N_h^-(i,p')). \tag{5.3}$$

It is then easy to see that there exists a neighbor j, of i, such that

$$h(j,p-1) \le \max(N_h^-(i,p-1)) \tag{5.4}$$

and j makes at least one decreasing move between moves $M(i,p')$ and $M(i,p)$. Furthermore, j itself may have made several decreasing moves between $M(i,p')$ and $M(i,p)$. Let $M(j,q)$ be the *last* decreasing move by process j such that $p' < q < p$. Define *cause*$(M(i,p)) = M(j,q)$. Note that so far *cause*() has been defined only for those decreasing moves that are not initial moves. We extend the definition to include decreasing moves that are initial, by setting *cause*$(M(i,p)) = M(i,p)$ if $M(i,p)$ is decreasing and is the initial move by process i. Hence, the cause of a decreasing move by a process is always a decreasing move made by that process or made by a neighbor.

We now state and prove two useful properties related to the function *cause*(). The first property is that distinct decreasing moves by a process have distinct causes.

LEMMA 5.1. *Suppose that $M(i,p')$ and $M(i,p)$ are distinct decreasing moves by a process i. Then cause$(M(i,p')) \neq$ cause$(M(i,p))$.*

Proof. Without loss of generality assume that $p' < p$. Then *cause*$(M(i,p')) = M(\_,a')$ for some $a' \le p'$ and *cause*$(M(i,p)) = M(\_,a)$ for some a, $p' < a < p$. Thus $a' < a$ and the lemma follows. ☐

The next property that we establish is that the cause relationship is "acyclic."

LEMMA 5.2. *Let $M(i,p)$ be a decreasing move by process i. If $M(i,p)$ is not the initial move by process i, then the move cause(cause$(M(i,p))$) is not made by process i.*

Proof. Suppose that $M(i,p)$ is not the initial move of process i. Then there is a neighbor of i, say j, such that *cause*$(M(i,p)) = M(j,q)$. If $M(j,q)$ is the initial move by process j, then *cause*$(M(j,q)) = M(j,q)$ and the lemma holds. So we assume that $M(j,q)$ is not the initial move by process j. We combine inequalities (2) and (4) to obtain

$$h(j,p-1) \le \max(N_h^-(i,p-1)) < h(i,p-1) - 1. \tag{5.5}$$

After the decreasing move $M(i,p)$ we have $h(j,p) \le \max(N_h^-(i,p)) = h(i,p) - 1$, implying that $h(j,p) < h(i,p)$. By the same token, if *cause*$(M(j,q)) = M(i,r)$ for some $r < q$, then we should have

$$h(i,q) < h(j,q). \tag{5.6}$$

But, we know that $h(j,q) \leq h(j,p-1)$ because $M(j,q)$ is the last decreasing move by process j before move $M(i,p)$. In addition, it follows from (5) that $h(j,p-1) < h(i,p-1)$. We also know that $h(i,p-1) = h(i,q)$ because the h-value of process i remains unchanged between moves $M(j,q)$ and $M(i,p)$. Thus we have

$$h(j,q) \leq h(j,p-1) < h(i,p-1) = h(i,q),$$

which contradicts inequality (6). Hence, *cause*$(M(j,q))$ cannot be a move by process i. $\square$

5.1.2. The source of a decreasing move. Based on the definition of the cause of a decreasing move, we define the notion of the source of a decreasing move.

For each decreasing move $M(i,p)$ define *source*$(M(i,p))$ as follows:

$$\mathit{source}(M(i,p)) = \begin{cases} M(i,p) & \text{if } M(i,p) \text{ is decreasing and is} \\ & \text{the first move of process } i, \\ \mathit{source}(\mathit{cause}(M(i,p))) & \text{if } M(i,p) \text{ is decreasing, but is} \\ & \text{not the first move by process } i. \end{cases}$$

Intuitively, *source*$(M(i,p))$ can be thought of as the initial move that is decreasing and causes $M(i,p)$ through a chain of decreasing moves.

Now suppose that

$$M(i_0,p_0), M(i_1,p_1), \ldots, M(i_a,p_a)$$

is a sequence of decreasing moves where

(a) $M(i_0,p_0) = M(i,p)$,

(b) $M(i_a,p_a) = \mathit{source}(M(i,p))$, and

(c) for all b, $0 \leq b \leq a-1$, we have that $M(i_{b+1},p_{b+1}) = \mathit{cause}(M(i_b,p_b))$, and $M(i_{b+1},p_{b+1}) \neq M(i_b,p_b)$.

Using Lemma 5.2 and the fact that our algorithm is running on an acyclic network, we conclude that the sequence $i_0, i_1, \ldots, i_a$ is a path. Thus, we can associate with each decreasing move $M(i,p)$ a path $i_0, i_1, \ldots, i_a$, called the *path to the source* of move $M(i,p)$. By inductively applying Lemma 5.1 to paths to the sources of moves $M(i,p)$ and $M(i,p')$, we easily obtain the following lemma (see [23] for a proof).

LEMMA 5.3. *Suppose that $M(i,p')$ and $M(i,p)$ are distinct decreasing moves by process i. Then source$(M(i,p')) \neq$ source$(M(i,p))$.*

Since the source of a decreasing move is the first move by some process, Lemma 5.3 has the following immediate consequence.

COROLLARY 5.4. *Each process in T can make at most n decreasing moves.*

As an immediate consequence we have the following lemma.

LEMMA 5.5 (decreasing moves lemma). *The processes in T can make a total of at most n^2 decreasing moves.*

5.2. Increasing moves. In this section we show that the number of increasing moves by the center-finding algorithm is finite. Consider an arbitrary execution sequence of the center-finding algorithm. Since the algorithm can make at most n^2 decreasing moves, there exists a finite prefix of the execution sequence that contains all the decreasing moves made by the algorithm. Let the length of the smallest such prefix be t. Hence the execution sequence $X = M(\_,t+1), M(\_,t+2), M(\_,t+3), \ldots$ contains only increasing moves. Our goal is to show that X is a finite sequence.

LEMMA 5.6 (increasing moves lemma). *X is a finite sequence.*

Proof (by contradiction). To obtain a contradiction, suppose that X is infinite. Clearly, there exists at least one process, say j_0, such that infinitely many moves in X are made by j_0. Recall that a leafprocess can make at most one move (and that move is decreasing) and hence j_0 is not a leaf-process and has at least two neighbors. Clearly, j_0 has at least one nonleaf neighbor, say j_1, that makes infinitely many moves in X. We will now show that j_0 has at least two neighbors that make infinitely many moves in X. Again we show this by contradiction.

Suppose that j_1 is the only neighbor of j_0 that makes infinitely many moves in X. Then there exists $t' > t$ such that no neighbor of j_0, except j_1, makes moves in the execution sequence

$$X' = M(\_, t'), M(\_, t'+1), M(\_, t'+2), \ldots .$$

Note that j_0 and j_1 make infinitely many moves in X'. Clearly, after finitely many moves in X', $h(j_1)$ will be strictly greater than the h-value of any other neighbor of j_0. This means that after finitely many moves in X', $N_h^-(j_0)$ will no longer contain $h(j_1)$ and will therefore subsequently remain unchanged. But if $N_h^-(j_0)$ remains unchanged, j_0 has no reason to make any more moves. The implication is that j_0 makes only finitely many moves in X' and as a result j_0 makes finitely many moves in X. This contradicts our original supposition that j_0 makes infinitely many moves in X.

Thus j_0 has at least two nonleaf neighbors that make infinitely many moves in X. The same is true of j_1 and hence we claim that j_1 has a nonleaf neighbor j_2, distinct from j_0, that makes infinitely many moves in X. This argument can be carried on further to claim the existence of an infinite sequence of distinct nonleaf processes that make infinitely many moves in X. But this is a contradiction because T has a finite number of processes. Hence, our original assumption that X is an infinite execution sequence is false. $\square$

The increasing moves lemma along with the decreasing moves lemma (Lemma 5.5) leads to the following theorem.

THEOREM 5.7. *Any execution sequence of the center-finding algorithm is finite.*

This establishes the total correctness of the center-finding algorithm.

6. Time complexity.

In this section we establish upper bounds on the time complexity of the center-finding and median-finding algorithms. In section 6.1 we show that the center-finding algorithm makes $O(n^3 + n^2 \cdot c_h)$ moves in the worst case, where c_h is the maximum initial h-value of any process. A similar proof shows that the median-finding algorithm makes $O(n^3 \cdot c_s)$ moves in the worst case, where c_s is the maximum initial s-value of any process. This proof is omitted (see [23] for the proof). By worst case, we mean a case in which an adversary is allowed not only to maliciously select an initial global state, but is also allowed to choose an execution sequence out of many possible execution sequences. In [12] we show that, when measured in rounds, the time complexity of our center-finding algorithms is $\Theta(r)$, where r is the radius of the tree (eccentricity of the centers). A similar proof suffices to show that when measured in rounds, the time complexity of our median-finding algorithm is $\Theta(d)$ where d is the maximum distance from a median to a leaf.

6.1. Complexity of the center-finding algorithm.

To be specific, define the global state of the distributed system running the center-finding algorithm to be the n-dimensional vector of natural numbers

$$\Big(h(1), h(2), \ldots, h(n)\Big).$$

Let $\mathbf{q}$ be an arbitrary state of the system. We denote the h-value of process i in state $\mathbf{q}$ by $h(i, \mathbf{q})$. Accordingly, we use $N_h^-(i, \mathbf{q})$ to denote the multiset $N_h^-(i)$ in state $\mathbf{q}$. Given two n-dimensional vectors of natural numbers, $\mathbf{q_0}$ and $\mathbf{q}$, we write $\mathbf{q_0} \rightarrow \mathbf{q}$ if there is a move of the center-finding algorithm by some process in T that takes the system from state $\mathbf{q_0}$ to state $\mathbf{q}$. As is usual, we use $\overset{*}{\rightarrow}$ to denote the reflexive, transitive closure of $\rightarrow$. We use $\mathbf{q_0} \overset{p}{\rightarrow} \mathbf{q}$ to denote that there exists a sequence of p moves of the center-finding algorithm that takes the system from state $\mathbf{q_0}$ to state $\mathbf{q}$.

Consider two n-dimensional vectors of natural numbers $\mathbf{q_0} = (h_1, \dots, h_n)$ and $\mathbf{q_0'} = (h_1', \dots, h_n')$. We say that $\mathbf{q_0'}$ *dominates* $\mathbf{q_0}$ (or $\mathbf{q_0}$ is *dominated* by $\mathbf{q_0'}$) if $h_i \leq h_i'$ for all i, $1 \leq i \leq n$. The next three lemmas are used in Theorem 6.5 to establish an upper bound on the number of moves made by the center-finding algorithm in the worst case. The following lemma can be easily shown by induction on the number of moves required to take the system from state $\mathbf{q_0}$ to state $\mathbf{q}$ (see [23] for the proof).

LEMMA 6.1. *Suppose that $\mathbf{q_0}$ and $\mathbf{q_0'}$ are states of the center-finding algorithm such that $\mathbf{q_0}$ is dominated by $\mathbf{q_0'}$. Let $\mathbf{q}$ be a state such that $\mathbf{q_0} \overset{*}{\rightarrow} \mathbf{q}$. Then there exists a state $\mathbf{q}'$, with $\mathbf{q_0'} \overset{*}{\rightarrow} \mathbf{q}'$, that dominates $\mathbf{q}$.*

In any state $\mathbf{q}$ of the system denote the maximum h-value of any process in T, by $h_m(\mathbf{q})$. Let $h_M(\mathbf{q_0})$ be the maximum h-value of any process in any state that is reachable from $\mathbf{q_0}$. In other words,

$$h_M(\mathbf{q_0}) = \max\{h_m(\mathbf{q}) \mid \mathbf{q_0} \overset{*}{\rightarrow} \mathbf{q}\}.$$

We know that $h_M(\mathbf{q_0})$ exists because we have shown that the center-finding algorithm converges to a state in which all guards are false in a finite number of moves. This implies that every execution sequence starting at $\mathbf{q_0}$ is finite in length and hence the number of states reachable from $\mathbf{q_0}$ is also finite. Thus $h_M(\mathbf{q_0})$ is the maximum h-value that any process can achieve when the center-finding algorithm is executed starting in state $\mathbf{q_0}$. The following corollary immediately follows from the above lemma.

COROLLARY 6.2. *Let $\mathbf{q_0}$ and $\mathbf{q_0'}$ be states of the center-finding algorithm such that $\mathbf{q_0}$ is dominated by $\mathbf{q_0'}$. Then $h_M(\mathbf{q_0}) \leq h_M(\mathbf{q_0'})$.*

The following lemma uses Corollary 6.2 to establish an upper bound on the maximum h-value reachable from a state $\mathbf{q_0} = (c_h, c_h, \dots, c_h)$, in terms of the maximum h-value reachable from the state $\mathbf{q_0'} = (0, 0, \dots, 0)$. Since the proof of this lemma is similar to the proof of Lemma 6.1, it is omitted.

LEMMA 6.3. *Suppose that $\mathbf{q_0} = (c_h, c_h, \dots, c_h)$ for some $c_h \in \mathcal{N}$ and $\mathbf{q_0'} = (0, 0, \dots, 0)$. Then $h_M(\mathbf{q_0}) \leq h_M(\mathbf{q_0'}) + c_h$.*

The following lemma establishes a relationship between $h_m(\mathbf{q_0})$ and $h_M(\mathbf{q_0})$.

LEMMA 6.4. *Let $\mathbf{q_0}$ be a state of T. Then, $h_M(\mathbf{q_0}) \leq h_m(\mathbf{q_0}) + \lfloor n/2 \rfloor$.*

Proof. Let $\mathbf{q_0'} = (h_m(\mathbf{q_0}), h_m(\mathbf{q_0}), \dots, h_m(\mathbf{q_0}))$. Since $\mathbf{q_0}$ is dominated by $\mathbf{q_0'}$, by Corollary 6.2, $h_M(\mathbf{q_0}) \leq h_M(\mathbf{q_0'})$. Now let $\mathbf{q_0''} = (0, 0, \dots, 0)$. By Lemma 6.3, $h_M(\mathbf{q_0'}) \leq h_M(\mathbf{q_0''}) + h_m(\mathbf{q_0})$. This leads to the inequality $h_M(\mathbf{q_0}) \leq h_M(\mathbf{q_0''}) + h_m(\mathbf{q_0})$. It is easy to see that an upper bound on $h_M(\mathbf{q_0''})$ is $\lfloor n/2 \rfloor$. □

THEOREM 6.5. *The center-finding algorithm can make at most $O(n^3 + n^2 \cdot h_m(\mathbf{q_0}))$ moves starting from state $\mathbf{q_0}$.*

Proof. Let $\mathbf{q}$ be a state such that $\mathbf{q_0} \overset{*}{\rightarrow} \mathbf{q}$. Let $H(\mathbf{q})$ be the sum of the h-values of all the processes in state $\mathbf{q}$. Thus $H(\mathbf{q}) = \Sigma_{i \in V} h(i, \mathbf{q})$. Clearly, $0 \leq H(\mathbf{q}) \leq n h_M(\mathbf{q_0})$, for all $\mathbf{q}$, with $\mathbf{q_0} \overset{*}{\rightarrow} \mathbf{q}$. If the center-finding algorithm makes no decreasing moves, then an upper bound on the total number of increasing moves is $n h_M(\mathbf{q_0})$. But the algorithm may make up to n^2 decreasing moves and each decreasing move may reduce the value of $H()$ by at most $h_M(\mathbf{q_0})$. Hence, the total amount of reduction that the

```
{ Center-finding algorithm for vertex $i$ }
$*[(i$ is a leaf$) \wedge (h(i) \neq 0)$                                  $\longrightarrow$   $h(i) := 0$
$\square$ $(i$ is not a leaf$) \wedge (h(i) \neq \max(NC_h^-(i)))$   $\longrightarrow$   $h(i) := \max(NC_h^-(i))]$
```

FIG. 6. *The center-finding algorithm for a tree with nonunit edge costs.*

decreasing moves can cause is at most $n^2 h_M(\mathbf{q_0})$. Thus an upper bound on the total number of increasing moves is $nh_M(\mathbf{q_0}) + n^2 h_M(\mathbf{q_0})$. Using Lemma 6.4 this bound can be translated into $(h_m(\mathbf{q_0}) + \lfloor n/2 \rfloor)(n + n^2)$. Thus, including decreasing moves, the center-finding algorithm can make at most $O(n^3 + n^2 \cdot h_m(\mathbf{q_0}))$ moves. $\square$

7. Extensions. It is easy to extend the center-finding and median-finding algorithms to trees with arbitrary, positive edge costs. In particular, suppose that $c(i,j) \in \Re^+$ (the set of positive reals) is the cost associated with edge $(i,j) \in E$. Based on this, we can define the *distance* $d(i,j)$ between any pair of vertices $i, j \in V$ as follows. Let P be an arbitrary path between vertices i and j. The distance between i and j *along path* P is the sum of the costs of all the edges in P. The distance $d(i,j)$ is the smallest distance between i and j along any path between i and j. Having defined the distance $d(i,j)$ between any pair of vertices i and j, we can define the notions of eccentricity, weight, centers, and medians as in section 3.1.

We first focus on the task of modifying our original center-finding algorithm to work correctly for trees with possibly nonunit edge costs. Before we present the modified algorithm, we introduce some additional notation.

- $NC_h(i) = \{h(j) + c(i,j) \mid (i,j) \in E\}$ denotes the multiset that contains, for each neighbor j of i, the sum of the h-value of j and the cost of the edge (i,j).
- $NC_h^-(i) = NC_h(i) - \{\max(NC_h(i))\}$ denotes all of $NC_h(i)$ with one maximum element removed.

The modified version of the center-finding algorithm is presented in Figure 6. This algorithm is a generalization of our original center-finding algorithm (shown in Figure 5). This is because if $c(i,j) = 1$ for all edges $(i,j) \in E$, then $\max(NC_h^-(i)) = \max(N_h^-(i)) + 1$. The proof of total correctness of this algorithm is almost identical to the proof of total correctness of our original center-finding algorithm and is therefore omitted.

We now examine the task of generalizing the median-finding algorithm to work correctly for trees with possibly nonunit edge costs. For any edge (i,j) in T, on deleting (i,j) from T we obtain two trees, one containing i and the other containing j. Denote these trees by $T_{i,j}$ and $T_{j,i}$, respectively. Let $N_{i,j}$ and $N_{j,i}$ denote the number of vertices in $T_{i,j}$ and $T_{j,i}$, respectively. Korach, Rotem, and Santoro [27] characterize a median of a tree as follows.

LEMMA 7.1. *A vertex $i \in V$ is a median of T if and only if for all $j \in N(i)$ $N_{i,j} \geq N_{j,i}$.*

It is easy to see that even if edges have associated arbitrary, positive edge costs, the above lemma holds. Therefore, we have the following.

LEMMA 7.2. *The medians of a tree remain unchanged independent of any change in the costs of the edges.*

Thus, our median-finding algorithm, without any modification, works correctly for a tree with arbitrary, positive costs associated with its edges.

8. Conclusion. The proofs of correctness of our algorithms rely on the assumption that when a guard evaluates to true, then the corresponding action is executed in one *atomic step* by the process. However, it is possible to show that our algorithms are correct even when this assumption is relaxed in the following sense. Suppose that each action by a process i is a sequence of atomic reads of i's neighbors' local variables followed by an atomic write into process i's local variable. Now the execution of our algorithms can be viewed as a sequence in which atomic reads and writes by different processes are interleaved. The proofs of correctness of our algorithms under these weaker assumptions are essentially similar, but more tedious than those we have presented here. A sketch of the proof of the correctness of the center-finding algorithm can be found in [12].

Our time complexity analysis (section 6) reveals that the number of moves made by the center-finding algorithm depends not only on n, the number of processes in T, but also on the initial h-values of the processes. This is somewhat unfortunate because we can construct small trees with large initial h-values that could cause the center-finding algorithm to be rather slow. The median-finding algorithm has a similar problem. One possible way to overcome this problem is to assume that each process knows an upper bound on the total number of processes in T. Using this information, processes could refuse to make moves that would make their local variable unreasonably large. We also suspect that the problem could be overcome by the use of randomization.

Buckley and Harary [4] have defined various other types of "central" vertices in a graph. Some examples are *centroids*, *cores*, *pits*, *path-centers*, *p-centers*, and *p-medians*. Part of our future work will focus on extending the techniques presented in this paper for designing self-stabilizing algorithms to locate p-centers and p-medians of trees. Another aspect of this work that we wish to pursue further is applying the techniques presented in this paper to locating centers and medians in arbitrary graphs.

Acknowledgments. We gratefully acknowledge the comments of the referees that have improved the paper in many ways.

REFERENCES

[1] S. Aggarwal and S. Kutten, *Time optimal self-stabilizing spanning tree algorithm*, in Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 761, Springer-Verlag, New York, 1993, pp. 400–410.

[2] B. Awerbuch and G. Varghese, *Distributed program checking: a paradigm for building self-stabilizing distributed protocols*, in Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 258–267.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, Elsevier, New York, 1976.

[4] F. Buckley and F. Harary, *Distance in Graphs*, Addison-Wesley, Advanced Book Program, Redwood City, CA, 1990.

[5] J. E. Burns and J. Pachl, *Uniform self-stabilizing rings*, ACM Transactions on Programming Languages and Systems, 11 (1989), pp. 330–344.

[6] N. S. Chen, F. P. Yu, and S. T. Huang, *A self-stabilizing algorithm for constructing spanning trees*, Inform. Process. Lett., 39 (1991), pp. 147–151.

[7] Z. Collin and S. Dolev, *Self-stabilizing depth-first search*, Inform. Process. Lett., 49 (1994), pp. 297–301.

[8] E. W. Dijkstra, *Some beautiful arguments using mathematical induction*, Acta Inform., 13 (1980), pp. 1–8.

[9] S. Dolev, A. Israeli, and S. Moran, *Self-stabilization of dynamic systems assuming only read/write atomicity*, in Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, 1990, pp. 103–118.

[10] A. M. Farley, *Vertex centers of trees*, Transportation Science, 16 (1982), pp. 265–280.

[11] A. M. Farley and A. Proskurowski, *Computation of the center and diameter of outerplanar graphs*, Discrete Appl. Math., 2 (1980), pp. 185–191.

[12] S. Ghosh, A. Gupta, M. H. Karaata, and S. V. Pemmaraju, *Self-stabilizing dynamic programming algorithms on trees*, in Proceedings of the 2nd Workshop on Self-Stabilizing Systems, 1995.

[13] S. Ghosh, A. Gupta, and S. V. Pemmaraju, *A self-stabilizing algorithm for the maximum flow problem*, Distrib. Comput., 10 (1997), pp. 8–14.

[14] S. Ghosh and M. H. Karaata, *A self-stabilizing algorithm for coloring planar graphs*, Distrib. Comput., 7 (1993), pp. 55–59.

[15] A. J. Goldman, *Minimax location of a facility in a network*, Transportation Science, 6 (1972), pp. 407–418.

[16] M. G. Gouda and M. Schneider, *Maximum flow routing*, in Proceedings of the 2nd Workshop on Self-Stabilizing Systems, 1995.

[17] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1972.

[18] S. M. Hedetniemi, E. J. Cockayne, and S. T. Hedetniemi, *Linear algorithms for finding the jordan center and path center of a tree*, Transportation Science, 15 (1981), pp. 98–114.

[19] T. Herman, *Probabilistic self-stabilization*, Inform. Process. Lett., 35 (1990), pp. 63–67.

[20] S. C. Hsu and S. T. Huang, *A self-stabilizing algorithm for maximal matching*, Inform. Process. Lett., 43 (1992), pp. 77–81.

[21] S. T. Huang, *Leader election in uniform rings*, ACM Transactions on Programming Languages and Systems, 15 (1993), pp. 563–573.

[22] A. Israeli and M. Jalfon, *Token management schemes and random walks yield self-stabilizing mutual exclusion*, in Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, 1990, pp. 119–132.

[23] M. H. Karaata, S. V. Pemmaraju, S. C. Bruell, and S. Ghosh, *Self-Stabilizing Algorithms for Finding Centers and Medians of Trees*, Technical Report, TR 94-03, University of Iowa, Iowa City, IA, 1994.

[24] O. Kariv and S. L. Hakimi, *An algorithmic approach to network location problems.* I: *The p-centers*, SIAM J. Appl. Math., 37 (1979), pp. 513–538.

[25] O. Kariv and S. L. Hakimi, *An algorithmic approach to network location problems.* II: *The p-medians*, SIAM J. Appl. Math., 37 (1979), pp. 539–560.

[26] S. Katz and K. J. Perry, *Self-stabilizing extensions for message passing systems*, in Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, 1990, pp. 91–101.

[27] E. Korach, D. Rotem, and N. Santoro, *Distributed algorithms for finding centers and medians in networks*, ACM Transactions on Programming Languages and Systems, 6 (1984), pp. 380–401.

[28] H. S. M. Kruijer, *Self-stabilization (in spite of distributed control) in tree-structured systems*, Inform. Process. Lett., 8 (1979), pp. 91–95.

[29] R. Laskar and D. Shier, *On powers and centers of chordal graphs*, Discrete Appl. Math., 6 (1983), pp. 139–147.

[30] X. Lin and S. Ghosh, *Self-stabilizing maxima finding*, in Proceedings of the 28th Annual Allerton Conference, 1991, pp. 662–671.

[31] A. Rosenthal and J. Pino, *A generalized algorithm for centrality problems on trees*, J. ACM, 36 (1989), pp. 349–361.

[32] M. Schneider, *Self-stabilization*, ACM Computing Surveys, 25 (1993), pp. 45–67.

[33] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi, *Observations of self-stabilizing graph algorithms for anonymous networks*, in Proceedings of the 2nd Workshop on Self-Stabilizing Systems, 1995.

[34] S. Sur and P. K. Srimani, *A self-stabilizing distributed algorithm for BFS spanning trees of a symmetric graph*, Parallel Process. Lett., 2 (1992), pp. 171–179.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 615–647

AUTOMATIC METHODS FOR HIDING LATENCY IN PARALLEL AND DISTRIBUTED COMPUTATION*

MATTHEW ANDREWS†, TOM LEIGHTON‡, P. TAKIS METAXAS§, AND LISA ZHANG†

Abstract. In this paper we describe methods for mitigating the degradation in performance caused by high latencies in parallel and distributed networks. For example, given any "dataflow" type of algorithm that runs in T steps on an n-node ring with unit link delays, we show how to run the algorithm in $O(T)$ steps on any n-node bounded-degree connected network with *average* link delay $O(1)$. This is a significant improvement over prior approaches to latency hiding, which require slowdowns proportional to the *maximum* link delay. In the case when the network has average link delay d_{ave}, our simulation runs in $O(\sqrt{d_{\text{ave}}}T)$ steps using $n/\sqrt{d_{\text{ave}}}$ processors, thereby preserving efficiency. We also show how to efficiently simulate an $n \times n$ array with unit link delays using slowdown $\tilde{O}(d_{\text{ave}}^{2/3})$ on a two-dimensional array with average link delay d_{ave}. Last, we present results for the case in which large local databases are involved in the computation.

Key words. hiding latency, parallel and distributed computation, linear and two-dimensional arrays, complementary slackness

AMS subject classification. 68Q22

PII. S0097539797326502

1. Introduction.

Most papers describing algorithms for parallel or distributed computation assume a model of computation in which all the links have unit delay. Such a model is nice to work with and it is realistic for some parallel machines, but not for most. In reality, there are often substantial delays associated with some or all of the links. These delays can be caused by long wires, links that are realized by paths that go through one or more intermediate switches, wires that are required to go off-chip or off-board, communication overheads, and/or by the method which is used to prepare a packet for entry into the network. Link delays are an even greater concern for distributed machines and networks of workstations (NOWs). This is because some latencies can be very high (due to the fact that some processors can be far apart physically) and also because the variation among latencies can be high (since some processors may be very close or even part of the same tightly coupled parallel machine).

1.1. Traditional approaches.

Since communication latency is an important factor in the performance of a parallel or distributed algorithm, several methods have been devised in an attempt to compensate for latency. The simplest of these methods is to slow down the computation to the point where the latency is accommodated. This approach is most commonly used at the circuit level, where the clock speed

*Received by the editors August 27, 1997; accepted for publication (in revised form) August 11, 1998; published electronically November 23, 1999.
http://www.siam.org/journals/sicomp/29-2/32650.html

†Bell Laboratories, Murray Hill, NJ (andrews@research.bell-labs.com, ylz@research.bell-labs.com). The research of the first author was supported by NSF contract 9302476-CCR and ARPA contract N00014-95-1-1246. The work of these authors was performed while at MIT.

‡Department of Mathematics and Laboratory for Computer Science, MIT, Cambridge, MA (ftl@math.mit.edu). The work of this author was supported by ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246.

§Department of Computer Science, Wellesley College, Wellesley, MA (pmetaxas@wellesley.edu). The work of this author was supported by NSF contract 9504421-CCR and ARPA contract N00014-95-1-1246.

is set to be slow enough so that all of the data has time to reach its destination before the next step begins. This means that the circuit needs to be slowed down to accommodate the highest latency. Such an approach is clearly less than desirable in the context of a NOW with high-latency links.

An alternative approach is to organize the network in a hierarchical fashion so that the latencies are consistent with the hierarchy. For example, the CM-5 [1, 14] is organized into a fat tree and the KSR consists of two levels of nested rings. In both cases, the highest latency links are segregated into the top levels of the network hierarchy. This type of architecture works well for applications in which most of the computation is local since local computation can proceed using the low-level low-latency links. Only rarely, it is hoped, would the high-latency links be needed. Thus, only certain steps of the computation would be slow. Unfortunately, this approach is not suitable for scenarios where the network is unstructured (which is often the case for a NOW) or when the underlying application requires frequent communications through the high-level links.

Redundant computation is another approach that has been used in the past [6, 11, 13] to hide the effects of latency. Here the idea is to avoid latency by recomputing data locally instead of waiting to receive it through a high-latency link.

Probably the most generally applicable method of hiding latency is the approach known as *complementary slackness*. The idea behind this approach is to load each processor with enough work so that it stays productive while waiting for data to be supplied by the network. There are many implementations and incarnations of this method. For example, each processor in the CRAY YMP C-90 keeps busy by operating on a pipeline of 128 64-bit words. Processors on the HEP machine [21] swapped between unrelated threads while waiting for the data. The CM-1 and CM-2 were designed to simulate much larger virtual machines so that a single processor would perform the computation of many virtual processors [4, 22]. The technique also forms a critical component of Valiant's bulk synchronous model of parallel computing [23, 24] and it has been employed in several papers [3, 10, 11, 15, 20].

Unfortunately, in all of the preceding examples, it is incumbent on the programmer to provide the slackness or pipelining needed or to determine what part of the computation must be redundantly duplicated and by which processors to overcome the latencies in the network. Even in the scenario where a large virtual network is being simulated on a small parallel machine, it is incumbent on the programmer to find the parallelism necessary to efficiently implement the algorithm on a (potentially very large) virtual network.

The goal of our research is to devise *automatic* methods for hiding latency. Our approach falls within the broad class of methods based on complementary slackness, but it does not require the programmer to provide slackness, pipelines, or greater parallelism in order to hide the latency. Rather, our methods attempt to find the slackness automatically. By automatically finding the slackness, we hope to allow the programmer to assume that there are uniform delays on each link of the network, thereby easing the task of writing code. Moreover, our methods will enable us to automatically convert a program that was written for a well-structured unit-delay machine into a program that will run with minimal degradation in performance on a network with potentially large and variable latencies, at least for certain classes of networks.

1.2. Model and problem. We consider the problem of simulating a network G with unit-delay links on a network H with arbitrary delays on its links. We refer to G

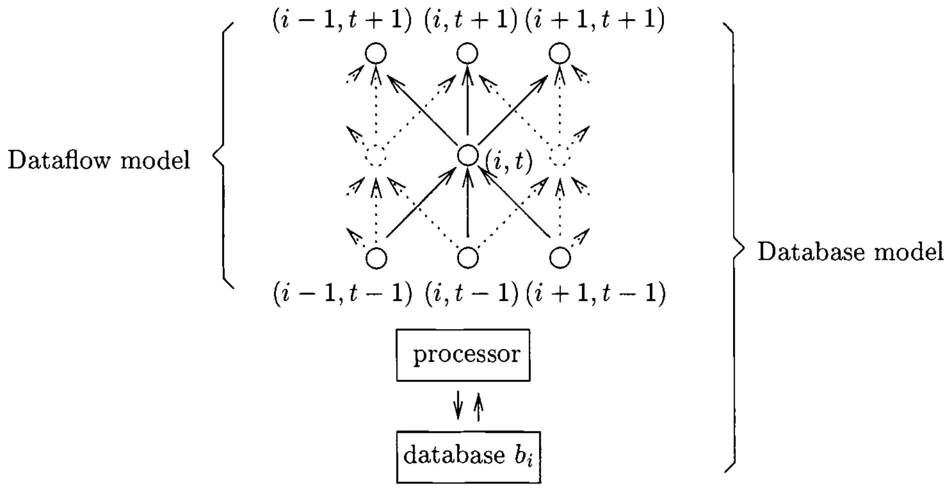

Dataflow model
Database model
$(i-1,t+1)$ $(i,t+1)$ $(i+1,t+1)$
(i,t)
$(i-1,t-1)$ $(i,t-1)$ $(i+1,t-1)$
processor
database b_i

FIG. 1. *The computation pebbles created by a guest linear array.*

as the *guest* and H as the *host.* Let $g_1, g_2, \ldots$ be the processors of G and $p_1, p_2, \ldots$ be the processors of H. We shall use *pebbles* to record the computations performed by the guest processors. In particular, pebble (i,t) represents the tth step of computation by processor g_i. In a *simulation* of G, H carries out the same step-by-step computation as G. In other words, H simulates G by computing every pebble created by G in an order that preserves the "dependency" of the pebbles. Our goal is to provide methods that would allow H to simulate G with a minimum amount of *slowdown* when G is used in a general-purpose way. Formally, slowdown is the ratio of T_H to T_G, where T_G is the time taken by G to compute all the pebbles and T_H is the time taken by H to simulate this computation. Two computation models are studied here: the *dataflow model* and the *database model.*

Dataflow model. In the dataflow model, each computation solely depends on the computation of the previous step. Creating a pebble (i,t) involves two time units. The first time unit is for communication, where g_i obtains pebbles of the form $(j,t-1)$ from all its neighbors g_j. The second time unit is for computation, where g_i performs computation based on pebbles $(j,t-1)$ and records the result in pebble (i,t). Take an example of an n-node guest linear array. In $2T$ time steps, G creates $n \times T$ pebbles, where pebble (i,t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1,t-1)$, $(i,t-1)$, and $(i+1,t-1)$. (See Figure 1.) Any host processor p can compute pebble (i,t) as long as p has the information in pebbles $(i-1,t-1)$, $(i,t-1)$, and $(i+1,t-1)$, either by directly computing these pebbles or by receiving them from neighboring processors.

The dataflow model is applicable to many computations such as matrix operations, Fourier transform, sorting, algorithms for computational geometry, etc. A large number of examples can be found in [12].

Database model. In the database model each guest processor g_i has a potentially large local memory that may be accessed and updated by g_i during each step. We refer to the local memory of g_i as the *database*, b_i. Each computation not only depends on the computation of the immediate past but also the state of the database. For example, let G be a linear array. To create pebble (i,t), g_i first communicates with its neighbors, then performs computations based on pebbles $(i-1, t-1)$, $(i, t-1)$, and $(i+1, t-1)$ and the current state of database b_i. Last, g_i updates database b_i. Hence, creating a pebble involves two time units as in the dataflow model: one for communication and one for computation and recording.

In the database model, a pebble not only records the result of a computation but also the *changes* to the database incurred by this computation. To emphasize, a pebble does not contain a snapshot of the whole database but rather the changes incurred by one computation. Therefore, a pebble has small size and can be passed along links.

In order to simulate G on H, we assume that the initial contents of each database can be copied *before* the computation begins (thereby allowing redundant computations), but that the large size of a database makes it impractical to transmit a copy of a database through the network *during* the computation. Suppose processor p of H copies databases b_i and b_j; then p only has access to b_i and b_j and hence can only compute pebbles of the form (i, t) and (j, t) for $t \geq 1$. Moreover, if both processors p and q decide to copy b_i, then p and q each maintains a copy of b_i, and each looks up and updates its own copy. If p is to compute pebble (i, t), then p needs an updated copy of the database that includes all the changes incurred by the computations (i, t') for all $t' < t$. Hence, p must either have directly computed all the pebbles (i, t') or else have received the information from its neighbors.

Unlike the dataflow model, the database model captures a scenario where the computation performed by a processor depends on the state of a local memory or where part of the computation performed by a processor is to update its local memory. These situations could be critical in some applications involving a network of workstations.

Bandwidth. The guest network G has unit bandwidth on each link. This allows each pebble to be passed along a unit-delay link of G in one time step. In our simulation we assume that the link bandwidth of the host network H is w. That is, P pebbles can be passed along a d-delay link of H in $d + \lceil \frac{P}{w} \rceil - 1$ steps by pipelining. In many cases of our study, it is sufficient to assume that the host and the guest have comparable link bandwidth; i.e., w is a constant. However, in certain situations the bandwidth needs to be $\tilde{O}(\log n)$. Otherwise, we pay an extra factor of $\tilde{O}(\log n)$ in the slowdown. The details are discussed in sections 3.2.4 and 4.2.4.

1.3. Results. Table 1 summarizes our results. In the table, n is the size of the guest, d_{ave} is the average delay of the host and "Bd-deg" stands for bounded-degree. The ratio of n and the slowdown is the size of the host, since all the simulations are *work efficient*; i.e., it takes the guest and the host the same amount of work to compute the same result, where *work* is the product of the number of processors used and the running time.

The first two results in Table 1 are proved in terms of linear arrays. An n-node unit-delay ring is essentially the same as an n-node unit-delay linear array, since the latter can simulate the former with a slowdown of 2 [12]. Result 1 is asymptotically optimal in some cases. In addition, we also have a constant-approximation algorithm for simulating rings and linear arrays in the dataflow model. Results 2 and 3 are optimal up to a polylogarithmic factor in some cases. Result 3 is for a worst-case model. When the delays on the host are randomly arranged, the bound can be improved to $O(d_{\text{ave}}^{2/3})$. Results 4 and 5 are easy generalizations of results 1 and 2, respectively. Sections 2 and 3 present latency hiding methods for the dataflow model.

Section 4 concentrates on the database model.

The methods for latency hiding in the two computation models are substantially different. For example, we make heavy use of redundant computation in the database model, whereas redundancy is apparently not useful for the dataflow model.

Our bounds indicate that hiding latency in the database model is more difficult than in the dataflow model. Intuitively, this is because computation in the dataflow model is processor independent and hence can be done by any processor with the information of the previous computation. In the database model, computation can only be done by the processors with the right databases. One cannot afford to pass large databases across the links with limited bandwidth, because this will cause high slowdown. One also cannot afford to keep many copies of the databases, because memory is expensive and keeping every copy of the databases updated is difficult.

In section 4, we also establish limits on the degree to which the high latency can be mitigated when each database is allowed a small number of copies. For example, if each database has only one copy, we show that the slowdown can be as much as $d_{\max}$ even if d_{ave} is a constant and the best simulation is used. When each database has at most two copies and each host processor copies a constant number of databases, we give an example of a host whose average delay is a constant, but for which the slowdown has a lower bound of $\Omega(\log n)$. These results demonstrate that it is easier to overcome latencies in dataflow types of computations than in computations that require access to large local databases.

1.4. A related scheduling problem. The problem of latency hiding in the dataflow model can be viewed as the following scheduling problem. The pebbles created by the guest network together with their dependencies form a directed acyclic graph (dag), whose nodes represent computational tasks of equal execution time, and whose arcs represent precedence. All these tasks are to be computed by the processors in a given host network. If the same host processor computes two tasks of direct dependence, no communication cost is incurred. Otherwise, there is a communication cost between the two host processors that compute these two tasks, and this cost is equal to the total delay between the processors in the host network. The goal here is to schedule the dag (with possible repetitions of the nodes) using the given host processors so as to minimize the *makespan*, i.e., the total time taken to execute all the tasks.

A variation of the above scheduling problem has been studied. Here, we are given any task dag (not necessarily created by a guest network in the dataflow model). All the arcs in the dag are associated with a fixed quantity that indicates the communica-

TABLE 1
Result summary.

	Guest	Host	Model	Order of slowdown
1	Ring/linear array	Bd-deg network	Dataflow	$\sqrt{d_{\text{ave}}}$
2	Ring/linear array	Bd-deg network	Database	$\sqrt{d_{\text{ave}}}\log^3 n$
3	2-D array	2-D array	Dataflow	$d_{\text{ave}}^{2/3}\log^{5/3} n$
4	2-D array	Bd-deg network	Dataflow	$n^{1/4}(\sqrt{d_{\text{ave}}}+n^{1/4})$
5	2-D array	Bd-deg network	Database	$n^{1/4}\log^3 n(\sqrt{d_{\text{ave}}}+n^{1/4})$

tion cost. Note that, unlike our problem, the communication cost here is the same for any processor pair. In [18] Papadimitriou and Ullman studied an $n \times n$ grid dag (which they called a diamond dag). They showed a nontrivial time-communication tradeoff and gave an asymptotically optimal schedule. Their result was similar to the special case of our Result 1 stated in section 1.3, where all the link delays in our host network are the same. In [19] Papadimitriou and Yannakakis presented a 2-approximation algorithm for general dags where an unlimited number of processors could be used. For well-known families of dags such as the full binary tree, the diamond dag, and the fast Fourier transform, only a finite number of processors were needed and their approximation algorithms were optimal (or near optimal). Redundant computation was used in [19].

Dag scheduling has been studied in other papers, including [2, 5, 7, 8, 9, 16, 17]. Some variations of the problem are the cases in which the dags are limited to certain topologies, the task nodes require different execution times, arcs require different communication time, and processors have different processing powers.

2. Dataflow model—Linear arrays.

We begin our presentation with the methods for hiding latency in linear arrays. Our basic approach is to transfer a process that involves a two-way communication to a process that involves one-way communication only. (This idea is also essential for simulating two-dimensional arrays in section 3.) We present an asymptotically tight bound on the slowdown for linear arrays. All the results for linear arrays are applicable to rings.

2.1. Average delay—An upper bound.

Let the network G be an n-processor guest linear array with unit delay on all the edges. Let the network H be an n-processor host linear array with arbitrary delays, where d_i is the delay on the ith edge of H. As discussed in section 1.2, in $2T$ time steps G creates $n \times T$ pebbles, where pebble (i, t), for $1 < i < n$ and $1 < t \leq T$, depends on pebbles $(i-1, t-1)$, $(i, t-1)$, and $(i+1, t-1)$. We first present algorithm Stripe in which H simulates G with a slowdown of $O(d_{\text{ave}})$, where $d_{\text{ave}} = \sum_{k=1}^{n-1} d_k/(n-1)$ is the average delay of H.

Consider the first $n/2$ rows of pebbles created by G. Let L be the triangle formed by pebbles (i, t), where $i + t \leq n + 1$. Let R be the triangle formed by pebbles (i, t), where $i \leq t$. (See Figure 2.) In Stripe, H first simulates the bottom half of L and then the bottom half of R. At this point every pebble in the first $n/2$ rows is simulated. If the entire computation of G is partitioned into groups each of which consists of $n/2$ rows of pebbles, then H can repeat the process and simulate every group in a similar manner.

To simulate the bottom half of L, the computation pebbles of G are divided into n slanted stripes, and each processor of H simulates one stripe. (See Figure 2.) In particular, processor p_i of H simulates a stripe consisting of pebbles $(i - t + 1, t)$ for $1 \leq t \leq i$ and $t \leq n/2$. Note that in the original computation by G, processor g_i depends on both g_{i-1} and g_{i+1}. However, in the simulation by H p_i depends on p_{i-1} and p_{i-2}. Hence, Stripe transforms a process that involves two-way communication into a process that involves only one-way communication.

Lemma 2.1. *Processor p_i $(1 \leq i \leq n)$ is able to compute pebble $(i - t + 1, t)$ at step $t + \sum_{k=1}^{i-1} d_k$.*

Proof. We use induction on i. The base case for p_1 is obvious. Pebble $(i-t+1, t)$ depends on pebbles $(i-t, t-1)$, $(i-t+1, t-1)$, and $(i-t+2, t-1)$, which are computed by processors p_{i-2}, p_{i-1}, and p_i, respectively. By induction these three pebbles are computed at step $(t-1) + \sum_{k=1}^{i-3} d_k$, $(t-1) + \sum_{k=1}^{i-2} d_k$, and $(t-1) + \sum_{k=1}^{i-1} d_k$,

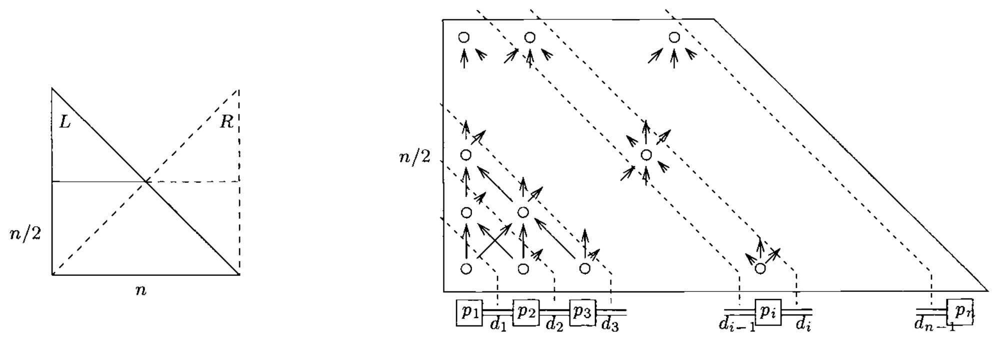


FIG. 2. *(Left) Triangles L and R. (Right) Algorithm* STRIPE. *Each slanted stripe is simulated by one processor of H. Arrows correspond to communications. Dashed lines correspond to the delays d_i encountered by communications.*

respectively. It follows that $(i-t+1,t)$ can be computed at step $t+\sum_{k=1}^{i-1} d_k$. □

Hence, pebbles $(i+1,n/2)$, for $0 \le i \le n/2$, are computed at steps $n/2+\sum_{k=1}^{i+n/2-1} d_k$, and so the bottom half of L is simulated in $n/2+\sum_{k=1}^{n-1} d_k$ steps by H. The bottom half of R is simulated in a similar manner. (Note that the intersection of R and L only needs to be computed once.) Thus, H has completed simulating the first $n/2$ rows of pebbles created by G. To continue the simulation, each pebble $(i,n/2)$ is passed to processor p_i. With pipelining, this can be done in $\sum_{k=1}^{n-1} d_k$ steps. The next $n/2$ and every subsequent $n/2$ rows of pebbles can be simulated in a similar manner. Therefore, the slowdown is upper bounded by

$$s = \frac{2\cdot(n/2+\sum_{k=1}^{n-1} d_k)+\sum_{k=1}^{n-1} d_k}{n/2} = O(d_{\text{ave}}).$$

2.2. A better upper bound. To get a better upper bound on the best achievable slowdown, we use the idea of "complementary slackness" in our new algorithm called FATSTRIPE. Each host processor is loaded with enough work to balance out the communication time. Suppose FATSTRIPE uses an interval of m processors to carry out the simulation. For simplicity, assume that this interval consists of processors $p_1,\dots,p_m$. The bottom half of L is divided into m slanted stripes, each of which has width $\ell = n/m$. Again, p_i computes every pebble in stripe i. (See Figure 3.) Within each stripe i, p_i first computes all the pebbles in the bottom row and then moves up.

LEMMA 2.2. *Processor p_i finishes simulating stripe i by step $\ell n/2+\sum_{k=1}^{i-1} d_k$.*

Proof. We inductively show that p_i can compute the pebbles in the xth row of stripe i by time step $\ell x+\sum_{k=1}^{i-1} d_k$. The base of the induction holds trivially for $i=1$ and $x=1$, since processor p_1 does not depend on other processors and pebbles in the first row do not depend on other pebbles. Let us consider the pebbles on the $(x+1)$st row of stripes $i+1$ for $x \ge 1$ and $i \ge 1$. These pebbles could only depend on pebbles on the xth row of stripe $i-1$, i, and $i+1$, which can be computed by processors p_{i-2}, p_{i-1}, and p_i by steps $\ell x+\sum_{k=1}^{i-3} d_k$, $\ell x+\sum_{k=1}^{i-2} d_k$, and $\ell x+\sum_{k=1}^{i-1} d_k$, respectively, by induction. Hence, p_i is able to receive all the information necessary to compute its $(x+1)$st row by step $\ell x+\sum_{k=1}^{i-1} d_k$ and therefore finish computing the $(x+1)$st row by step $\ell(x+1)+\sum_{k=1}^{i-1} d_k$. Since each stripe contains at most $n/2$ rows, p_i finishes simulating stripe i by step $\ell n/2+\sum_{k=1}^{i-1} d_k$. □

Hence, the slowdown is $O(n/m+\sum_{k=1}^{m-1} d_k/n)$ in simulating the first $n/2$ rows of pebbles. All the subsequent $n/2$ rows can be simulated in a similar manner. To

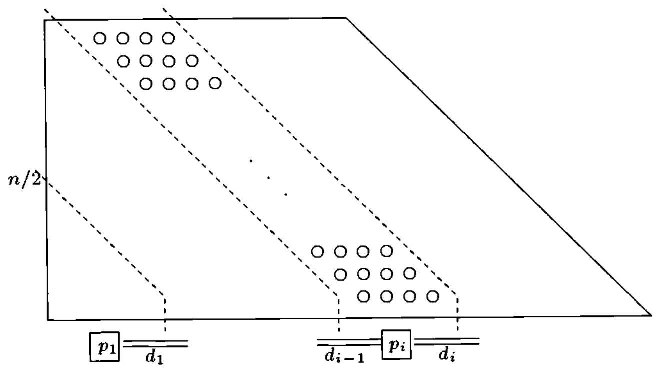


FIG. 3. *Algorithm* FATSTRIPE. *Processor p_i simulates stripe i, which has width $\ell = n/m$. (In the figure, $\ell = 4$.) All the pebbles in stripe i are computed by time step $\ell n/2 + \sum_{k=1}^{i-1} d_k$.*

minimize the slowdown, FATSTRIPE uses the interval I (with m_I processors and d_I average delay) that minimizes the quantity $n/m_I + d_I m_I/n$. Therefore, Theorem 2.3 follows.

THEOREM 2.3. FATSTRIPE *achieves slowdown of* $\min_{intervals\ I} O(n/m_I + d_I m_I/n)$.

In the case when $\sqrt{d_{\text{ave}}} \le n$, there exists an interval I with $M_I = n/\sqrt{d_{\text{ave}}}$ processors and average delay $d_I \le d_{\text{ave}}$ by the pigeon-hole principle. Theorem 2.3 implies that the slowdown is $O(\sqrt{d_{\text{ave}}})$ when M_I simulates G. In the case when $\sqrt{d_{\text{ave}}} > n$ a single host processor is used to carry out the simulation, which incurs a slowdown of $n = O(\sqrt{d_{\text{ave}}})$. The simulation is work efficient in both cases. Therefore, Corollary 2.4 holds.

COROLLARY 2.4. FATSTRIPE *efficiently simulates G on H and achieves a slowdown of $O(\sqrt{d_{ave}})$, where d_{ave} is the average delay of H.*

Let us consider the effect of bandwidth on the slowdown. In FATSTRIPE as long as the stripe width is at least 2, then pebbles cross the edges one at a time by using pipelining. In STRIPE (i.e., FATSTRIPE with stripe width 1) at most two pebbles may cross an edge at the same time. Therefore, it is sufficient for the host bandwidth to be twice as large as that of the guest bandwidth. Otherwise, we pay another factor of 2 in the slowdown.

2.3. A matching lower bound. We proceed to show that the upper bound, $\min_I O(n/m_I + d_I m_I/n)$, in Theorem 2.3 is asymptotically tight by showing that $\min_I \max\{n/2m_I, d_I m_I/2n\}$ is a lower bound on the best achievable slowdown even if we allow *redundant computation.* Note that with redundant computation, a pebble may be computed by several host processors. This technique makes it more likely for the host to simulate the guest efficiently. However, we show below that redundancy does not help in this case.

LEMMA 2.5. *The top pebble,* $(1, n)$ *of triangle L, cannot be computed at a time step earlier than*

$$\tau = \min_{intervals\ I} \max\{n^2/2m_I, d_I m_I/2\}.$$

Proof. We consider how the pebbles in L are computed in some simulation of G by H. In particular, we build a ternary tree T to keep track of the processors that have "effectively" computed the pebbles in L. The top pebble $(1, n)$ has to be computed by some processor of H. Call this processor q. (If more than one processor of H has computed $(1, n)$, then we pick any one of them to be q.) We label the root of tree T with $q^{(1,n)}$. Let u be a processor that has computed $(1, n-1)$ *and* has passed this information to q, and let v be a processor that has computed $(2, n-1)$ *and* has passed this information to q. (Note that other processors may compute $(1, n-1)$ and $(2, n-1)$. We are only concerned with processors that pass information to q.) Now label the children of $q^{(1,n)}$ with $u^{(1,n-1)}$ and $v^{(2,n-1)}$. We proceed to construct the children of $u^{(1,n-1)}$ and $v^{(2,n-1)}$. In general, node $a^{(i,t)}$ in T has children $b^{(i-1,t-1)}$, $c^{(i,t-1)}$, and $d^{(i+1,t-1)}$ if the following holds. Processors a, b, c, and d compute pebbles (i, t), $(i-1, t-1)$, $(i, t-1)$, and $(i+1, t-1)$, respectively, *and* a receives the values of $(i-1, t-1)$, $(i, t-1)$, and $(i+1, t-1)$ from b, c, and d before a is able to compute (i, t). The leaves of T are nodes of the form $p^{(i,1)}$. The important observation is the following. If $p^{(i,t)}$ is a node in T, then information has to be passed from processor p to q in H. The total delay from p to q lower bounds the number of steps in the simulation.

Let J be the smallest interval that contains all the processors appearing in tree T. If processors x and y are at the two ends of J, then there exist two nodes of the form $x^{(i_x,t_x)}$ and $y^{(i_y,t_y)}$ in T. Hence, information has to be passed from x and y to q in H. This takes at least $d_J m_J/2$ steps, and pebble $(1, n)$ therefore cannot be computed at a step earlier than $d_J m_J/2$. Since m_J processors are computing $n^2/2$ pebbles, a work argument shows that $(1, n)$ cannot be computed before step $n^2/2m_J$. Hence, $(1, n)$ cannot be computed at a step earlier than $\tau = \min_I \max\{n^2/2m_I,\ d_I m_I/2\}$. ∎

It follows that the slowdown in simulating triangle L is lower bounded by τ/n. By a similar argument to Lemma 2.5 none of the pebbles (i, n), for $1 \le i \le n$, can be computed at a time step earlier than τ. By repeating this argument the first kn rows of G cannot be simulated in time less than $k\tau$. Therefore, we obtain Theorem 2.6.

THEOREM 2.6. *The slowdown of any simulation of an n-node guest linear array G by a host linear array H is lower bounded by* $\min_I \Theta(n/m_I + d_I m_I/n)$*, where I is a subarray of H and has m_I processors and average delay d_I. Hence,* FATSTRIPE *is optimal up to a constant factor.*

2.4. Simulating linear arrays on general networks. We now consider simulating a linear array G on a general n-node network H with average delay d_{ave}. We first embed a linear array $\mathcal{H}$ in H and then use $\mathcal{H}$ to carry out the simulation of G.

LEMMA 2.7. *Let H be a connected n-node network with arbitrary topology. Then an n-node linear array $\mathcal{H}$ can be one-to-one embedded in H such that every edge of H is used at most twice in $\mathcal{H}$.*

Proof. Our proof follows the approach of Theorem 3.15 in [12, page 470]. We include the proof here for completeness. It is sufficient to embed a linear array $\mathcal{H}$ in a spanning tree of H. The proof proceeds by induction on the height of the tree with the following inductive hypothesis. For any child u of the root v, there is a one-to-one embedding of a linear array in the tree such that v and u form two endpoints of the array, the edge uv is used at most once, and all other edges of the tree are used at most twice. (Note that we treat all the edges as undirected.)

Let T be any spanning tree of H. The base of the induction in which T is a single node, i.e., the height is 0, is trivial. Otherwise, let v be the root of T and u be any child of v. We label the children of v as $u_1, \ldots, u_d$ and assume $u = u_d$ without loss of generality. We place the first node of the linear array at v, and we place the second node of the array at any child w of u_1 (if any) using edges vu_1 and u_1w. Next, we inductively place the nodes of the array in each node of the subtree of T rooted at u_1, making sure that the last node is placed at u_1, the edge u_1w is used at most once, and that all other edges in the subtree are used twice. Therefore, edge u_1w is used at

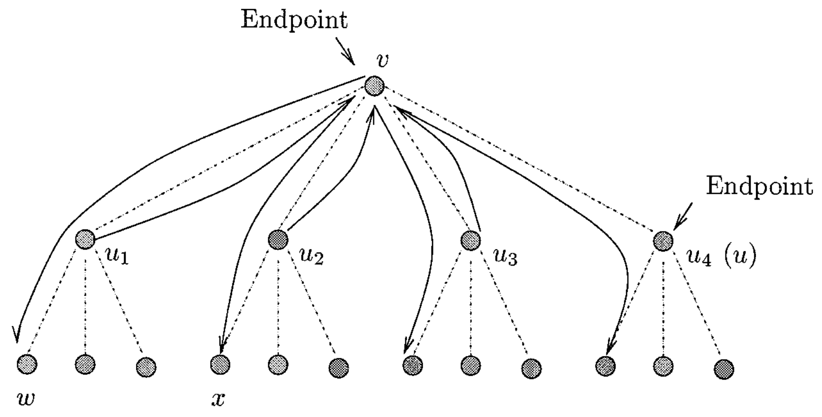


FIG. 4. *Embed a linear array one to one in a tree such that each tree edge is used at most twice. The dotted lines indicate tree edges and the solid lines indicate array edges.*

most twice in total.

We place the next node of the linear array at any child x of u_2 (if any), using edges u_1v, vu_2 and u_2x. Again, we inductively place nodes of the linear array in the subtree rooted at u_2 such that u_2 and x are endpoints. We continue in this fashion. At the last subtree rooted at u, we enter this subtree at a child of u (if any) and exit at u. This completes the embedding of the linear array. Our lemma follows from the observation that the linear array has endpoints v and u, edges vu_1, …, vu_{d-1} are used twice, and vu_d is used once. (See Figure 4.) □

Since H has n nodes and degree δ, H has at most $\delta n/2$ edges, and therefore the total delays on all edges of H are at most $\delta d_{\text{ave}} n/2$. By Lemma 2.7, $\mathcal{H}$ uses each edge of H at most twice. Hence, the total delays on all edges of $\mathcal{H}$ is at most $\delta d_{\text{ave}} n$, and the average delay of $\mathcal{H}$ is at most δd_{ave}. By Corollary 2.4, $\mathcal{H}$ can simulate G with a slowdown of $O(\sqrt{\delta d_{\text{ave}}})$. When H has bounded degree, i.e., $\delta = O(1)$, we have Theorem 2.8.

THEOREM 2.8. *A bounded-degree host network with average delay d_{ave} can efficiently simulate an n-processor guest linear array with a slowdown of $O(\sqrt{d_{ave}})$.*

Theorem 2.8 does not hold when H has unbounded degree. Consider the following example. Let H be a linear array of $\sqrt{n}$ cliques, in which each clique contains $\sqrt{n}$ nodes. If a clique edge has delay 1 and an edge connecting two adjacent cliques has delay n, then H has $d_{\text{ave}} < 4$. Suppose m connected cliques are used to simulate n steps of G. Lemma 2.5 implies a slowdown of $\min_m \max\{\sqrt{n}/2m, m/2\}$ in simulating every n steps of computation by the guest. The first term follows from a work argument, since $m\sqrt{n}$ processors are in m cliques. The second term comes from the communication delay, since a linear array embedded in these m connected cliques has a total delay of at least mn. Hence, the slowdown is at least $\min_m \max\{\sqrt{n}/2m, m/2\}$, which is $\Omega(n^{1/4})$, whereas the average delay is a constant.

3. Dataflow model—Two-dimensional arrays.

In this section we present methods for hiding latency in two-dimensional arrays. The analysis here is substantially more complex than that for the one-dimensional case. We focus on simulating a two-dimensional array on a two-dimensional array. Section 3.1 generalizes the approach for the linear arrays. Section 3.2 introduces some new mechanism to improve the bound. Section 3.3 discusses the case when the delays are randomly arranged.

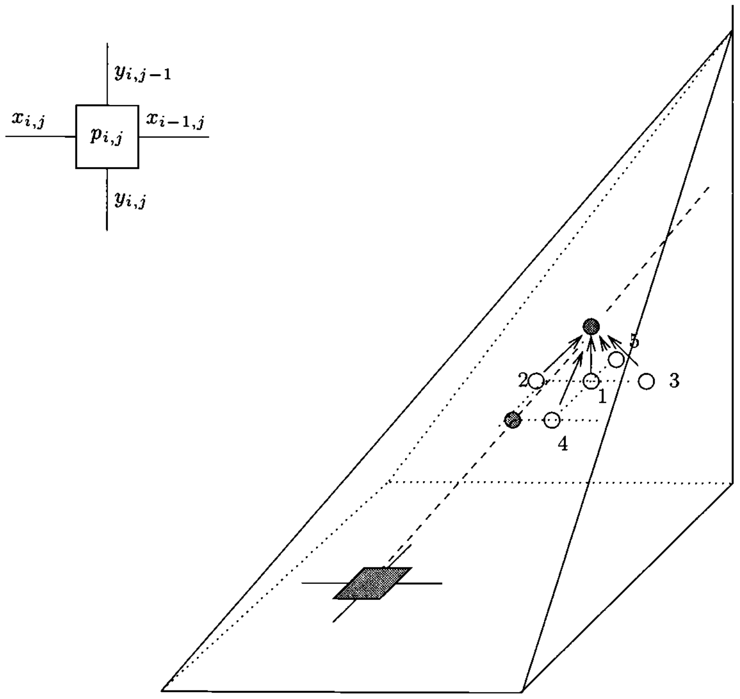


FIG. 5. *Algorithm* 2D-RAY. *In pyramid P_1 the dashed line represents the ray of pebbles computed by processor $p_{i,j}$ (which is shown in the upper left corner). Two of those pebbles, computed at times t and $t-1$, are shown shaded. The five numbered pebbles are those upon which $(i-t+1, j-t+1, t)$ depends.*

3.1. An analogue of the one-dimensional case. Let the guest network G be an $n \times n$ two-dimensional array with unit delay on all the edges. Let the host network H be an $n \times n$ two-dimensional array with arbitrary delays. Let $x_{i,j}$ be the delay between processors $p_{i,j}$ and $p_{i+1,j}$ of H for $1 \le i \le n-1$ and $1 \le j \le n$, and let $y_{i,j}$ be the delay between $p_{i,j}$ and $p_{i,j+1}$ of H for $1 \le i \le n$ and $1 \le j \le n-1$. The tth step of computation by processor $g_{i,j}$ of G is recorded in pebble (i,j,t). In $2T$ steps, G creates $n \times n \times T$ pebbles, where pebble (i,j,t), for $1 < i,j < n$ and $1 < t \le T$, depends on $(i-t+1, j-t+1, t-1)$, $(i-t, j-t+1, t-1)$, $(i-t+2, j-t+1, t-1)$, $(i-t+1, j-t, t-1)$, and $(i-t+1, j-t+2, t-1)$.

Consider the first $n/2$ steps of computation by G. We define four pyramids P_1, P_2, P_3, and P_4 analogous to the left and right triangles in the linear array case. All four pyramids have the square, defined by vertices $(1,1,1)$, $(1,n,1)$, $(n,1,1)$, and $(n,n,1)$, as their bases. The top vertices of P_1, P_2, P_3, and P_4 are $(1,1,n)$, $(1,n,n)$, $(n,1,n)$, and (n,n,n), respectively. Note that the bottom half of the four pyramids contains all the pebbles created by G for the first $n/2$ steps of computation.

Algorithm 2D-RAY is a two-dimensional analogue of STRIPE. To simulate the first $n/2$ steps of computation of G, 2D-RAY simulates P_1, P_2, P_3, and P_4 one by one. Pyramid P_1 is divided into n^2 rays, each of which is simulated by one processor of H. In particular, processor $p_{i,j}$ of H simulates ray $R_{i,j}$, consisting of pebbles $(i-t+1, j-t+1, t)$ for $1 \le t \le \min\{i, j, n/2\}$. (See Figure 5.) When every pebble for the first $n/2$ steps of computation of G is simulated, 2D-RAY repeats the process and simulates the next $n/2$ steps of computation. In the following we bound the slowdown in terms of the total delay on *monotone paths*, where a monotone path travels in two directions, up and right. Let the length of a path be the total delay on the path, and let $D_{i,j}$ be the length of the longest monotone path from processor $p_{1,1}$ to $p_{i,j}$ in H. We have Lemma 3.1.

LEMMA 3.1. *Processor $p_{i,j}$ of H is able to compute pebble $(i-t+1, j-t+1, t)$ at step $D_{i,j}+t$.*

Proof. We use induction on the indices (i,j) of the processors. The base of the induction for $p_{1,1}$ is obvious. Pebble $(i-t+1, j-t+1, t)$ depends on pebbles $(i-t+1, j-t+1, t-1)$, $(i-t, j-t+1, t-1)$, $(i-t+2, j-t+1, t-1)$, $(i-t+1, j-t, t-1)$, and $(i-t+1, j-t+2, t-1)$, which are computed by processors $p_{i-1,j-1}$, $p_{i-2,j-1}$, $p_{i,j-1}$, $p_{i-1,j-2}$, and $p_{i-1,j}$, respectively. (See Figure 5.) By induction, these five pebbles are computed at steps $D_{i-1,j-1}+(t-1)$, $D_{i-2,j-1}+(t-1)$, $D_{i,j-1}+(t-1)$, $D_{i-1,j-2}+(t-1)$, and $D_{i-1,j}+(t-1)$, respectively. It follows that pebble $(i-t+1, j-t+1, t)$ can be computed at step $\max\{D_{i-1,j}+x_{i-1,j}, D_{i,j-1}+y_{i,j-1}\}+t = D_{i,j}+t$. ∎

Hence, 2D-RAY simulates pyramid P_1 in $D_{n,n}+n$ steps. Since P_2, P_3, and P_4 can be simulated similarly, 2D-RAY simulates the first $n/2$ steps of computation of G in $O(D_{n,n}+n)$ steps. The simulation is repeated for every $n/2$ steps of computation of G. Therefore, Lemma 3.2 holds.

LEMMA 3.2. *Algorithm* 2D-RAY *achieves a slowdown of $O(D_{n,n}/n)$, where $D_{n,n}$ is the length of the longest monotone path in H.* ∎

Unfortunately, $D_{n,n}$ can be large compared with d_{ave}, the average delay of H. In the worst case $D_{n,n}$ can be $\Theta(n^2 d_{\text{ave}})$, implying a slowdown of $\Theta(n d_{\text{ave}})$. We introduce algorithm FATRAY, a two-dimensional analogue of FATSTRIPE, to achieve a slowdown that is often better than $O(D_{n,n}/n)$. Pyramid P_1 is divided into m^2 rays, each of which has size $\ell \times \ell = \frac{n}{m} \times \frac{n}{m}$. FATRAY uses an $m \times m$ contiguous subarray of processors in H to carry out the simulation. For simplicity, assume FATRAY uses processors $p_{i,j}$ $(1 \le i,j \le m)$. Again, $p_{i,j}$ computes every pebble in ray $R_{i,j}$, and $p_{i,j}$ first computes all the pebbles on the bottom plane and then moves up. The following lemma is analogous to Lemma 2.2.

LEMMA 3.3. *Processor $p_{i,j}$ finishes simulating ray $R_{i,j}$ by using step $\ell^2 n/2 + D_{i,j}$.*

Proof. As in Lemma 2.2 we can inductively show that $p_{i,j}$ can compute all the pebbles in the xth plane in ray $R_{i,j}$ by using time step $\ell^2 x + D_{i,j}$. Since each ray contains at most $n/2$ planes of pebbles, $p_{i,j}$ finishes simulating ray $R_{i,j}$ by using step $\ell^2 n/2 + D_{i,j}$. ∎

This implies a slowdown of $O(n^2/m^2 + D_{m,m}/n)$. To minimize the slowdown, FATRAY uses the contiguous subarray S that minimizes $n^2/m_S^2 + D_S/n$, where $m_S \times m_S$ is the size of S and D_S is the length of the longest monotone path in S.

THEOREM 3.4. FATRAY *achieves a slowdown of* $\min_{subarrays\ S} O(n^2/m_S^2 + D_S/n)$.

Unfortunately, the slowdown can still be big compared with d_{ave}. For example, suppose that H is partitioned into n squares of size $\sqrt{n} \times \sqrt{n}$ with one edge of delay n in the center of each square and unit delay on all other edges. The slowdown is $\min_S \Theta(n^2/m_S^2 + D_S/n) = \Theta(n^{1/3})$, whereas d_{ave} is a constant. Matters are better, however, when all the delays are the same, as we show in the following theorem.

THEOREM 3.5. *In the case where all the delays in H are d,* FATRAY *efficiently simulates G on H and achieves a slowdown of $\Theta\left(\min\{d^{2/3}, n^2\}\right)$. The slowdown is optimal up to a constant factor.*

Proof. When $d \le n^3$ FATRAY uses a subarray of size $\frac{n}{d^{1/3}} \times \frac{n}{d^{1/3}}$. Theorem 3.4 implies a slowdown of $O(d^{2/3})$. We show that the slowdown is asymptotically tight as follows. Consider pebble $(i, j, d^{1/3})$, and suppose processor q computes it in a simulation.

Let A be the set of pebbles of the form (i', j', t), for $1 \leq t < d^{1/3}$, on which $(i, j, d^{1/3})$ depends; i.e., $(i, j, d^{1/3})$ cannot be computed until after (i', j', t) is computed. If every pebble in A is computed q, then it takes at least $|A| = \Omega\left((d^{1/3})^3\right) = \Omega(d)$ time steps to simulate A. Otherwise, a processor $p \neq q$ computes some pebble in A and passes this information to q. The delay from p to q is at least d. Hence, the slowdown on simulating the first $d^{1/3}$ steps is $d^{2/3}$. The same argument applies for the slowdown in the next $d^{1/3}$ steps.

When $d > n^3$ FatRay uses a single host processor for the simulation and achieves a slowdown of $O(n^2)$. This slowdown is asymptotically tight for the same reason as in the previous case. We consider pebbles (i, j, n) instead of $(i, j, d^{1/3})$. In both cases the simulation is work efficient. □

Theorem 3.5 can be generalized to any k-dimensional array for $k \geq 1$.

Theorem 3.6. *Suppose G is an $n \times \cdots \times n$ k-dimensional array with unit-delay edges, and H is an $n \times \cdots \times n$ k-dimensional array with delay-d edges; then H can efficiently simulate G with a slowdown of* $\Theta(\min\{d^{k/k+1}, n^k\})$. *The slowdown is optimal up to a constant factor.*

3.2. Improved bounds for worst-case delays. In order to improve the slowdown, we observe that not all the host processors are useful. If a host processor is surrounded by high delays, then the benefit to be gained by using its computing power is nullified by the communication cost. We first describe criteria of removing such host processors. We then embed guest processors to the unremoved host processors. Suppose that guest processor $g_{i,j}$ is mapped to host processor p; then p computes the pebbles in ray $R_{i,j}$ in the 2D-Ray algorithm. For any arrangement of the delays in H, we show how to embed G on H such that, for any monotone path in G, its image in H has length of $O(d_{\text{ave}} n \log^{5/2} n)$. As a result, Lemma 3.2 implies a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$ as long as only $O(1)$ guest processors are mapped to each host processor. By applying the idea used in FatRay, we improve the slowdown to $O(d_{\text{ave}}^{2/3} \log^{5/3} n)$ and achieve work efficiency at the same time.

3.2.1. Removing useless processors. We first recursively represent H using a quadtree, in which each node corresponds to a subarray of H. The root represents the entire $n \times n$ array. The four children of the root represent the four $\frac{n}{2} \times \frac{n}{2}$ subarrays, etc. In general, a node at depth k of the quad tree corresponds to an $\frac{n}{2^k} \times \frac{n}{2^k}$ subarray of H. We refer to this subarray as a *depth-k* array. The leaves represent the individual processors of H. (See Figure 6.)

We describe a two-stage procedure to remove "useless" processors of H. A processor is removed if it is surrounded by high delays (Stage 1) or few unremoved processors (Stage 2). (When a processor is removed, its incident edges remain in the network.) For each depth k, we define two quantities D_k for "delay threshold" and m_k for "survival threshold." Note that D_k is larger than the average delay on a row/column in a depth-k array by a factor of $\Theta(\log n)$, and m_k is smaller than the number of processors in a depth-k array by a factor of $\Theta(\log n)$:

$$D_k = (c \log n)\left(\frac{n}{2^k} d_{\text{ave}}\right), \tag{1}$$

$$m_k = \left(\frac{1}{c \log n}\right)\left(\frac{n^2}{4^k}\right). \tag{2}$$

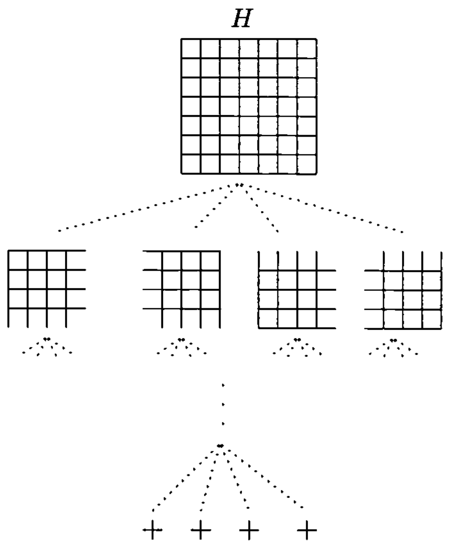


FIG. 6. *The quad tree that represents H.*

A constant c is specified later. We also define a maximum depth $k_{\max}$ such that when $k = k_{\max}$ the survival threshold m_k becomes 1.

$$k_{\max} = \log n - \frac{1}{2}\log c - \frac{1}{2}\log\log n. \tag{3}$$

- **Stage 1.** From depth $k = k_{\max}$ down to depth 0, if the total delay on a row/column of a depth-k array exceeds the threshold D_k, then all the $\frac{n}{2^k}$ processors on that row/column are removed.
- **Stage 2.** From depth $k = k_{\max}$ down to depth 0, if the number of unremoved processors in a depth-k array is smaller than the threshold m_k, then all the processors in that array are removed. Moreover, we also remove processors so that the number of remaining processors in any depth-k array is an integer multiple of m_k.

LEMMA 3.7. *At most $2n^2/c$ processors are removed in Stage* 1.

Proof. The total delay of H is $2n^2 d_{\text{ave}}$. At most $\frac{2n2^k}{c\log n}$ depth-k rows and columns can have delay more than D_k. Since each depth-k row/column contains $\frac{n}{2^k}$ processors, at most $\frac{2n^2}{c\log n}$ processors are removed at depth k. There are $\log n$ depths, and so the lemma follows. □

LEMMA 3.8. *At most n^2/c processors are removed at Stage* 2.

Proof. Since there are 4^k depth-k arrays, at most $\frac{n^2}{c\log n}$ processors are removed at depth k. □

We label each array with the number of unremoved processors contained in it. By Lemmas 3.7 and 3.8, at most $3n^2/c$ processors of H are removed. Therefore, H is labeled with $c_1 n^2$, where $c_1 \geq 1 - (3/c)$. Any constant $c > 3$ works for our argument.

3.2.2. The embedding. For clarity of presentation, we create an intermediate two-dimensional array $\mathcal{G}$ that has size $\sqrt{c_1}n \times \sqrt{c_1}n$ and unit-delay edges only. We describe an algorithm EMBED that maps the processors of $\mathcal{G}$ one to one to the

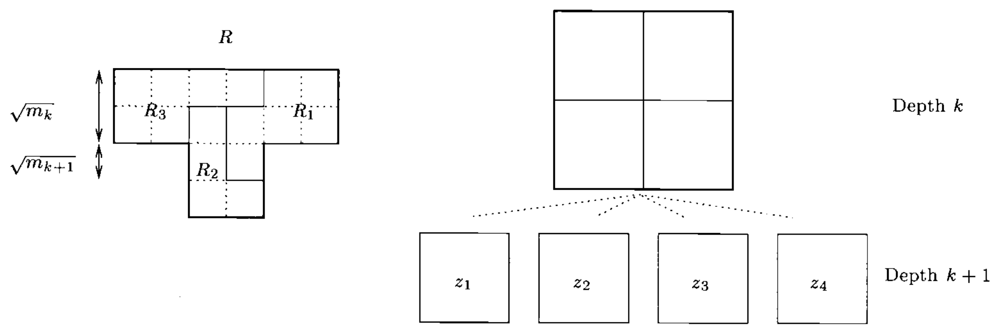


FIG. 7. *(Left) Depth-k region R and depth $k+1$ regions R_1, R_2, and R_3 of $\mathcal{G}$. (Right) Depth-k and $k+1$ arrays of H. Depth $k+1$ region R_i has size z_i, where z_i is the number of unremoved processors in the corresponding array of H. In this figure, $z_4 = 0$, and R_4 is therefore empty.*

unremoved processors of H. The goal is to show that for any monotone path in $\mathcal{G}$ its image in H under EMBED has length $O(d_{\text{ave}} n \log^{5/2} n)$. As a result, H can simulate $\mathcal{G}$ with a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$. Obviously $\mathcal{G}$ can simulate G with constant slowdown.

EMBED partitions $\mathcal{G}$ into *regions* recursively, and each depth-k region of $\mathcal{G}$ corresponds to a depth-k array of H. The depth-0 region is the entire network $\mathcal{G}$. By the construction of Stage 2, $c_1 n^2$ (the number of processors in $\mathcal{G}$) is a multiple of m_0. Hence, $\mathcal{G}$ can be viewed as a collection of contiguous squares of size $\sqrt{m_0} \times \sqrt{m_0}$. We inductively assume that each depth-k region consists of contiguous squares of size $\sqrt{m_k} \times \sqrt{m_k}$, where m_k is defined in equation (2). Each depth-k region R is partitioned into four depth $k+1$ regions R_1, R_2, R_3, and R_4 as follows. First, each $\sqrt{m_k} \times \sqrt{m_k}$ square of R is divided into four squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$, where $\sqrt{m_{k+1}} = \sqrt{m_k}/2$. Suppose that R_i corresponds to a depth $k+1$ square of H that has z_i unremoved processors; then R_i has size z_i. By the construction of Stage 2, z_i is a multiple of m_{k+1}. Hence, R_i can be formed as a collection of contiguous squares of size $\sqrt{m_{k+1}} \times \sqrt{m_{k+1}}$. Note that if z_i is 0, then the corresponding R_i is empty. (See Figure 7.)

At depth $k_{\max}$, each depth-$k_{\max}$ region consists of contiguous squares of size 1×1. EMBED maps the processors in a depth-$k_{\max}$ region of $\mathcal{G}$ to the unremoved processors in the corresponding depth-$k_{\max}$ array of H in an arbitrary one-to-one manner. Thus, we have a one-to-one mapping from the processors of $\mathcal{G}$ to the unremoved processors of H.

We also define the *depth-k boundaries* in $\mathcal{G}$ to be the borders of depth-k regions of $\mathcal{G}$. Note that the depth-k boundaries are at least $\sqrt{m_k}$ apart in both horizontal and vertical directions.

3.2.3. Bounding monotone path length. In this section we bound the total delay on the image of P in H, where P is any monotone path in $\mathcal{G}$. Suppose a and b are two neighboring processors in $\mathcal{G}$; then their images a_H and b_H in H are connected by a 1-*bend* route as follows. First, a_H is routed along its row to b_H's column and then routed to b_H along the column. We define a and b (resp., a_H and b_H) to be *k-related* if k is the largest integer such that a and b (resp., a_H and b_H) are in a same depth-k region (resp., depth-k array). We also define a_H and b_H to be *peers* of each other. Note that each unremoved host processor can have four peers.

LEMMA 3.9. *Let P be any monotone path in $\mathcal{G}$. The image of P in H has a total delay of $O(d_{ave} n \log^{5/2} n)$ under* EMBED.

Proof. Let a and b be two neighboring processors on P and let a_H and b_H be their images. Suppose a and b (resp., a_H and b_H) are k-related. We first bound the length of the 1-bend route from a_H to b_H. By the construction of Stage 1, the total delay on a depth-k row/column that contains a_H or b_H is at most D_k. Hence, the distance from a_H to b_H is at most $2D_k$.

We now bound the number of neighboring a's and b's that can be k-related. If $k < k_{\max}$, P must cross some depth $k+1$ boundary of $\mathcal{G}$ in traveling from a to b. Since P is monotone and the depth-k boundaries are $\sqrt{m_k}$ apart in both horizontal and vertical directions, P can cross the depth-k boundaries at most $\frac{2n}{\sqrt{m_k}}$ times. Hence, at most $\frac{2n}{\sqrt{m_{k+1}}}$ neighboring a's and b's on P can be k-related. This implies that the total delay incurred by k-related peers on the image of P is at most $2D_k \frac{2n}{\sqrt{m_{k+1}}}$ for $k < k_{\max}$. Obviously, at most $2n$ neighboring a's and b's can be $k_{\max}$-related. Summing over all depths, we conclude that the total delay on the image of P is at most $2D_{k_{\max}} \cdot 2n + \sum_{k<k_{\max}} 2D_k \frac{2n}{\sqrt{m_{k+1}}}$, which is $O(d_{\text{ave}} n \log^{5/2} n)$ by the definitions of D_k, m_k and $k_{\max}$. ◻

Hence, we can embed G on H such that $O(1)$ guest processors are mapped to each host processor and that the image in H of any monotone path in G has length $O(d_{\text{ave}} n \log^{5/2} n)$. Lemma 3.2 implies that H can simulate G with a slowdown of $O(d_{\text{ave}} \log^{5/2} n)$. To improve the slowdown and achieve efficiency, we apply the idea of complementary slackness and use an $m \times m$ contiguous subarray of H for simulation as in FATRAY. Theorem 3.4 and Lemma 3.9 imply a slowdown of $O((d_{\text{ave}} m \log^{5/2} m)/n + n^2/m^2)$. By choosing m to be $\max\{n d_{\text{ave}}^{-1/3} \log^{-5/6} n, 1\}$, we have Theorem 3.10.

THEOREM 3.10. *Network H with average delay d_{ave} can efficiently simulate G with a slowdown of $O(d_{ave}^{2/3} \log^{5/3} n)$.*

3.2.4. Bandwidth. The preceding analysis focuses entirely on the issue of latency and ignores bandwidth constraints. This does not present any problems if the link bandwidth available on the host array is $\Omega(\log^{3/2} n)$ times larger than that on the guest array. If the bandwidth of the host and guest arrays are comparable, however, and if the guest array is fully utilizing the bandwidth on its links, then *congestion* becomes an issue. Formally, the congestion on an edge equals the number of pebbles that wish to cross this edge simultaneously. In this case, we may need to slow down the simulation by an additional factor of $O(\log^{3/2} n)$.

In section 3.2.3, peers a_H and b_H are connected by a 1-bend route in H. To address the congestion issue, we present a more sophisticated method of connecting a_H and b_H such that each edge in H has $O(\log^{3/2} n)$ routes going through it and that the distance between a_H and b_H remains unchanged asymptotically.

We begin with some definitions. Recall that EMBED maps each depth-k region R_k of $\mathcal{G}$ to a depth-k array S_k of H. A depth-k row/column of S_k is *live* if it contains some unremoved host processors. A boundary point of S_k is live if it belongs to some live row or column of S_k. We first bound the number of connections from inside of S_k to outside of S_k in terms of the number of live rows and columns of S_k.

LEMMA 3.11. *Consider any depth-k array, S_k, of H. The number of processors in S_k that have peers outside S_k is $O(x\sqrt{\log n})$, where x is the number of live rows and columns in S_k.*

Proof. Let z be the number of unremoved processors in S_k; then the number of live rows and columns is at least $\frac{z}{n/2^k}$. The number of host processors in S_k that have peers outside S_k is proportional to the perimeter of R_k, the depth-k region that corresponds to S_k. By the construction of EMBED, R_k consists of squares of size $\sqrt{m_k} \times \sqrt{m_k}$. Hence, R_k has perimeter of $O(z/\sqrt{m_k})$, which is $O(\frac{z}{n/2^k}\sqrt{\log n})$ by the definition of m_k in (2). Our lemma follows. $\square$

We now describe a recursive procedure that connects the peers. The following facts are used in our routing.

FACT 3.12. *Consider a routing problem on a square array of size $x \times x$.*

1. *If each node has $O(y)$ requests, then the routing can be done in one bend and $O(xy)$ congestion.*
2. *Let the nodes on the* cross *divide the square array into four $\frac{x}{2} \times \frac{x}{2}$ quadrants. If each boundary node and cross node have $O(y)$ requests and all other nodes have no requests, then the routing can be done in $O(1)$ bends and $O(y)$ congestion.*

Our recursive routing starts at depth $k = k_{\max}$. Consider all the depth-k arrays S_k. For all the peers that are k-related, we connect them through a 1-bend routing within S_k. Since S_k has size $\sqrt{\log n} \times \sqrt{\log n}$ and each host processor has at most 4 peers, the congestion caused by this 1-bend routing within S_k is $O(\sqrt{\log n})$ by item 1 of Fact 3.12. For all the processors that have peers outside S_k, we route them to live boundary points such that the following two conditions hold. First, each live boundary point of S_k receives $O(\sqrt{\log n})$ requests. This is possible because of Lemma 3.11. Second, the routing uses 1 bend and causes a congestion of $O(\log n)$ by item 1 of Fact 3.12.

We proceed recursively to depths $k < k_{\max}$. Consider all the depth-k arrays S_k. From the previous stage the host processors that are not connected to their peers are routed to some live boundary points of depth $k+1$ arrays. Hence, they are either on the boundary or on the cross of S_k, and $O(\sqrt{\log n})$ host processors are routed to the same location. For all the peers that are k-related, we connect them within S_k. Otherwise, we route them to the live boundary points of S_k such that each live point receives $O(\sqrt{\log n})$ requests (including those from all previous stages but have not yet connected to their peers). This is possible by Lemma 3.11. In both cases, item 2 of Fact 3.12 implies that the routing can be done in $O(1)$ bends and that the congestion incurred is $O(\sqrt{\log n})$.

The congestion incurred at depth k, for $1 \le k < k_{\max}$, is $O(\sqrt{\log n})$ and at depth $k_{\max}$ is $O(\log n)$. Since each of the depths uses the same underlying edges, the overall congestion is $O(\log^{3/2} n)$. The host processors are routed to live boundary points in $O(1)$ bends at each depth, and therefore the length incurred at depth k is $O\left(\frac{n}{2^k} d_{\text{ave}} \log n\right)$. Suppose that a_H and b_H are k-related; then the distance between them is $\sum_{k' \ge k} O(\frac{n}{2^{k'}} d_{\text{ave}} \log n)$, which remains $O(\frac{n}{2^k} d_{\text{ave}} \log n)$ as in Lemma 3.9. In summary, we have Lemma 3.13.

LEMMA 3.13. *In the above routing scheme the congestion is $O(\log^{3/2} n)$ on all edges of H. Furthermore, for any monotone path P in $\mathcal{G}$, the image of P in H has length $O(d_{ave} n \log^{5/2} n)$.*

3.3. Improved bounds for randomly arranged delays. In this section, we show that the length of the longest monotone path in H is often short when the delays are randomly arranged. If M is the number of edges in an $n \times n$ array H, then for a given set of M delays with average d_{ave} the longest monotone path in H has length $O(nd_{\text{ave}})$ for most of the $M!$ permutations of the delays. That is, in the uniform distribution of the $M!$ permutations, the longest monotone path has length $O(nd_{\text{ave}})$ with high probability, and therefore the slowdown is $O(d_{\text{ave}})$ with high probability.

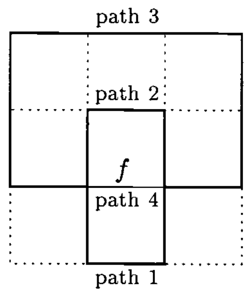


FIG. 8. *The bounding box and four edge-disjoint alternate paths for edge f.*

Without loss of generality we assume that d_{ave} is a constant. (For a nonconstant d_{ave}, each delay d is normalized to $\max\{d/d_{\text{ave}}, 1\}$. The normalized delays have average $O(1)$, and the total original delay on any monotone path is at most d_{ave} times the total normalized delay.) We divide the delays in H into $O(\log n)$ levels. Level ℓ contains the delays that are in the range of $[2^\ell, 2^{\ell+1})$, and level ℓ^+ contains the delays that are at least 2^ℓ.

3.3.1. Shortcuts and edge coloring. If an edge with a large delay is surrounded by edges with small delays, we can route around this large delay. The intuition is that in a random permutation most long delays can be shortcut. For each edge f, we consider four edge-disjoint *alternate* paths that connect the two endpoints of f. (See Figure 8.) The 3×3 box that contains these four paths is called the *bounding box* of f. After the *shortcut*, the total delay on f equals the shortest alternate path length. For clarity, we shall refer to the delay before the shortcut as the *original delay* and the delay after the shortcut as the *shortcut delay.*

For a given set of delays with a constant average, the number of level-ℓ^+ original delays is $O(n^2 2^{-\ell})$. Therefore, the probability for an edge f to have a level-ℓ^+ original delay is $O(2^{-\ell})$. However, shortcutting dramatically decreases this probability as the following lemma shows.

LEMMA 3.14. *The probability for an edge f to have a level-ℓ^+ shortcut delay is $O(2^{-4\ell})$.*

Proof. If edge f has a shortcut delay from level ℓ^+, then the four alternate paths must each have an edge whose original delay is from level $(\ell-4)^+$. For a particular set of four edges to have level $(\ell-4)^+$ original delays, the probability is $\binom{B}{4}/\binom{M}{4}$, where B is the number of level $(\ell-4)^+$ original delays, and M is the number of edges in H. Since there are $3 \cdot 3 \cdot 9 \cdot 1 = 81$ ways to choose four edges from four alternate paths, we derive the following from a union bound:

$$\Pr[\text{ Shortcut delay on } f \text{ is from level } \ell^+ \;] \le 81 \cdot \binom{B}{4} \bigg/ \binom{M}{4}.$$

Our lemma follows from the observation that $B = O(n^2 2^{-\ell})$ since $d_{\text{ave}} = O(1)$, and that $M = \Theta(n^2)$. □

Unfortunately, these probabilities are *not* independent from edge to edge for two reasons. First, the arrangement of delays is a permutation of a given set of delays. This does not cause a problem, however, as the analysis in Lemmas 3.15 and 3.17 will show. Intuitively, in a permutation if one edge has a large delay, then other edges are less likely to have large delays. Second, the bounding boxes are not necessarily disjoint. To resolve this problem we introduce an *edge coloring*, so that any two distinct edges with the same color have edge-disjoint bounding boxes. Clearly, only a constant number of colors are needed.

We show in the following that, for *any* monotone path in H, the total delay incurred from the edges in one particular color group is $O(n)$ with high probability. Since there are $O(1)$ color groups, our results follows from a union bound. For each color group we consider two cases, edges with large shortcut delays and edges with small shortcut delays.

3.3.2. Large delays. In this section we show that, with high probability, the total delay in H due to shortcut delays from large levels is $O(n)$. Therefore, any monotone path can only pick up $O(n)$ delays from these levels.

LEMMA 3.15. *With probability* $1 - O(n^{-1})$, any *monotone paths pick up a total delay of* $O(n)$ *from levels* $\ell \geq L$, *where* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

Proof. By Lemma 3.14, the probability that one particular edge has a shortcut delay from level $(\frac{3}{4}\log n)^+$ is $O(n^{-3})$. Since H has $\Theta(n^2)$ edges, with probability $1 - O(n^{-1})$ no edge in H has shortcut delay from level $(\frac{3}{4}\log n)^+$.

We show below that, with high probability, H has $O(\log^3 n)$ shortcut delays from level L^+. Let $A = an^2$ be an upper bound on the number of edges in one particular color group, where a is a constant. Since $d_{\text{ave}} = O(1)$, at most $B = bn^{3/2}\log^{1/2} n$ original delays can be from levels $(L-4)^+$, where b is a constant. We show that, with a small probability, more than $C = c\log^3 n$ edge delays are from level L^+ for a sufficiently large constant c.

For a particular set of C edges to have level-L^+ shortcut delays, at least four edges in each of these C bounding boxes have level $(L-4)^+$ original delays. For a particular set of four edges in each bounding box to have level $(L-4)^+$ original delays, the probability is at most $\binom{B}{4C}/\binom{M}{4C}$. This is true since all the C bounding boxes are edge disjoint. There are at most $\binom{A}{C}$ ways to choose C edges whose shortcut delays are from L^+ and 81^C ways to choose four edges from each of the C boxes. We therefore derive the following from a union bound:

$$\begin{aligned} p &= \Pr[\text{ At least } C \text{ edges have level-}L^+ \text{ shortcut delays }] \\ &\leq 81^C \binom{A}{C}\binom{B}{4C} \Big/ \binom{M}{4C}. \end{aligned} \tag{4}$$

We bound probability p with the inequalities

$$\left(\frac{y}{x}\right)^x \leq \binom{y}{x} \leq \left(\frac{ye}{x}\right)^x, \tag{5}$$

where $e = 2.718$ is the base of the natural logarithm. By the definitions of A, B, and C and the fact that $M \approx 2n^2$, we have

$$p \leq \left(\frac{81 \cdot Ae}{C}\right)^C \left(\frac{Be}{M}\right)^{4C} = \left(\frac{81 \cdot a \cdot b^4 \cdot e^5}{2^4 \cdot c \cdot \log n}\right)^{c\log^3 n}.$$

Let c be a sufficiently large constant; then the above probability is bounded by $O(n^{-1})$. Summing over all the $O(1)$ color groups, we conclude that with probability $1-O(n^{-1})$ H has no shortcut delays from level $(\frac{3}{4}\log n)^+$ and $O(\log^3 n)$ shortcut delays from level L^+. Hence, any monotone path picks up a total delay of $O(n^{3/4}\log^3 n) = O(n)$ from levels $\ell \geq L$. $\square$

3.3.3. Small delays. In this section we show that the shortcut delay from small levels do not accumulate too much on any monotone path with high probability. In particular, with probability $1 - O(n^{-2})$, each monotone path picks up an $O(n2^{-\ell})$ delay from each "small" level ℓ. Summing over all $O(\log n)$ "small" levels, we can conclude that each monotone path picks up a total of $O(n)$ small delays with probability $1 - O(n^{-1})$.

Consider a particular level $\ell < L$, where $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$. We divide H into m^2 squares of size $2^{2\ell} \times 2^{2\ell}$, where $m = n2^{-2\ell}$. There are $\binom{2m-1}{m}$ sequences of $2m-1$ squares that some monotone path could possibly go through. We call these sequences of $2m-1$ squares *monotone sequences.* If the total number of level-ℓ shortcut delays in each of these sequences is bounded, then the total level-ℓ shortcut delay that any monotone path picks up is also bounded.

LEMMA 3.16. *With probability* $1 - O(n^{-2})$, *any monotone path picks up a total of a* $O(n2^{-\ell})$ *delay from level-*ℓ *shortcut delays, where* $\ell < L$ *is one particular level and* $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.

Proof. Consider one particular monotone sequence of $2m-1$ squares of size $2^{2\ell} \times 2^{2\ell}$, where $m = n2^{-2\ell}$. Let random variable X be the number of level-ℓ shortcut delays in this sequence of squares, and let random variable X_i be the number of level-ℓ shortcut delays from the ith square in the sequence. We use a moment generating function argument to upper bound $X = X_1 + \cdots + X_{2m-1}$. We first bound the probability $\Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,]$. Let $A = a2^{4\ell}$ be an upper bound on the number of edges from one particular color group in each $2^{2\ell} \times 2^{2\ell}$ square, where a is a constant. Since $d_{\text{ave}} = O(1)$, at most $B = bn^2 2^{-\ell}$ original delays can be from level $(\ell-4)^+$, where b is a constant. Let $k = \sum_{i=1}^{2m-1} k_i$. By applying the same logic as for inequality (4), we have

$$\begin{aligned} P &= \Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,] \\ &\le 81^k \cdot \binom{A}{k_1} \cdots \binom{A}{k_{2m-1}} \cdot \binom{B}{4k} \Big/ \binom{M}{4k}. \end{aligned}$$

By inequality (5), the probability is bounded by

$$\begin{aligned} P &\le 81^k \cdot \left(\frac{Ae}{k_1}\right)^{k_1} \cdots \left(\frac{Ae}{k_{2m-1}}\right)^{k_{2m-1}} \cdot \left(\frac{Be}{M}\right)^{4k} \\ &= \prod_{i=1}^{2m-1} \left(\frac{81 \cdot a \cdot b^4 \cdot e^5}{2^4 \cdot k_i}\right)^{k_i}. \end{aligned} \tag{6}$$

We proceed to bound the expectation of e^X:

$$\begin{aligned} E[e^X] &= E[e^{X_1 + \cdots + X_m}] \\ &= \sum_{k \ge 0} e^k \sum_{\sum k_i = k} \Pr[\, X_1 = k_1, \ldots, X_{2m-1} = k_{2m-1} \,] \\ &\le \sum_{k \ge 0} \sum_{\sum k_i = k} \prod_{i=1}^{2m-1} \left(\frac{y}{k_i}\right)^{k_i}, \qquad \text{where } y = 81 \cdot a \cdot b^4 \cdot e^6 \cdot 2^{-4} \\ &\le \sum_{k \ge 0} \sum_{\sum k_i = k} \prod_{i=1}^{2m-1} \frac{y^{k_i}}{k_i!} \\ &= e^{y(2m-1)}. \end{aligned}$$

The first inequality follows from inequality (6), and the last equality follows from $e^{y(2m-1)} = (\sum_{j\geq 0} \frac{y^j}{j!})^{2m-1}$. We use Markov's inequality to bound the probability that the total number of level-ℓ shortcut delays exceeds $\beta(2m-1)$ in this particular monotone sequence:

$$\Pr\left[\, X \geq \beta(2m-1)\right] = \Pr\left[\, e^X \geq e^{\beta(2m-1)}\right] \leq \frac{E[e^X]}{e^{\beta(2m-1)}} \leq e^{(y-\beta)(2m-1)}.$$

There are $\binom{2m-1}{m} < 2^{2m-1}$ monotone sequences. By a union bound, the probability that every sequence has fewer than $\beta(2m-1)$ level-ℓ shortcut delays is at least $1 - 2^{2m-1}e^{(y-\beta)(2m-1)}$. Let β be the constant $y+2$; then this probability is bounded by $1 - O(n^{-2})$, since $m = n/2^{2\ell} \geq \log n$. Therefore, every monotone path picks up a total of $O(n2^{-\ell})$ shortcut delays from level ℓ with probability $1 - O(n^{-2})$. $\square$

Summing over all levels $\ell < L$ results in a total delay that is linear in n as desired.

LEMMA 3.17. *With probability $1-O(n^{-1})$, all the monotone paths pick up a total delay of $O(n)$ from levels $\ell < L$ for $L = \frac{1}{2}\log n - \frac{1}{2}\log\log n$.*

For the case in which $d_{\text{ave}} = O(1)$, Lemmas 3.15 and 3.17 show that any monotone path has a total delay of $O(n)$ with high probability. For the case in which d_{ave} is nonconstant, the discussion at the beginning of the section implies that Theorem 3.18 holds.

THEOREM 3.18. *Suppose H is a network with average delay d_{ave}; then with high probability every monotone path in H has delay $O(nd_{ave})$.*

To make the algorithm work efficient, we use an $m \times m$ subarray of H of average delay at most d_{ave} to simulate G. Theorems 3.4 and 3.18 imply that the slowdown is $O(n^2/m^2 + d_{\text{ave}}m/n)$ with high probability. By choosing m to be $\max\{nd_{\text{ave}}^{-1/3}, 1\}$, we have Theorem 3.19.

THEOREM 3.19. *Suppose the delays on network H are from a random permutation of a set of delays whose average is d_{ave}; then with high probability H can simulate G with slowdown $O(d_{ave}^{2/3})$.*

Congestion problems are not an issue here, since each edge of H is used $O(1)$ times by alternate paths in the shortcut process.

4. Database model. We switch our attention to the database model. As discussed in section 1, simulation in the database model is more difficult than in the dataflow model. For algorithms such as STRIPE in section 2 to work for the database model a host processor needs $\Theta(n)$ copies of the databases on average. This is unrealistic because of the memory requirement as well as the difficulty in updating the databases. We therefore develop new machinery for the database model. Contrary to the dataflow model, we make substantial use of redundant computation. Apart from the slowdown, another important parameter for the database model is *load*, which is the number of databases that a host processor copies.

The main contribution of this section is an algorithm called OVERLAP that simulates linear arrays in the database model with a small load and a small slowdown. Since OVERLAP is technically involved, we begin with a special case in section 4.1, where the host linear array has delay d on all edges. The simulation in this special case is much simpler, and it conveys some intuition for using redundant computation in the general case. Section 4.2 presents OVERLAP in detail. The techniques are generalized to simulate linear and two-dimensional arrays on general networks in sections 4.3 and 4.4. Last, in section 4.5 we discuss the lower bounds on slowdown when each database is allowed a small number of copies.

4.1. A special case. In this section we consider a special case. Let G be a guest linear array with n processors and unit-delay edges, and let H be a host linear array with n processors and delay d on all edges. We use redundant computation to achieve an optimal slowdown of $O(\sqrt{d})$. Recall that in the dataflow model, the optimal slowdown is achieved for linear arrays without using redundancy.

THEOREM 4.1. *In the database model, H can efficiently simulate G with a slowdown and a load of $O(\sqrt{d})$. This slowdown is optimal up to a constant factor.*

Proof. We consider two cases. If $n \le \sqrt{d}$, then one host processor copies all the databases and carries out the entire computation by itself. Hence the load and the slowdown are n, which is $O(\sqrt{d})$. Otherwise, the first $\frac{n}{\sqrt{d}}$ host processors are used for the simulation. For $1 \le j \le \frac{n}{\sqrt{d}}$, processor p_j copies $3\sqrt{d}$ databases b_i and computes $3\sqrt{d}$ columns of pebbles (i,t), where $(j-2)\sqrt{d}+1 \le i \le (j+1)\sqrt{d}$ and $1 \le t$. In this way each processor shares $\sqrt{d}$ databases with its right and left neighbors and each pebble is redundantly computed by three neighboring processors.

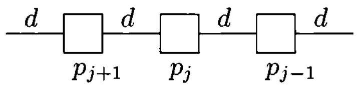
We show how to simulate the first $\sqrt{d}$ rows of pebbles created by G in $O(d)$ steps by H. Every subsequent $\sqrt{d}$ rows of pebbles are simulated in the same manner. The algorithm is demonstrated in Figure 9. For $1 \le j \le \frac{n}{\sqrt{d}}$, let

$$\begin{array}{rcll}
P_j &=& \{\text{Pebbles } (i,t): & 1 \le t \le \sqrt{d}, \quad -2\sqrt{d}+1 \le i - j\sqrt{d} \le \sqrt{d}\},\\
L_j &=& \{\text{Pebbles } (i,t): & 1 \le t \le \sqrt{d}, \quad 1 \le i-(j-2)\sqrt{d} \le t\},\\
R_j &=& \{\text{Pebbles } (i,t): & 1 \le t \le \sqrt{d}, \quad -t+1 \le i-(j+1)\sqrt{d} \le 0\},\\
T_j &=& P_j - (L_j \cup R_j), & \\
A_j &=& \{\text{Pebbles } ((j-2)\sqrt{d},t): & 1 \le t \le \sqrt{d}\},\\
B_j &=& \{\text{Pebbles } ((j-1)\sqrt{d}+1,t): & 1 \le t \le \sqrt{d}\},\\
C_j &=& \{\text{Pebbles } ((j\sqrt{d},t): & 1 \le t \le \sqrt{d}\},\\
D_j &=& \{\text{Pebbles } ((j+1)\sqrt{d}+1,t): & 1 \le t \le \sqrt{d}\}.
\end{array}$$

Processor p_j of H computes all the pebbles in P_j. First, p_j computes the pebbles in the trapezium T_j without communicating with its neighbors. There are $2d$ pebbles in T_j, and so this takes $2d$ steps. Next, p_j passes column B_j to processor p_{j-1} and receives column A_j from p_{j-1}. It also passes column C_j to processor p_{j+1} and receives column D_j from p_{j+1}. This communication takes $d+\sqrt{d} < 2d$ steps using pipelining. Processor p_j can now compute the pebbles in triangles L_j and R_j in d steps. It is important for p_j to compute the pebbles in L_j and R_j in order to continue the simulation of the next $\sqrt{d}$ rows of pebbles, since databases need to be updated. This presents a major difference between the dataflow and database models.

Hence, it takes at most $5d$ steps in total for processor p_j to compute every pebble in P_j. The next $\sqrt{d}$ steps of computation can be simulated in a similar fashion. The slowdown is therefore $O(\sqrt{d})$. The lower bound proof in Theorem 3.5 implies that the slowdown of $\Omega(\sqrt{d})$ is necessary.

Note that during the computation of T_j, the pebbles in columns B_j and C_j can start to travel to the neighboring processors of p_j as soon as they are ready. Processor p_j can also start to compute triangles L_j and R_j before the entire columns of A_j and D_j are transferred. In this way, the communication time can be saved. Although it does not make a difference asymptotically in this case we take advantage of this observation in OVERLAP. $\square$



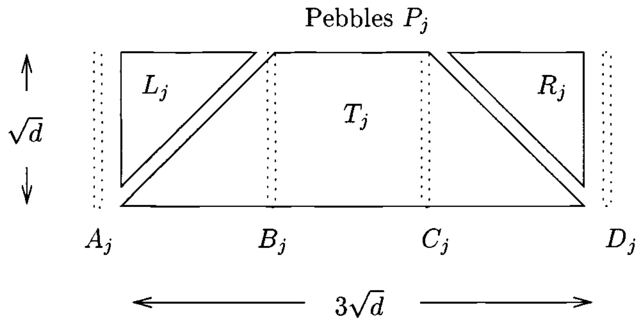


FIG. 9. *Simulating $\sqrt{d}$ steps of computation of G on H.*

4.2. Algorithm OVERLAP. To simulate a guest linear array on a host linear array with arbitrary delays we use an algorithm called OVERLAP. In OVERLAP, we remove host processors that are surrounded by high delays. The motivation of this step is similar to that of section 3.2. For the remaining processors, we decide how much redundancy is needed for neighboring processors and how much computation each processor is able to carry out. During the simulation, some pebbles are redundantly computed to ensure that the communication is not too costly. We first obtain a slowdown of $O(d_{\text{ave}} \log^3 n)$, where d_{ave} is the average delay of H and n is the size of G and H, and later improve the slowdown to $O(\sqrt{d_{\text{ave}}} \log^3 n)$ while achieving work efficiency.

4.2.1. Removing useless processors. We recursively represent H using a binary tree, in which each node corresponds to a subarray of H. The root represents the entire array. The left and right children of the root represent the left and right halves of the array, respectively. In general, a node at depth k of the binary tree corresponds to a subarray of H that contains $\frac{n}{2^k}$ processors. We refer to this subarray as a *depth-k interval*. The leaves represent the individual processors of H. (See Figure 10.)

We describe a two-stage process that removes the processors that are surrounded by high delays (Stage 1) and the processors that are surrounded by few unremoved processors (Stage 2). During Stage 2, we also label each *live* subarray, where a live subarray contains some unremoved processor. These labels indicate how many columns of pebbles the live subarrays are able to compute.

For every depth k, we define D_k to be the "delay threshold" and m_k to be the "overlap size" as follows. Note that D_k is larger than the average delay in a depth-k interval by a factor of $\Theta(\log n)$, and m_k is smaller than the number of processors in a depth-k interval by a factor of $\Theta(\log n)$. We shall use m_k to indicate the size of overlap between neighboring depth-k intervals, i.e., the number of columns of pebbles redundantly computed by both intervals:

$$D_k = (c \log n) \left(\frac{n}{2^k} d_{\text{ave}} \right), \tag{7}$$

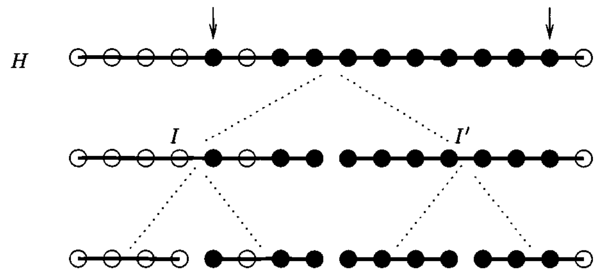


FIG. 10. *The binary tree that represents H. In this figure, unremoved processors of H are represented by black circles; removed processors are represented by white circles. Arrows indicate the endpoints of the root interval. Interval I has one live child and I' has two live children.*

$$m_k = \left(\frac{1}{c\log n}\right)\left(\frac{n}{2^k}\right). \tag{8}$$

As we shall see, any constant $c > 5/2$ works for our argument. We also define a maximum depth $k_{\max}$ such that when $k = k_{\max}$ the overlap size m_k becomes 1:

$$k_{\max} = \log n - \log\log n - \log c. \tag{9}$$

- **Stage 1.** From depth $k = k_{\max}$ down to depth 0, if the total delay in a depth-k interval exceeds D_k, then all the processors in that interval are removed.
- **Stage 2.** At depth $k = k_{\max}$, let I be a live depth-k interval and let x be the number of unremoved processors in I. If x is smaller than $2m_k$, then we remove all the remaining processors in I and I is no longer live. Otherwise, we label I with x.
  Suppose all the live depth-$(k+1)$ intervals are labeled. Now consider each live depth-k interval I. If I has two live children I_1 and I_2 that are labeled with x_1 and x_2, then let $x = x_1 + x_2 - m_{k+1}$. If I has one live child I_1 that is labeled with x_1, then let $x = x_1$. If $x < 2m_k$, we remove all the remaining processors in I and I is no longer live. Otherwise, we label I with x. We proceed to depth $k-1$ until reaching depth 0.

LEMMA 4.2. *At most n/c processors are removed at Stage* 1.

Proof. The total delay in the array H is nd_{ave}. At most $\frac{2^k}{c\log n}$ depth-k intervals can have delay more than D_k. Each depth-k interval contains $\frac{n}{2^k}$ processors and so at most $\frac{n}{c\log n}$ processors are removed at depth k. Since there are $\log n$ depths, at most n/c processors are removed at Stage 1. ☐

LEMMA 4.3. *The label on the root interval is at least $\left(1 - \frac{5}{2c}\right) n$ at Stage* 2.

Proof. Before Stage 2, the number of remaining processors in H is at least $(1 - 1/c)n$ by Lemma 4.2. At depth $k = k_{\max}$ of Stage 2, the sum of the labels on the live depth-k intervals is at least $(1-1/c)n - 2m_k2^k$, which is $(1-1/c)n - \frac{2n}{c\log n}$. At each depth $k < k_{\max}$, the sum of the labels on the live depth-k intervals decreases by at most $(2m_k + m_{k+1})2^k$, which is $\frac{5n}{2c\log n}$. Summing over all depths, we conclude that the label at the root interval is at least $\left(1 - \frac{5}{2c}\right) n$. ☐

4.2.2. Assigning databases. For clarity of presentation, we first assume that G has n' processors, where n' is the label on the root interval of G and n' is a constant fraction of n by Lemma 4.3. We also assume the existence of pebbles $(0, t)$ and $(n'+1, t)$, for all $t \geq 1$, which are known to H at time step 0. This ensures that each pebble computed by G is dependent on three pebbles.

Algorithm Overlap assigns one database to each remaining processor of H so that H has load one. In particular, a depth-k interval with label x is assigned x databases. The depth-0 interval, i.e., H, has all the databases $b_1, \ldots, b_{n'}$. We assume inductively that a depth-k interval I labeled x is assigned databases $b_{i+1}, \ldots, b_{i+x}$. If I has only one child I_1, then Overlap assigns $b_{i+1}, \ldots, b_{i+x}$ to I_1. If I has two children I_1 and I_2 that are labeled x_1 and x_2, respectively, then $x = x_1 + x_2 - m_{k+1}$ by the construction of Stage 2. Overlap assigns $b_{i+1}, \ldots, b_{i+x_1}$ to interval I_1 and $b_{i+x-x_2+1}, \ldots, b_{i+x}$ to I_2. Note that m_{k+1} databases, namely, $b_{i+x-x_2+1}, \ldots, b_{i+x_1}$, are assigned to both I_1 and I_2. These m_{k+1} columns of pebbles will be redundantly computed by both I_1 and I_2. At depth $k_{\max}$ each remaining processor of H is assigned one database.

4.2.3. The simulation. In Overlap, H recursively simulates every $m_0 = \frac{n}{c \log n}$ rows of pebbles created by G as follows. If H (the depth-0 interval) has two live depth-1 intervals I_1 and I_2 as children, then I_1 and I_2 recursively compute the first $m_1 = m_0/2$ rows of pebbles and then repeat for the next m_1 rows. In particular, I_1 (resp., I_2) computes all the pebbles of the form (i, t), where I_1 (resp., I_2) owns database b_i and $1 \le t \le m_1$. Intervals I_1 and I_2 share m_1 databases and therefore redundantly compute these m_1 columns of pebbles. If H has one live child I_1, then I_1 recursively computes the first m_1 rows and then repeats for the second m_1 rows. At depth $k = k_{\max}$, each depth-k interval computes $m_k = 1$ row of pebbles. Theorem 4.4 explains the simulation in detail.

Let us define a set of values $s_t^{(k)}$ for $0 \le k \le k_{\max}$ and $1 \le t \le m_k$, where the superscript k represents the depth of the recursion, and the subscript t represents the row number. Roughly speaking, $s_t^{(k)}$ corresponds to the time by which a depth-k interval computes its pebbles in the tth row. We are interested in the slowdown $s_{m_0}^{(0)}/m_0$, where $s_{m_0}^{(0)}$ corresponds to the time that H takes to simulate the first m_0 steps of computation by G. Recall that the delay threshold D_k defined in (7) is an upper bound on the total delay in any live depth-k interval. The recurrences are as follows.

$$s_t^{(k)} = s_t^{(k+1)} + D_k \qquad \text{for} \qquad 1 \le t \le m_{k+1}, \tag{10}$$

$$s_t^{(k)} = s_{t-m_{k+1}}^{(k)} + s_{m_{k+1}}^{(k)} \qquad \text{for} \quad m_{k+1} + 1 \le t \le m_k. \tag{11}$$

The base of the recurrence is defined to be

$$s_{m_k}^{(k)} = s_1^{(k)} = 1 \qquad \text{for } k = k_{\max}. \tag{12}$$

Let the *left endpoint* of interval I be the leftmost unremoved processor in I, and let the *right endpoint* be the rightmost unremoved processor in I. (See Figure 10.) For notational simplicity, we assume that I is the leftmost live depth-k interval and is assigned databases $b_1, \ldots, b_x$. Let $B_k = \{(i, t) : 1 \le i \le x, 1 \le t \le m_k\}$. The proof of the following theorem describes how algorithm Overlap performs the simulation.

Theorem 4.4. *For $1 \le t \le m_k$, if pebbles $(0, t)$ and $(x + 1, t)$ are known by time step $s_t^{(k)}$ by the left and right endpoints of interval I, respectively, then by time step $s_t^{(k)}$ every pebble (i, t) in B_k is computed by all the processors in interval I that have a copy of database b_i.*

Proof. We proceed by a backward induction on k. At level $k = k_{\max}$, we have $m_k = 1$ and box B_k has size $x \times 1$. Since the remaining processors of I have load one,

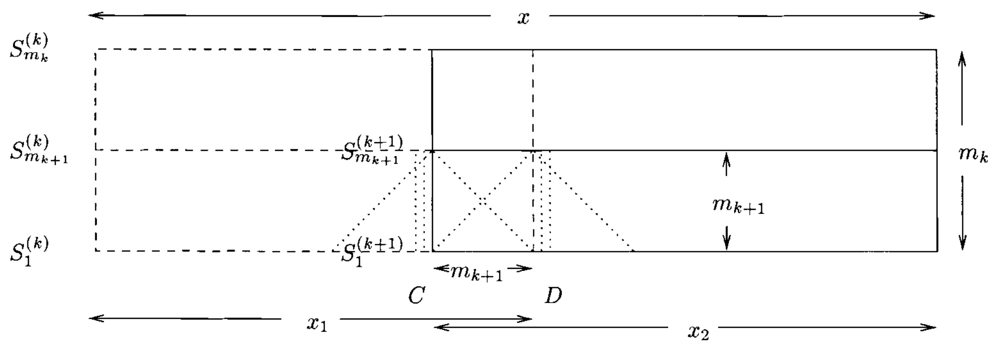


FIG. 11. *The box of pebbles B_{k+1} has size $x_1 \times m_{k+1}$ and is represented by the lower left box with a dashed boundary. B'_{k+1} has size $x_2 \times m_{k+1}$ and is represented by the lower right box with a solid boundary. B_k is the union of all four boxes. For interval I to compute every pebble in B_k, I_1 and I_2 (the live children of I) recursively compute B_{k+1} and B'_{k+1}. Once the bottom half of B_k is computed the top half is computed in a similar manner.*

each processor computes one pebble in B_k. By definition, $s_1^{(k)} = 1$. Hence, the base of the induction holds.

Suppose that the inductive hypothesis is true for $k+1$. Note that the hypothesis can be applied to any depth $k+1$ interval. Let us concentrate on I, the leftmost live depth-k interval. Suppose I is labeled with x. There are two cases to consider.

Case 1. Suppose I has two live children I_1 and I_2 that are labeled with x_1 and x_2, respectively. By construction $x = x_1 + x_2 - m_{k+1}$. Let $B_{k+1} = \{(i,t) : 1 \le i \le x_1, 1 \le t \le m_{k+1}\}$. Let $y = x_1 - m_{k+1}$ and $B'_{k+1} = \{(i,t) : y+1 \le i \le x, 1 \le t \le m_{k+1}\}$. Let column C consist of pebbles (y,t) and column D consist of pebbles (x_1+1,t), where $1 \le t \le m_{k+1}$. Note that boxes B_{k+1} and B'_{k+1} have an overlap of width m_{k+1}; i.e., the m_{k+1} columns between C and D are common to both B_{k+1} and B'_{k+1}. (See Figure 11.) Two observations can be made from the inductive hypothesis.

- **Observation 1.** For $1 \le t \le m_{k+1}$, every pebble (y,t) in column C can be computed by I_1 by time step $s_t^{(k+1)}$ *without any conditions on pebbles* $(0,t)$ *and* (x_1+1,t). Since C and D are m_{k+1} columns apart and $x_1 \ge 2m_{k+1}$ by the construction of Stage 2, the pebbles in column C therefore do not depend on the pebbles $(0,t)$ and (x_1+1,t). (The dotted diagonal lines in Figure 11 show the dependencies of columns C and D.)
- **Observation 2.** Let $z \ge 0$ be some constant. For $1 \le t \le m_{k+1}$, if the value of pebbles $(0,t)$ and (x_1+1,t) are known at time step $s_t^{(k+1)} + z$ by the left and right endpoints of interval I_1, respectively, then by time step $s_t^{(k+1)} + z$, every pebble (i,t) in B_{k+1} is computed. This is true because there is no difference between starting the simulation at time step z and at time step 0.

Similar statements can be made about the box B'_{k+1} and column D. Now suppose that the value of pebbles $(0,t)$ and $(x+1,t)$ are known at time step $s_t^{(k)}$ by the left and right endpoints of interval I, respectively. Observation 1 and the inductive hypothesis imply that any pebble (y,t) in column C can be computed by I_1 by time $s_t^{(k+1)}$. Since the total delay in interval I is at most D_k, then the left endpoint of interval I_2 can receive the pebble (y,t) (together with any relevant database changes) by time $s_t^{(k+1)} + D_k$, which equals $s_t^{(k)}$ (10). Similarly, all of the pebbles in column D can be sent to the right endpoint of interval I_1 by time $s_t^{(k)}$. Since $s_t^{(k)}$ is greater than

$s_t^{(k+1)}$ by a constant amount, namely, D_k, for $1 \leq t \leq m_{k+1}$, Observation 2 and the inductive hypothesis imply that pebbles (i, t) in box B_{k+1} (resp., B'_{k+1}) are computed by I_1 (resp., I_2) by time $s_t^{(k)}$. Therefore, pebbles (i, t) in the bottom half of B_k are computed by time $s_t^{(k)}$.

Once the bottom half of B_k is simulated I simulates the top half in a similar manner. Thus, pebbles (i, t) in the top half of B_k are computed by time $s_{m_{k+1}}^{(k)} + s_{t-m_{k+1}}^{(k)}$, which equals $s_t^{(k)}$ (11).

Case 2. The case in which I has one live child is simpler. Let I_1 be the child of I. By construction, I_1 has label $x_1 = x$. By Observation 2 and the induction hypothesis, if the values of the pebbles $(0, t)$ and $(x_1 + 1, t)$, for $1 \leq t \leq m_{k+1}$, are known at time steps $s_t^{(k)}$ by the left and right endpoints of interval I_1, respectively, then every pebble (i, t) in B_{k+1} (i.e., the bottom half of B_k) is computed by I_1 by time step $s_t^{(k)}$. Since intervals I and I_1 have the same remaining processors (and hence the same endpoints), the above statement holds for I. Interval I then computes the top half in the same manner. Thus, pebbles (i, t) in the top half of B_k are computed by time $s_{m_{k+1}}^{(k)} + s_{t-m_{k+1}}^{(k)}$, which equals $s_t^{(k)}$ (11).

The inductive step is complete. Hence, given that the value of pebbles $(0, t)$ and $(x + 1, t)$ are known at time step $s_t^{(k)}$ by the left and right endpoints of interval I, all pebbles (i, t) in box B_k are computed by time step $s_t^{(k)}$. ∎

Recall that n' is the label of the tree root and n' is a constant fraction of n by Lemma 4.3. We have the following theorem.

THEOREM 4.5. *Suppose that guest linear array G has n' processors and the host linear array H has n processors and an average delay of d_{ave}. Algorithm* OVERLAP *simulates G with H such that the load on H is one and the slowdown is $O(d_{ave} \log^3 n)$.*

Proof. The load on H follows directly from the database assignment. The box B_0 contains all of the pebbles for the first m_0 steps of computations by G, where $m_0 = \frac{n}{c \log n}$. The root interval I_0 contains all the remaining processors of H. Since pebbles $(0, t)$ and $(n' + 1, t)$ are available at time step 0 by assumption, Theorem 4.4 implies that I_0, i.e., H, computes the pebbles in box B_0 by time $s_{m_0}^{(0)}$. We derive $s_{m_0}^{(0)}$ from the recurrence of $s_t^{(k)}$ in (10) and (11) and the definition of D_k in (7).

$$s_{m_0}^{(0)} = 2^k s_{m_k}^{(k)} + 2kD_0 \text{ for } k = k_{\max}. \tag{13}$$

Therefore, $s_{m_0}^{(0)} \leq \frac{n}{c \log n} + 2cd_{\text{ave}} n \log^2 n = O(d_{\text{ave}} n \log^2 n)$. Since $m_0 = \frac{n}{c \log n}$, the slowdown is $O(d_{\text{ave}} \log^3 n)$. ∎

4.2.4. Bandwidth. It is clear that the bandwidth required for the communication between depth-k intervals is at most the bandwidth of G. Therefore, congestion is not an issue if the bandwidth on H is at least $\log n$ times the bandwidth on G. If, however, the bandwidth on G and H are comparable, then we need to pay an extra factor of $\log n$ in the slowdown.

4.2.5. Improvements. In this section we first modify OVERLAP to achieve work efficiency. So far each host processor is assigned at most one database, and the base of the recurrence is therefore $s_{m_k}^{(k)} = 1$ for $k = k_{\max}$ as defined in (12). Observe that in (13) the second term of $s_{m_0}^{(0)}$ dominates the first term. We can balance the two terms by increasing the value of $s_{m_k}^{(k)}$ for the base case, i.e., increasing the load on the host processors.

In particular, we use an m-processor subarray of the host linear array H to simulate an n-processor guest linear array, where $m = \max\{1, \frac{n}{d_{\text{ave}}} \log^{-3} n\}$ and the subarray has average delay at most d_{ave}. If $m = 1$, the slowdown and the load are both n. Otherwise, we carry out the two-stage process to remove the useless processors of the m-processor subarray as as described in section 4.2.1. The only difference is that the network size is m instead of n, and the variables such as D_k, m_k, and $k_{\max}$ are also defined in terms of m. Each unremoved host processor is assigned $\Theta\left(\frac{n}{m}\right)$ databases, and hence the base case is $s^{(k)}_{m_k} = \Theta\left(\frac{n}{m}\right)$ for $k = k_{\max}$. We obtain

$$s^{(0)}_{m_0} = \Theta\left(\frac{m}{c \log m} \cdot \frac{n}{m} + 2cd_{\text{ave}} m \log^2 m\right)$$

from (13). Since $m = \frac{n}{d_{\text{ave}}} \log^{-3} n$, we have $s^{(0)}_{m_0} = O(n \log^{-1} \frac{n}{d_{\text{ave}}})$. Since $m_0 = \frac{m}{c \log m}$, the slowdown $s^{(0)}_{m_0}/m_0$ is $O(d_{\text{ave}} \log^3 n)$. This implies that the simulation is work preserving.

THEOREM 4.6. *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of $O\left(d_{ave} \log^3 n\right)$, where the host has average delay d_{ave}.*

Combining Theorems 4.1 and 4.6 we can improve the slowdown by a factor of $O(\sqrt{d_{\text{ave}}})$ while preserving efficiency. Suppose that G is an n-processor guest linear array, and H is an n-processor host linear array with average delay d_{ave}. We make use of an intermediate linear array H_0 that has a delay of d_{ave} on every edge. Theorem 4.1 implies that network H_0 can efficiently simulate G with a slowdown of $O(\sqrt{d_{\text{ave}}})$, where $\max\{\frac{n}{\sqrt{d_{\text{ave}}}}, 1\}$ processors of H_0 are used. In the simulation by H_0, every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication. If we treat every $O(d_{\text{ave}})$ steps as one time unit, then H_0 acts like a guest linear array with unit-delay edges and H has a normalized average delay of $O(1)$. Theorem 4.6 implies that H can simulate H_0 with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{d_{\text{ave}}} \log^3 n)$. It is obvious that the combined load is $O(\sqrt{d_{\text{ave}}} \log^3 n)$. Theorem 4.6 is improved to the following.

THEOREM 4.7. *In the database model, an n-processor guest linear array can be efficiently simulated by an n-processor host linear array with a slowdown and a load of $O(\sqrt{d_{ave}} \log^3 n)$, where the host has average delay d_{ave}.*

4.3. Simulating linear arrays on general networks. We generalize algorithm OVERLAP to simulate a guest linear array on an arbitrary bounded-degree connected host network. Given a connected bounded-degree n-processor network H with average delay d_{ave}, we first find a linear array $\mathcal{H}$ that can be embedded one to one to H and has average delay d_{ave}. As discussed in section 2.4 such $\mathcal{H}$ can be found, and $\mathcal{H}$ is used to carry out the simulation. Combined with Theorem 4.7, we obtain Theorem 4.8.

THEOREM 4.8. *An n-processor guest linear array can be efficiently simulated by a connected bounded-degree n-processor host with a slowdown of $O(\sqrt{d_{ave}} \log^3 n)$, where the host has average delay d_{ave}.*

For the same reason as in section 2.4, Theorem 4.8 does not hold when H has unbounded degree.

4.4. Simulating two-dimensional arrays on general networks. Our techniques can also be generalized to simulate a two-dimensional array on any connected bounded-degree network.

THEOREM 4.9. *In the database model, an $n \times n$ guest can be efficiently simulated by a bounded-degree host network with a slowdown of $O(n \log^3 n + \sqrt{nd_{ave}} \log^3 n)$, where the host has average delay d_{ave}.*

Proof. As discussed in section 2.4 there exists a linear array $\mathcal{H}$ such that $\mathcal{H}$ is embedded one to one in H and that $\mathcal{H}$ has average delay $O(d_{\text{ave}})$. The simulation of G on H will be performed by simulating G on $\mathcal{H}$. We first show how to simulate G on an intermediate linear array $\mathcal{H}_0$, where $\mathcal{H}_0$ has delay d_{ave} on all the edges. The size of $\mathcal{H}_0$ depends on the relative sizes of d_{ave} and n.

Case 1. If $d_{\text{ave}} < n$, then $\mathcal{H}_0$ has n processors, each of which simulates one column of processors of G. To simulate one step of G, a processor of $\mathcal{H}_0$ computes n pebbles and then communicates with both of its neighbors. The communication takes at most $n + d_{\text{ave}}$ steps, which is $O(n)$ steps. Hence the slowdown of $\mathcal{H}_0$ simulating G is $O(n)$. Also, in this simulation every $O(n)$ steps of computation interleave with every $O(n)$ steps of communication.

Since $d_{\text{ave}} < n$, if every $O(n)$ step is treated as one time unit, then $\mathcal{H}$ has a normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a guest linear array with unit-delay edges. Therefore, Theorem 4.7 implies that $\mathcal{H}$ can efficiently simulate $\mathcal{H}_0$ with a slowdown of $O(\log^3 n)$. The combined slowdown is therefore $O(n \log^3 n)$.

Case 2. If $d_{\text{ave}} \geq n$, then $\mathcal{H}_0$ has n/x processors, where $x = \sqrt{d_{\text{ave}}/n}$. Each processor of $\mathcal{H}_0$ simulates $3x$ columns of G, overlapping x columns with each neighbor. (The redundant computation used here is similar to that in Theorem 4.1.) To simulate x steps of G, each processor of $\mathcal{H}_0$ computes at most $3x^2 n$ pebbles and then communicates with both of its neighbors. The communication takes at most $3x^2 n + d_{\text{ave}}$ steps, which is $O(d_{\text{ave}})$ steps. Hence the slowdown of simulating every x steps is d_{ave}/x, which is $O(\sqrt{nd_{\text{ave}}})$. Also, in this simulation every $O(d_{\text{ave}})$ steps of computation interleave with every $O(d_{\text{ave}})$ steps of communication.

If every $O(d_{\text{ave}})$ step is treated as one time unit, $\mathcal{H}$ has normalized average delay $O(1)$ and $\mathcal{H}_0$ acts like a linear array with unit-delay edges. If n/x processors of $\mathcal{H}$ are used to simulate $\mathcal{H}_0$, Theorem 4.7 implies a slowdown of $O(\log^3 \frac{n}{x})$, which is $O(\log^3 n)$. The combined slowdown is therefore $O(\sqrt{nd_{\text{ave}}} \log^3 n)$. □

The above technique can be applied to the dataflow model, where $\mathcal{H}_0$ simulates G in the same manner and $\mathcal{H}$ simulates $\mathcal{H}_0$ with a slowdown of $O(1)$ in both cases.

THEOREM 4.10. *In the dataflow model, an $n \times n$ guest can be efficiently simulated by a bounded-degree host network with a slowdown of $O(n + \sqrt{nd_{ave}})$, where the host has average delay d_{ave}.*

4.5. Lower bounds. In this section we discuss the impact on the slowdown of the simulation when the number of copies of each database is bounded and the load is a constant. We consider the case in which each database can have one copy and the case in which each database can have at most two copies. Notice that although we are restricting the number of copies of each database to either one or two, a particular processor in the host can have a copy of many databases.

For the case in which each database is allowed one copy we give an example to show that the slowdown can be $d_{\max}$. Let G and H_1 be n-processor guest and host linear arrays. Every $\sqrt{n}$th edge of H_1 has a delay of $\sqrt{n}$, and all other edges have unit delay. Therefore, H_1 has an average delay of $O(1)$. If at most $\sqrt{n}$ processors of H_1 have copies of databases, then by a work argument the slowdown when H_1 simulates G is at least $\sqrt{n}$. Otherwise, there exist databases b_i and b_{i+1} such that they are assigned to processors p and q of H_1, respectively, and that the delay between p and q is at least $\sqrt{n}$. Hence, for all time steps t, processor p cannot compute pebble (i, t)

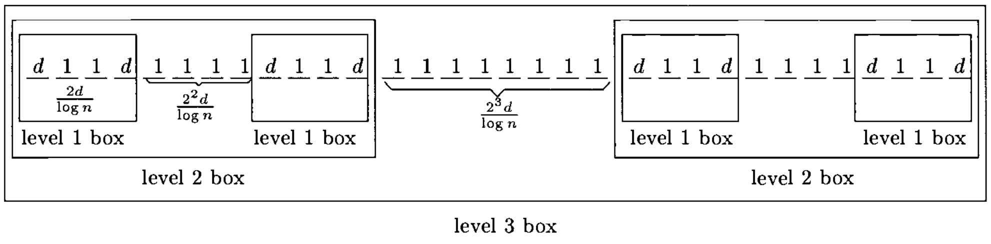


FIG. 12. *A level-3 box. Host network H_2 is a level-k box, where $k = \log \frac{n}{d}$.*

until $\sqrt{n}$ steps after q computes $(i+1, t-1)$, and q cannot compute $(i+1, t)$ until $\sqrt{n}$ steps after p computes $(i, t-1)$. This implies a slowdown of $d_{\max} = \sqrt{n}$, whereas d_{ave} is a constant. Note that the above argument makes no assumption on the load.

THEOREM 4.11. *If each database can have at most one copy, then there exists a host with $d_{ave} = O(1)$ such that the slowdown is $\Omega(\sqrt{n})$.*

For the case in which each database is allowed at most two copies we construct a host network H_2 whose average delay is $O(1)$, but for which the simulation slowdown is $\Omega(\log n)$. Network H_2 is made up of $\Theta(n)$ processors and the edge delays are either 1 or d. The following is a recursive construction of H_2 in which we define a series of boxes. (See Figure 12.) We regard H_2 as a level-k box, where $k = \log \frac{n}{d}$. Network H_2 consists of two level $k-1$ boxes that are connected by $\frac{2^k d}{\log n}$ edges of delay 1. In general, a level-ℓ box, for $1 \le \ell \le k$, consists of two level $\ell - 1$ boxes that are connected by $\frac{2^\ell d}{\log n}$ edges of delay 1. We say that these $\frac{2^\ell d}{\log n}$ processors are in a *segment.* A level-0 box consists of a single edge of delay d.

Let $d = \log n$. Since a level-ℓ box contains 2^ℓ edges of delay d and $\frac{2^\ell d \ell}{\log n}$ edges of delay 1, H_2 has $\Theta(n)$ processors and constant average delay d_{ave}. Furthermore, Lemma 4.12 holds.

LEMMA 4.12. *If processors p and q are in two different segments I and J, then the delay between p and q is at least* $\min\left\{\frac{u}{2}\log n, \frac{v}{2}\log n\right\}$*, where u and v are the numbers of processors in segments I and J, respectively. In particular, the delay between p and q is at least $d = \log n$.*

THEOREM 4.13. *If each database is allowed at most two copies and the load is a constant c, then there exists a host with $d_{ave} = O(1)$ such that the slowdown is $\Omega(\log n)$.*

Proof. We consider the following two cases when H_2 simulates G.

Case 1. There exists some "overlap" in the database assignment. In particular, suppose databases b_i, $b_{i+1}, \ldots,$ b_{i+j} are assigned to processors in segment I and $b_{i+1}, \ldots,$ b_{i+j}, b_{i+j+1} are assigned to segment $J \ne I$ for some $j \ge 1$. Suppose also that the other copy of b_{i+j+1} is assigned to $J' \ne I$ and the other copy of b_i is assigned to $I' \ne J$. Notice that pebbles of the form $(i+k, t)$, for $1 \le k \le j$, can only be computed by processors in segment I or J. Since the load is c, the number of processors in segment I is at least j/c. The same is true for segment J. We shall find a path of $4j$ pebbles such that either a delay of $O(j \log n)$ occurs, or a delay of $\log n$ occurs $O(j)$ times during the simulation. For simplicity we assume that j is even. The case in which j is odd is similar.

We use a triple (i, t, p) to say that processor p computes pebble (i, t), and we use expressions of the form $(i, t, p) \leftarrow (i-1, t-1, q)$ to indicate dependency. That is, processor p receives pebble $(i-1, t-1)$ from processor q before p computes (i, t).

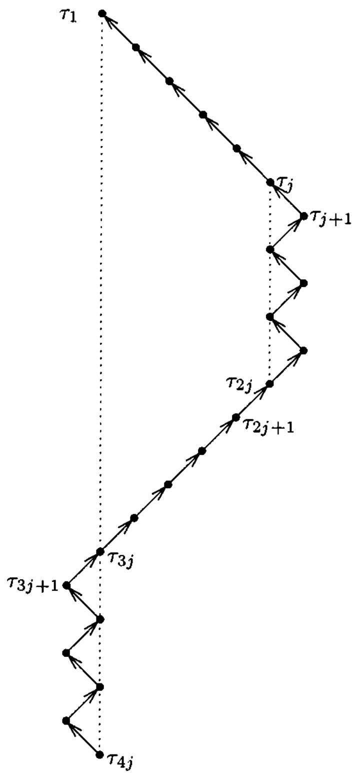


FIG. 13. *A path of* $4j$ *pebbles, where* j *is even.*

(Note that p may be the same as q.) Consider the computation of the following path of $4j$ pebbles, $\tau_1 \leftarrow \cdots \leftarrow \tau_{4j}$, where τ_k is a triple of the form

$$\tau_k = \begin{cases} (i+k, t-k, p_k) & \text{for } k \in A, \text{ where } \quad A = \{k : 1 \le k \le j\}, \\ (i+j+1, t-k, p_k) & \text{for } k \in B, \text{ where } \quad B = \{k \text{ odd} : j < k \le 2j\}, \\ (i+j, t-k, p_k) & \text{for } k \in C, \text{ where } \quad C = \{k \text{ even} : j < k \le 2j\}, \\ (i-k+3j, t-k, p_k) & \text{for } k \in D, \text{ where } \quad D = \{k : 2j < k \le 3j\}, \\ (i+1, t-k, p_k) & \text{for } k \in E, \text{ where } \quad E = \{k \text{ even} : 3j < k \le 4j\}, \\ (i, t-k, p_k) & \text{for } k \in F, \text{ where } \quad F = \{k \text{ odd} : 3j < k \le 4j\}. \end{cases}$$

This path goes backward in time and zigzags during time steps k for $k \in B \cup C \cup E \cup F$. (See Figure 13.)

By assumption, processors p_k, for $k \in C \cup E$, can only belong to segment I or J. If processors p_k, for $k \in C \cup E$, do not belong to the same segment, then Lemma 4.12 implies a delay of $\frac{j}{2c} \log n$ for the communication between segments I and J. Hence, it takes more than $\frac{j}{2c} \log n$ steps to compute this path of $4j$ pebbles. Otherwise, processors p_k, for $k \in C \cup E$, all belong to segment I. Lemma 4.12 implies a delay of $\log n$ in computing every τ_k for $j < k \le 2j$. This is because processors p_k, for $k \in B$, cannot be in segment I by assumption. Similarly, if processors p_k, for $k \in C \cup E$, all belong to segment J, then there is a delay of $\log n$ in computing every τ_k for $3j < k \le 4j$. Hence, it takes more than $j \log n$ steps to compute this path of $4j$ pebbles.

We can repeat this argument for every $4j$ steps. Hence the slowdown is $\Omega(\log n)$.

Case 2. There exists no "overlapping" of the databases as in Case 1. Let $b_i, \ldots, b_j$, for $j \geq i$, be the longest sequence of consecutive databases assigned to one segment. Call this segment I and the sequence of databases S_I. Notice that processors in I do not have a copy of b_{i-1}. Let J be a segment that is assigned a copy of b_{i-1}. Let S_J be the sequence of consecutive databases such that b_{i-1} is a member of S_J and that each member of S_J has a copy in J. If b_i were a member of S_J, then either the database sequences S_J and S_I would produce the "overlapping" pattern sufficient for Case 1 or S_J would be longer than S_I. This latter case contradicts the definition of S_I. Hence, any segment that has a copy of b_{i-1} cannot have a copy of b_i. This implies that the processors computing the pebbles in the $(i-1)$st and ith column are at least $\log n$ delay apart by Lemma 4.12. Therefore, the slowdown is $\Omega(\log n)$. □

5. Conclusions.

In this paper we presented methods for latency hiding in simple networks such as linear arrays and two-dimensional arrays. Ultimately, we are interested in the efficient implementation of algorithms designed for networks that appear often in the architectures of parallel computers, such as trees, arrays, butterflies, and hypercubes, on a network with arbitrary topology and arbitrary link delays, such as NOWs. The special case in which two networks have identical topology but different link delays is a starting point where we can study the effect of latencies in isolation. Indeed, the general case of simulating a unit-delay guest on a host with arbitrary delays and arbitrary topology so as to minimize slowdown seems to be a very challenging problem.

REFERENCES

[1] *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, 1991.

[2] F. Afrati, C. H. Papadimitriou, and G. Papageorgiou, *Scheduling DAGS to Minimize Time and Communication*, Aegean Workshop on Computing (AWOC), Corfu, Greece, 1988, pp. 134–138.

[3] Y. Aumann and M. Ben-Or, *Computing with faulty arrays*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, 1992, pp. 162–169.

[4] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, *Implementation of a portable nested data-parallel language*, in Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, ACM Press, New York, 1993, pp. 102–112.

[5] P. Chretienne, *A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints*, European J. Oper. Res., 43 (1989), pp. 225–230.

[6] R. Cole, B. Maggs, and R. Sitaraman, *Multi-scale self-simulation: A technique for reconfiguring arrays with faults*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, CA, 1993, pp. 561–572.

[7] J. Y. Colin and P. Chretienne, *C.P.M. scheduling with small communication delay and task duplication*, Oper. Res., 39 (1991), pp. 680–684.

[8] D. N. Jayasimha and M. C. Loui, *The Communication Complexity of Parallel Algorithms*, Technical Report CSRD 629, University of Illinois at Urbana–Champaign, 1986.

[9] H. Jung, L. Kirousis, and P. Spirakis, *Lower bounds and efficient algorithms for multiprocessor scheduling for dags with communications delays*, Inform. and Comput., 105 (1993), pp. 94–104.

[10] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas, *Asymptotically tight bounds for computing with faulty arrays of processors*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 285–296.

[11] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg, *Work-preserving emulations of fixed-connection networks*, J. ACM, 44 (1997), pp. 104–147.

[12] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*, Morgan-Kaufmann, San Mateo, CA, 1992.

[13] F. T. Leighton, B. Maggs, and R. Sitaraman, *On the fault tolerance of some popular bounded-degree networks*, SIAM J. Comput., 27 (1998), pp. 1303–1333.

[14] C. E. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. S. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak, *The network architecture of the connection machine CM*-5, in Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, 1992, pp. 272–285.

[15] C. E. Leiserson, S. Rao, and S. Toledo, *Efficient out-of-core algorithms for linear relaxation using blocking covers*, J. Comput. System Sci., 54 (1997), pp. 332–334.

[16] M. Palis, J.-C. Liou, S. Rajasekaran, S. Shende, and D. L. Wei, *On-Line Scheduling of Dynamic Trees*, manuscript, 1994.

[17] M. Palis, J.-C. Liou, and D. L. Wei, *Task Clustering and Scheduling for Distributed Memory Parallel Architectures*, Technical Report Fukushima 965-80, University of Aizu, Japan, 1994.

[18] C. H. Papadimitriou and J. D. Ullman, *A communication-time tradeoff*, SIAM J. Comput., 16 (1987), pp. 639–646.

[19] C. H. Papadimitriou and M. Yannakakis, *Towards an architecture-independent analysis of parallel algorithms*, SIAM J. Comput., 19 (1990), pp. 322–328.

[20] M. O. Rabin, *Efficient dispersal of information for security, load balancing and fault tolerance*, J. ACM, 36 (1989), pp. 335–348.

[21] B. J. Smith, *Architecture and applications of the HEP multiprocessor computer system*, in Real-time Signal Processing IV, 298, SPIE, Bellingham, WA, 1981, pp. 241–248.

[22] L. W. Tucker and G. G. Robertson, *Architecture and applications of the connection machine*, Computer, 21 (1988), pp. 26–38.

[23] L. G. Valiant, *Bulk-Synchronous Parallel Computers*, Technical Report TR-08-89, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1989.

[24] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), pp. 103–111.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 2, pp. 648–670

IMPROVED APPROXIMATION GUARANTEES FOR PACKING AND COVERING INTEGER PROGRAMS[*]

ARAVIND SRINIVASAN[†]

Abstract. Several important NP-hard combinatorial optimization problems can be posed as *packing/covering integer programs*; the *randomized rounding* technique of Raghavan and Thompson is a powerful tool with which to approximate them well. We present one elementary unifying property of all these integer linear programs and use the FKG correlation inequality to derive an improved analysis of randomized rounding on them. This yields a *pessimistic estimator*, thus presenting deterministic polynomial-time algorithms for them with approximation guarantees that are significantly better than those known.

Key words. approximation algorithms, combinatorial optimization, correlation inequalities, covering integer programs, derandomization, integer programming, linear programming, linear relaxations, packing integer programs, positive correlation, randomized rounding, rounding theorems

AMS subject classifications. 90C27, 68R05, 60C05, 90B80, 90C05, 90C10

PII. S0097539796314240

1. Introduction.

Several important NP-hard combinatorial optimization problems, such as basic problems on graphs and hypergraphs, can be posed as *packing/covering integer programs*; the *randomized rounding* technique of Raghavan and Thompson is a powerful tool with which to approximate them well [30]. We present an elementary property (*positive correlation*) of all these integer linear programs (ILPs); we then use the FKG inequality (Fortuin, Kasteleyn, and Ginibre [14], Sarkar [31]) to derive an improved analysis of randomized rounding on the problems. Interestingly, this yields a *pessimistic estimator*, thus presenting deterministic polynomial algorithms with approximation guarantees that are significantly better than those known, in a unified way.

1.1. Previous work.

Let Z_+ and $\Re_+$ denote the nonnegative integers and the nonnegative reals, respectively. For a (column) vector v, let v^T denote its transpose and let v_i stand for the ith component of v. We first define the packing and covering integer programs.

DEFINITION 1.1. *Given $A \in [0,1]^{n\times m}$, $b \in [1,\infty)^n$, and $c \in [0,1]^m$ with $\max_j c_j = 1$, a packing (respectively, covering) integer program (PIP) (respectively, CIP) seeks to maximize (respectively, minimize) $c^T \cdot x$ subject to $x \in Z_+^m$ and $Ax \leq b$*

[*]Received by the editors October 10, 1996; accepted for publication (in revised form) October 28, 1998; published electronically November 23, 1999. Part of the work was done at the National University of Singapore, supported in part by National University of Singapore Academic Research Fund grant RP970607. Part of the work was done while the author was at the School of Mathematics, Institute for Advanced Study, Princeton, NJ, and was supported by grant 93-6-6 of the Alfred P. Sloan Foundation to the Institute for Advanced Study. Part of the work was done while the author was at the NSF Center for Discrete Mathematics and Theoretical Computer Science and was supported in part by NSF grant NSF-STC91-19999 and in part by the New Jersey Commission on Science and Technology. Part of the work was done while the author was visiting the Department of Computer Science, University of Warwick, Coventry, UK, supported in part by the ESPRIT Basic Research Action Programme of the EC under contract 7141 (project ALCOM II). A preliminary version of this work appeared in the *Proceedings of the ACM Symposium on the Theory of Computing*, 1995, pp. 268–276.

http://www.siam.org/journals/sicomp/29-2/31424.html

[†]School of Computing, National University of Singapore, Singapore 119260, Republic of Singapore (aravind@comp.nus.edu.sg).

(respectively, $Ax \geq b$). Furthermore if $A \in \{0,1\}^{n \times m}$, we assume that each entry of b is integral. We also define $B = \min_i \ b_i$; we assume without loss of generality (w.l.o.g.) that $B \geq 1$.

Although there usually are no restrictions on the entries of A, b, and c beyond nonnegativity, it is easily seen that the above restrictions are w.l.o.g. because of the following. First, we may assume that $\forall i,j$, A_{ij} is at most b_i. If this is not true for a PIP, then we may as well set $x_j := 0$; if this is not true for a CIP, we can reset $A_{ij} := b_i$. Next, by scaling each row of A such that $\max_j \ A_{i,j} = 1$ for each row i and by scaling c so that $\max_j \ c_j = 1$, we get the above form for A, b, and c. (In particular, we get $B \geq 1$ since $A_{i,j} \leq b_i \ \forall i,j$.) Finally, if $A \in \{0,1\}^{n \times m}$, then for a PIP we can always reset $b_i := \lfloor b_i \rfloor$ for each i, and for a CIP we can reset $b_i := \lceil b_i \rceil$, hence the assumption on the integrality of each b_i in this case.

Remark. The reader is requested to take note of the parameter B; it will occur frequently in the rest of the paper. Whenever we use the symbol B as a parameter for any given problem, B will play the same role in the "natural" PIP/CIP formulation of the problem, as it does in Definition 1.1.

As mentioned above, PIPs and CIPs model some basic problems in combinatorial optimization, but most of these problems are NP-hard; hence we are interested in efficient approximation algorithms for PIPs and CIPs with a good performance guarantee. We now turn to an important technique for approximating integer linear programs—"relaxing" their integrality constraints and considering the resulting linear program.

DEFINITION 1.2. *The standard linear programming (LP) relaxation of PIPs/CIPs lets $x \in \Re_+^m$; given a PIP/CIP, x^* and y^* denote, respectively, an optimal solution to, and the optimum value of, this relaxation. (For packing, we also allow constraints of the form $x_i \in \{0,1,\ldots,d_i\}$, where d_i is some positive integer; the LP relaxation sets $x_i \in [0,d_i]$ here.)*

Given a PIP or a CIP, we can solve its LP relaxation efficiently. However, how do we handle the possibility of possibly fractional entries in x^*? We need some mechanism to "round" fractional entries in x^* to integers, suitably. One possibility is to round every fractional value x_i^* to the closest integer, with some tie-breaking rule if x_i^* is half of an integer. However, it is known that such "thresholding" methods are of limited applicability.

A key technique with which to approximate a class of integer programming problems via a new rounding method—*randomized rounding*—was proposed in [30]. Given a positive real v, the idea is to look at its fractional part as a probability: round v to $\lfloor v \rfloor + 1$ with probability $v - \lfloor v \rfloor$, and round v to $\lfloor v \rfloor$ with probability $1 - v + \lfloor v \rfloor$. This has the nice property that the expected value of the result is v. How can we use this for packing and covering problems? Consider a PIP, for instance. Solve its LP relaxation and set $x_i' := x_i^*/\alpha$ for some parameter $\alpha > 1$ to be fixed later; this scaling down by α is done to boost the chance that the constraints in the PIP are all satisfied; recall that they are all $\leq$-constraints. Now define a random $z \in Z_+^m$, the outcome of randomized rounding, as follows. Independently for each i, set z_i to be $\lfloor x_i' \rfloor + 1$ with probability $x_i' - \lfloor x_i' \rfloor$ and $\lfloor x_i' \rfloor$ with probability $1 - (x_i' - \lfloor x_i' \rfloor)$.

We now need to show that all the constraints in the PIP are satisfied *and* that $c^T \cdot z$ is not "much below" y^*, with reasonable probability; we also need to choose α suitably. This is formalized in [30] as follows. In what follows, let $\Pr(\mathcal{E})$ denote the probability of an event $\mathcal{E}$ and $\mathbf{E}[X]$ denote the expected value of random variable X.

As seen above, an important observation is that $\mathbf{E}[z_i] = x'_i$. Hence,

$$\mathbf{E}[(Az)_i] = (Ax')_i \leq b_i/\alpha,\ 1 \leq i \leq n; \quad \mathbf{E}[c^T \cdot z] = y^*/\alpha.$$

For some $\beta > 1$ to be fixed later, define $(n+1)$ "bad" events

$$E_i \equiv ((Az)_i > b_i),\ 1 \leq i \leq n; \quad E_{n+1} \equiv (c^T \cdot z < y^*/(\alpha\beta)). \tag{1.1}$$

Now, z is an $(\alpha\beta)$-approximate solution to the PIP if

$$\bigwedge_{i=1}^{n+1} \overline{E_i} \neq \phi \tag{1.2}$$

holds. How small a value for $(\alpha\beta)$ can we achieve while still guaranteeing (1.2)? Bounding

$$\Pr\left(\bigvee_{i=1}^{n+1} E_i\right) \leq \sum_{i=1}^{n+1} \Pr(E_i), \tag{1.3}$$

we can pick $\alpha, \beta > 1$ such that $\sum_{i=1}^{n+1} \Pr(E_i) < 1$ holds, using the Chernoff–Hoeffding (CH) bounds. This gives us an $(\alpha\beta)$-aproximation z with nonzero probability, which is also made deterministic by Raghavan, using pessimistic estimators [28]. Similar ideas hold for CIPs—the fractions $\{x_i^*\}$ are scaled *up* by some $\alpha > 1$ here. Similar approximation bounds are derived through different methods by Plotkin, Shmoys, and Tardos [27]. See Raghavan [29] for a survey of randomized rounding and Crescenzi and Kann [9] for a comprehensive collection of approximation results on NP-optimization problems.

Although randomized rounding is a unifying idea with which to derive good approximation algorithms, there are better approximation bounds for specific key problems such as set cover (Johnson [21], Lovász [23], Chvátal [8]), hypergraph matching (Aharoni, Erdős, and Linial [1]), and file-sharing in distributed networks (Naor and Roth [26]), each derived through different means. (For instance, for the set cover problem, which will be defined soon, let n denote the number of vertices of a given hypergraph H and $d \leq n$ be the maximum number of vertices in any edge of H. The analysis of randomized rounding via (1.3) for appropriately defined events E_i yields an approximation of $\mathrm{O}(\ln n)$. However, an $\mathrm{O}(\log d)$ approximation is provided in [21], [23], [8].) One reason for this slack stems from bounding $\Pr(\bigvee_{i=1}^{n+1} E_i)$ by $\sum_{i=1}^{n+1} \Pr(E_i)$: to quote Raghavan [28],

> Throughout, we naively (?) sum the probabilities of all bad events—although these bad events are surely correlated. Can we prove a stronger result using algebraic properties (e.g., the rank) of the coefficient matrix? A tighter bound for the probabilistic existence proofs should lead to tighter approximation algorithms.

1.2. Proposed method. We make progress in the above-suggested direction by exploiting an elementary property—positive correlation—of CIPs and PIPs. To motivate this idea, let us take two constraints of a PIP and let E_1 and E_2 be the corresponding bad events, as defined before. For instance, suppose E_1 is the event that $0.1z_1 + z_3 + 0.5z_4 + 0.9z_6 > 1.1$ and E_2 stands for the event that $0.4z_1 + 0.3z_2 + z_5 + 0.1z_6 > 1.2$, where the z_i are all *independent* 0-1 random variables. Now suppose we are given that $\overline{E_1}$ holds. Very roughly speaking, this seems to suggest that "many" among z_1, z_3, z_4, and z_6 were "small" (i.e., zero), which seems to boost the chance that $\overline{E_2}$ holds, too. Formally, the claim is that $\Pr(\overline{E_2}|\overline{E_1}) \geq \Pr(\overline{E_2})$, i.e., that $\Pr(\overline{E_1} \bigwedge \overline{E_2}) \geq \Pr(\overline{E_1}) \cdot \Pr(\overline{E_2})$. This "intuitively clear" fact then can be easily generalized to

$$\forall k\ \forall 1 \leq i_1 < i_2 < \cdots < i_k \leq n, \ \Pr\left(\bigwedge_{j=1}^{k} \overline{E_{i_j}}\right) \geq \prod_{j=1}^{k} \Pr(\overline{E_{i_j}}). \tag{1.4}$$

In other words, (1.4) claims that the constraints are positively correlated—given that all of any given subset of them are satisfied, the conditional probability that any other constraint is also satisfied cannot go below its unconditional probability.

We prove (1.4), which seems plausible, using the FKG inequality. Thus,

$$\Pr\left(\bigvee_{i=1}^{n+1} E_i\right) \leq \Pr\left(\bigvee_{i=1}^{n} E_i\right) + \Pr(E_{n+1}) \leq 1 - \left(\prod_{i=1}^{n} (1 - \Pr(E_i))\right) + \Pr(E_{n+1}), \tag{1.5}$$

which is always as good as, and most often much better than, (1.3). (For a detailed study of the FKG inequality, see, e.g., Graham [15] and Chapter 6 of Alon, Spencer, and Erdős [3].)

It is not hard to verify such a property for CIPs also. Why have we been so lucky as to have positive correlation among the constraints of PIPs and CIPs? The features of PIPs and CIPs which guarantee this are

- all the entries of the matrix A are nonnegative, and
- all the constraints "point" in the same direction.

Of course, given that all of any given subset of the constraints are *violated*, the conditional probability that any other constraint is also violated cannot go below its unconditional probability; however, we will not have to deal with this situation! Also, (1.4) may not hold if the z_i's are not independent.

This approach usually only guarantees that z is a "good" approximation with very low (albeit positive) probability; it does not seem to provide a randomized algorithm with any good success probability. However, the structure of PIPs and CIPs implies a subadditivity property that yields a *pessimistic estimator* (a notion to be recalled in section 2); we thus get deterministic polynomial-time algorithms achieving these improved approximation bounds. The problem in arriving at a good pessimistic estimator is that while the previous estimator $\sum_{i=1}^{n+1} \Pr(E_i)$ (i.e., the one used in [28] and in related papers) is upper-bounded by $\mathbf{E}[Z]$ (for some random variable Z) on applying the CH bounds, such a fact does not seem to hold here. Nevertheless, the structure of CIPs/PIPs—in particular, the two simple properties itemized above—help to provide a good pessimistic estimator. This is a point that we would like to stress.

Thus we get, in a unified way, improved bounds on the *integrality gap*

$$\max\{(c^T \cdot z)/y^*, y^*/(c^T \cdot z)\}$$

and hence improved approximation algorithms for all PIPs and CIPs. In particular, we improve on the above-mentioned results of [21], [23], [1], [26]; our bound is incomparable with that of [8].

Positive correlation has been used in the context of network reliability by Esary and Proschan [11]; see, e.g., Chapter 4.1 in [32]. It has also been used for the set cover problem, an important covering problem, to be defined below, by Bertsimas and Vohra [5].

1.3. Approximation bounds achieved. Our best improvements are for PIPs. For PIPs, the standard analysis of randomized rounding, i.e., the usage of (1.3), guarantees integral solutions of value $t_1 = \Omega(y^*/n^{1/B})$ and $t_2 = \Omega(y^*/n^{1/(B+1)})$, respectively, if $A \in [0,1]^{n\times m}$ and $A \in \{0,1\}^{n\times m}$. (For completeness, we prove this in the appendix.) Our method provides $\Omega(\min\{y^*, (K_0 t_1)^{B/(B-1)}\})$ and $\Omega(\min\{y^*, t_2^{(B+1)/B}\})$ bounds, respectively, where $0 < K_0 < 1$ is an absolute constant, thus improving on the previous solutions. For instance, if $A \in \{0,1\}^{n\times m}$, $y^* = \Theta(n)$, and $B = 1$, we get an integral solution of value $\Theta(n)$, as opposed to the previous $\Omega(\sqrt{n})$ bound. This method also gives Turán's classical theorem on independent sets in graphs [39] to within a constant factor.

An important packing problem where $A \in \{0,1\}^{n\times m}$ is *simple B-matching in hypergraphs* [23]: given a hypergraph with nonnegative edge weights, find a maximum weight collection of edges such that no vertex occurs in more than B of them. Usual hypergraph matching has $B = 1$ and is a well-known NP-hard problem. To our knowledge, the only known good bound for this problem, apart from the standard analysis of randomized rounding, was provided by the work of [1], which focused on the special case of *unweighted* edges. The methods of [1] can be used to show that if f is the minimum size of an edge in the hypergraph, then there exists an integral matching of value at least

$$\frac{(y^*)^2}{B^2 n - (f-1)(y^*)^2/\min\{m,n\}} \geq \frac{(y^*)^2}{B^2 n}.$$

This matches our result to within a constant factor for $B = 1$. However, this bound *worsens* as B increases, while the standard analysis, as well as our present analysis, of randomized rounding show that the integrality gap decreases as B increases. (An interesting point is that approximation algorithms for packing and covering often have an approximation ratio that is independent of y^*. The work of [1], and our work here, are exceptions.)

For covering, we prove an

$$1 + \mathrm{O}(\max\{\ln(nB/y^*)/B, \sqrt{\ln(2\lceil nB/y^*\rceil)/B}\}) \tag{1.6}$$

integrality gap and derive the corresponding deterministic polynomial-time approximation algorithm. (To parse the "$\ln(2\lceil t\rceil)$" term for $t > 0$, first take "t large" (say, $t \geq 2$) to be the typical case, wherein the term equals $\ln t + \Theta(1)$. If t is "small," say, $t < 2$, then the term is $\Theta(1)$.) This improves on the

$$1 + \mathrm{O}(\max\{(\ln n)/B, \sqrt{(\ln n)/B}\})$$

bound given by the standard analysis of randomized rounding. Also, Dobson [10] and Fisher and Wolsey [13] bound the performance of a natural greedy algorithm for CIPs; their results are as follows. For a given CIP, let $\gamma_1 = \min_i \max_j A_{i,j}/c_j$, and let $\gamma_2 = \max_j(\sum_{i=1}^n A_{i,j}/c_j)$. Then the work of [13] shows that the greedy algorithm produces a solution of value at most $y^*(1 + \ln(\gamma_2/\gamma_1))$. Alternatively, if each row of the linear system $Ax \geq b$ is scaled so that the minimum nonzero entry in the row is at least 1, then if OPT denotes the value of the optimal *integral* solution to the CIP, it is shown in [10] that the greedy algorithm produces a solution of value at most $OPT(1 + \ln(\max_j \sum_{i=1}^n A_{i,j}))$. However, more general covering problems that have constraints of the form $x_j \leq u_j$ are also considered in [10]. Our bound is incomparable with these previous bounds, but for any given (A, b, c), our bound is always better for CIP instances parametrized by $(A, \lambda b, c)$ if the scalar λ is more than a certain threshold thresh(A, b, c). See [5] for a detailed study of approximating CIPs; our work improves on all of their randomized rounding bounds except for their *weighted* CIPs (wherein it is not the case that $c_i = 1$ for all i), for which our bounds are incomparable with theirs.

An important subclass of the CIPs models the *unweighted set cover problem*: $\forall i, j$, $A_{i,j} \in \{0, 1\}$, $b_i = 1$, and $c_j = 1$, here. The combinatorial interpretation is that we have a hypergraph $H = (V, E)$ and wish to pick a minimum cardinality collection of the edges so that every vertex is covered. (When viewed as an LP, this is the "dual" of the hypergraph matching problem.) The rows correspond to V and the columns to E. Clearly, this problem requires that $x \in \{0, 1\}^m$, which is not guaranteed by Definition 1.1; however, for this problem, any $x \in Z_+^m$ with $Ax \geq b$ trivially yields a $y \in \{0, 1\}^m$ with $Ay \geq b$ and $c^T \cdot y \leq c^T \cdot x$ (set $y_i = \min\{1, x_i\}$).

For set cover, we tighten the constants in (1.6) to derive a $\ln(n/y^*) + \ln\ln(n/y^*) + \mathrm{O}(1)$ approximation bound. Recent work of Feige [12], improving earlier results of [24], [4], shows that for any fixed $\epsilon > 0$, approximating this problem to within $(1 - \epsilon) \ln n$ is likely to take superpolynomial time. However, this problem is important enough to study approximations parametrized by other parameters of A, b, and c that are at least as good as and often much better than $\Theta(\log n)$; for instance, the work of [21], [23], [8] shows a $\ln d + \mathrm{O}(1)$ approximation bound, where d is the maximum column sum in A; note that $d \leq n$. Also, since there is a trivial solution of size n for any set cover instance, n/y^* is a simple upper bound on the approximation ratio. Our bound is a further improvement—it is easily seen that $n/y^* \leq d$ always and that there is a constant $\ell > 0$ such that for every nondecreasing function $f(n)$ with $1 \leq f(n) \leq \ell \ln n / \ln\ln n$, there exist families of (A, b, c) such that

$$\ln(n/y^*) \leq \min\{n/y^*, \ln d\}/f(n).$$

Thus our bound is never more than a multiplicative $(1 + o(1))$ factor above the classical bound and is usually much better; in the best case, our improvement is by $\Theta(\log n / \log\log n)$. (We can construct, e.g., instances with $d = n^{\Theta(1)}$ and $y^* = n/\log^{\Theta(1)} n$, giving a $\Theta(\log n/\log\log n)$ improvement.)

Another noteworthy class of CIPs is related to the B-domination problem: given a (directed) graph G with n vertices, we want to place a minimum number of facilities on the nodes such that every node has at least B facilities in its out-neighborhood. This is also a key subproblem in sharing files in a distributed system [26]; under the assumption that G is undirected and letting Δ be its maximum degree, an

$$1 + \mathrm{O}(\max\{\ln(\Delta)/B, \sqrt{\ln(\Delta)/B}\})$$

approximation bound is presented in [26], improving on the standard analysis of randomized rounding. Bound (1.6) improves further on this; in particular, even if G is directed with maximum in-degree Δ, (1.6) shows that the Naor–Roth bound holds. Furthermore, the comments regarding the $\Theta(\log n/\log\log n)$ improvement for set cover hold even in the undirected case. All of this, in turn, provides better bounds for the file-sharing problem.

Thus, the two main contributions of this work are as follows. First, we identify a very desirable "correlation" property of all packing and covering integer programs, which enables one to prove, quite easily, improved bounds on the integrality gap for the linear relaxations of these problems. However, as shown in section 4, this is often not constructive, since the probability of randomized rounding resulting in such good approximations can be (and usually is) negligibly small; section 4 shows a simple family of instances where this "success probability" is as small as $\exp(-\Omega(n+m))$. The second contribution is in showing that the structure of PIPs and CIPs presents a suitable pessimistic estimator (see section 2 for the definition), which allows one to obtain such approximations efficiently.

In section 2, we present some basic notions such as large-deviation inequalities, the FKG inequality, and the idea of pessimistic estimators. Section 3 covers PIPs. We devote section 4 to the important problem of finding a maximum independent set problem on graphs by looking at it in the usual way as a PIP. Its analysis shows the strengths and weaknesses of both our approach and related approaches. Section 5 deals with CIPs; a good understanding of section 3 is essential for this section. Section 6 concludes.

2. Preliminaries.

Let r.v. abbreviate "random variable" and for any positive integer k, let $[k]$ denote the set $\{1, 2, \ldots, k\}$. If a universe $N = \{a_1, a_2, \ldots, a_\ell\}$ is understood, then for any $S \subseteq N$, $\chi(S)$ denotes its characteristic vector: $\chi(S) \in \{0,1\}^\ell$ with $\chi(S)_j = 1$ iff $a_j \in S$. For a sequence $s_1, s_2, \ldots$ and any integer $i \geq 1$, $s^{(i)}$ denotes the vector $(s_1, s_2, \ldots, s_i)$. In our usage, $s_1, s_2, \ldots$ could be a sequence of reals or of random variables. For any i, any vector $w \in \{0,1\}^i$, and for $j \in \{0,1\}$, we define $wj \in \{0,1\}^{i+1}$ as $(w_1, w_2, \ldots, w_i, j)$. As usual, e denotes the base of the natural logarithm. We let $\exp(x)$ denote e^x.

Remark. The following is filled with formulae and calculations, many of them routine. The real ideas of this work are contained in Lemmas 3.1, 3.5, and 3.6. The reader might consider skipping the proofs of most of the rest of the lemmas, for the first reading.

We first recall the CH bounds for the tail probabilities of sums of bounded independent random variables (r.v.'s) [7], [19]. Theorem 2.1 presents these tail bounds; see, e.g., Motwani and Raghavan [25] for the proofs.

THEOREM 2.1. *Let $X_1, X_2, \ldots, X_\ell$ be* independent *r.v.'s, each taking values in* $[0,1]$*, with $R = \sum_{i=1}^{\ell} X_i$ and $\mathbf{E}[R] = \mu$. For any $\delta \geq 0$,*

$$\Pr(R \geq \mu(1+\delta)) < \mathbf{E}[(1+\delta)^{R-\mu(1+\delta)}] \leq G(\mu,\delta) \doteq (\exp(\delta)/(1+\delta)^{(1+\delta)})^\mu,$$

and if $0 \leq \delta < 1$*,*

$$\Pr(R \leq \mu(1-\delta)) < \mathbf{E}[(1-\delta)^{R-\mu(1-\delta)}] \leq H(\mu,\delta) \doteq \exp(-\mu\delta^2/2). \qquad \square$$

The following fact is easily seen.

FACT 1.

(a) $G(\mu,\delta) \leq (e/(1+\delta))^{(1+\delta)\mu}$.

(b) $G(\mu,\delta) \leq \exp(-\delta^2\mu/3)$ *if* $\delta \leq 1$.

(c) $G(\mu,\delta) \leq \exp(-(1+\delta)\ln(1+\delta)\mu/4)$ *if* $\delta \geq 1$.

(d) *If* $0 < \mu_1 \leq \mu_2$*, then* $G(\mu_1,\delta) \geq G(\mu_2,\delta)$.

Call a family $\mathcal{F}$ of subsets of a set N *monotone increasing* (respectively, *monotone decreasing*) if $\forall S \subseteq T \subseteq N$, $S \in \mathcal{F}$ implies that $T \in \mathcal{F}$ (respectively, $T \in \mathcal{F}$ implies that $S \in \mathcal{F}$). We next present Theorem 2.2, a special case of the powerful FKG inequality [14], [31]; for a proof, see, e.g., Chapter 6 of [3].

THEOREM 2.2. *Given a set $N = \{a_1, a_2, \ldots, a_\ell\}$ and some $p = (p_1, p_2, \ldots, p_\ell) \in [0,1]^\ell$, suppose we pick a random $Y \subseteq N$ by placing each a_i in Y* independently*, with probability p_i. For any $\mathcal{F} \subseteq 2^N$, let $Pr_p(\mathcal{F}) \doteq \Pr(Y \in \mathcal{F})$. Let $F_1, F_2, \ldots, F_s \subseteq 2^N$*

be any sequence of monotone increasing families, and let $G_1, G_2, \ldots, G_s \subseteq 2^N$ be any sequence of monotone decreasing families. Then,

$$Pr_p\left(\bigwedge_{i=1}^{s} F_i\right) \geq \prod_{i=1}^{s} Pr_p(F_i) \quad \textit{and} \quad Pr_p\left(\bigwedge_{i=1}^{s} G_i\right) \geq \prod_{i=1}^{s} Pr_p(G_i).$$

Finally, we recall the notion of *pessimistic estimators* [28]. For our purposes, we focus on the case of independent binary r.v.'s. Let $X_1, X_2, \ldots, X_\ell \in \{0,1\}$ be independent r.v.'s with $\Pr(X_i = 1) = p_i$, for some $p \in [0,1]^\ell$. Suppose, for some implicitly defined $L \subseteq \{0,1\}^\ell$, that $\Pr(X^{(\ell)} \in L) < 1$. Our goal is to find some $v \in \{0,1\}^\ell - L$ efficiently. Theorem 2.4 presents the idea of pessimistic estimators applied to the method of conditional probabilities as a means to achieve this goal. See [28] for a detailed discussion and proof. We define as follows.

Definition 2.3. *A function $U : [0,1]^\ell \to \Re^+$ is a pessimistic estimator with respect to $(X_1, \ldots, X_\ell)$ and L if*

(1) *$U(p_1, p_2, \ldots, p_\ell) < 1$, and*

(2) *$\forall i \in (\{0\} \cup [\ell])\ \forall w \in \{0,1\}^i$,*

  (a) *$U(w_1, \ldots, w_i, p_{i+1}, \ldots, p_\ell) \geq \Pr(X^{(\ell)} \in L | X^{(i)} = w)$, and*

  (b) *if $i \leq \ell - 1$, then $U(w_1, \ldots, w_i, p_{i+1}, \ldots, p_\ell)$ is at least as large as*

$$\min\{U(w_1, \ldots, w_i, 0, p_{i+2}, \ldots, p_\ell), U(w_1, \ldots, w_i, 1, p_{i+2}, \ldots, p_\ell)\}.$$

Theorem 2.4 (see [28]). *Let U be a pessimistic estimator with respect to $(X_1, \ldots, X_\ell)$ and L. The following algorithm produces a vector $v \notin L$:*

```
For $i := 0$ to $\ell - 1$ do
  if $U(v_1, \ldots, v_i, 0, p_{i+2}, \ldots, p_\ell) \leq U(v_1, \ldots, v_i, 1, p_{i+2}, \ldots, p_\ell)$, then $v_{i+1} := 0$, else $v_{i+1} := 1$.
```

Proof. It is not hard to see by induction on i that $\forall i \in \{0\} \cup [\ell]$, $\Pr(X^{(\ell)} \in L | X^{(i)} = v^{(i)}) < 1$. Using this for $i = \ell$ in conjunction with property 2(a) of Definition 2.3 completes the proof. ◻

The efficiency of the algorithm depends on how efficiently we can compute U. Also note that for the proof of Theorem 2.4, it suffices if property 2(a) of Definition 2.3 holds just for $i = \ell$.

3. Approximating PIPs.

Let a PIP be given, conforming to Definition 1.1. We assume that $x \in Z_+^m$ is the constraint on x. (Clearly, even if we have constraints such as $x_i \in \{0, 1, \ldots, d_i\}$, we will get identical bounds since scaling down by $\alpha > 1$ and then performing a randomized rounding cannot make $x_i \notin \{0, 1, \ldots, d_i\}$.) Lemmas 3.1 and 3.6 are crucial, wherein the structure of PIPs is exploited. It is essential to read this section before reading section 5—some proofs are omitted in section 5 since they are very similar to the ones in this section.

We solve the LP relaxation and let the scaling by α, events $E_1, E_2, \ldots, E_{n+1}$, and vectors z, x', etc., be as in section 1.1; α and β will be determined later. The main point of this section is to show that good integral solutions exist and to present a candidate for a pessimistic estimator (see (3.3)). We show that this estimator satisfies the conditions of Definition 2.3, and we invoke Theorem 2.4 to show that we can constructivize the existence proof for the improved integrality gap. The work of this section culminates in Theorem 3.7.

We first set up some notation to formulate our "failure probability." For every $j \in [m]$, let $s_j = \lfloor x'_j \rfloor$ and $p_j = x'_j - s_j \in [0,1)$. Let A_i denote the ith row of A. Let

$X_1, X_2, \ldots, X_m \in \{0,1\}$ be *independent* r.v.'s with $\Pr(X_j = 1) = p_j \; \forall j \in [m]$, and let $X \doteq X^{(m)}$. It is clear that

$$E_i \equiv \text{“}A_i \cdot X > \mu_i(1+\delta_i)\text{”} \;\forall i \in [n] \quad \text{and that} \quad E_{n+1} \equiv \text{“}c^T \cdot X < \mu_{n+1}(1-\delta_{n+1})\text{”},$$

where $\mu_i = \mathbf{E}[A_i \cdot X]$ and

$$\delta_i = (b_i - A_i \cdot s)/\mu_i - 1$$

for $i \in [n]$, $\mu_{n+1} = \mathbf{E}[c^T \cdot X]$, and

$$\delta_{n+1} = 1 - (y^*/(\alpha\beta) - \; c^T \cdot s)/\mu_{n+1}.$$

It is readily verified that $\delta_i \geq 0 \; \forall i \in [n]$ and that $\delta_{n+1} \geq 0$. Also, since $c^T \cdot X \geq 0$ with probability 1, we may assume that β is not so large that $\delta_{n+1} \geq 1$: if $\delta_{n+1} \geq 1$, then $\Pr(E_{n+1}) = \Pr(c^T \cdot X < \mu_{n+1}(1-\delta_{n+1})) = 0$, and the bad event E_{n+1} will never happen. Thus, we may assume that $0 \leq \delta_{n+1} < 1$.

Our first objective is to prove (1.4) and hence (1.5) using Theorem 2.2; this will also suggest a choice for a pessimistic estimator. In the notation of Theorem 2.2, $N = [m]$ and $Y = \{i \in N : X_i = 1\}$. For each $i \in [n]$, define $F_i \subseteq 2^N$ as

$$\{S \subseteq N : (A_i \cdot \chi(S)) \leq \mu_i(1+\delta_i)\}.$$

It is easy to see that *each* F_i *is monotone decreasing.* Since $\overline{E_i} \equiv (Y \in F_i)$ for each i, we deduce (1.4) from Theorem 2.2. In fact, a similar proof shows that since the components of X are picked *independently*, we have the following lemma.

LEMMA 3.1. *For any* $j \in \{0\} \cup [m]$ *and any* $w \in \{0,1\}^j$,

$$\Pr\left(\bigvee_{i=1}^{n+1} E_i | X^{(j)} = w\right) \leq 1 - \left(\prod_{i=1}^{n}(1 - \Pr(E_i|X^{(j)} = w))\right) + \Pr(E_{n+1}|X^{(j)} = w). \qquad \square$$

Let

$$F_{n+1} = \{S \subseteq [m] : c^T \cdot \chi(S) \geq \mu_{n+1}(1-\delta_{n+1})\}.$$

In the notation of Definition 2.3, the set to be avoided, L, is

$$\{x \in \{0,1\}^m : \exists i \in [n+1] \;\; \chi^{-1}(x) \notin F_i\}.$$

We next upper-bound $\Pr(E_i)$ for each i. Recall, by Theorem 2.1, that for each $i \in [n]$, $\Pr(E_i) \leq G(\mu_i, \delta_i)$; also, $\Pr(E_{n+1}) \leq H(\mu_{n+1}, \delta_{n+1})$. Lemma 3.2 upper-bounds these quantities.

LEMMA 3.2. (a) *For every* $i \in [n]$, $G(\mu_i, \delta_i) \leq G(b_i/\alpha, \alpha - 1) \leq G(B/\alpha, \alpha - 1)$.
(b) $H(\mu_{n+1}, \delta_{n+1}) \leq H(y^*/\alpha, 1 - 1/\beta)$.

Proof. (a) Note that $\mu_i + A_i \cdot s \leq b_i/\alpha$, with $\mu_i, A_i \cdot s \geq 0$. Subject to these constraints and that $\alpha > 1$, we will show that $G(\mu_i, \delta_i)$ is maximized when $\mu_i = b_i/\alpha$ and $A_i \cdot s = 0$; this will prove (a). Now,

$$G(\mu_i, \delta_i) = \mu_i^{b_i - A_i \cdot s} \exp(-\mu_i)(e/(b_i - A_i \cdot s))^{b_i - A_i \cdot s}. \tag{3.1}$$

If $A_i \cdot s$ is held fixed at some $\gamma \geq 0$, (3.1) is maximized at $\mu_i = \Delta \doteq b_i/\alpha - \gamma$, under the constraint that $\mu_i \in [0, \Delta]$. Thus,

$$G(\mu_i, \delta_i) \leq \exp(b_i - b_i/\alpha)((b_i/\alpha - A_i \cdot s)/(b_i - A_i \cdot s))^{b_i - A_i \cdot s},$$

which is readily shown to be maximized when $A_i \cdot s = 0$. A similar proof holds for (b). ∎

Now that we have good tail bounds, we set $\alpha, \beta > 1$ such that $(\alpha\beta)$ is small and such that for the PIP, $1 - (\prod_{i=1}^{n}(1 - \Pr(E_i))) + \Pr(E_{n+1}) < 1$ holds. This, combined with setting $j = 0$ in Lemma 3.1, will show that there exists a feasible integral solution with objective function value at least $y^*/(\alpha\beta)$. Observe that the bound of Lemma 3.3 makes sense only if $B > 1$. Lemma 3.4 handles the common case where $A_{i,j} \in \{0, 1\}$ $\forall i, j$, to get improved bounds which, in particular, work even if $B = 1$. We have not attempted to optimize the constants.

LEMMA 3.3. *There exist constants $K_1 \geq 3$ and $K_2 \geq 1$ such that for any PIP, if α is taken as $K_1 \cdot \max\{1, (K_2 n/y^*)^{1/(B-1)}\}$ and $\beta = 2$, then $1 - (\prod_{i=1}^{n}(1 - \Pr(E_i))) + \Pr(E_{n+1}) < 1$.*

Proof. Suppose we agree to guarantee

$$\alpha \geq 3 \tag{3.2}$$

and $\beta = 2$. By Lemma 3.2, it suffices to show that $H(y^*/\alpha, 1/2) < (1 - G(B/\alpha, \alpha - 1))^n$. Furthermore, Fact 1(a) shows that

$$\exp(-y^*/(8\alpha)) < (1 - \exp(-B(\ln \alpha - 1)))^n$$

suffices. Since $B \geq 1$ and $\ln \alpha - 1 \geq \ln 3 - 1 > 0$, there exists a fixed $d > 0$ such that

$$1 - \exp(-B(\ln \alpha - 1)) \geq \exp(-d \exp(-B(\ln \alpha - 1)))$$

and hence, it suffices if $y^*/(8\alpha) > nd \exp(-B(\ln \alpha - 1))$. Simplifying, we see that

$$\alpha \geq K_1 (K_2 n/y^*)^{1/(B-1)}$$

suffices for certain positive absolute constants $K_1 \geq 3$ and $K_2 \geq 1$. Thus, to satisfy the requirement (3.2), it will suffice to take $\alpha = K_1 \cdot \max\{1, (K_2 n/y^*)^{1/(B-1)}\}$. ∎

LEMMA 3.4. *There exists a constant $K_1 \geq 3$ for PIP instances with $A_{i,j} \in \{0, 1\}$ $\forall i, j$, such that if $\alpha = K_1 \cdot \max\{1, (n/y^*)^{1/B}\}$ and $\beta = 2$, then $1 - (\prod_{i=1}^{n}(1 - \Pr(E_i))) + \Pr(E_{n+1}) < 1$.*

Proof. Since $A_{i,j} \in \{0, 1\}$, we may assume w.l.o.g. that each b_i is an integer, as mentioned in Definition 1.1. Again, since $A_{i,j} \in \{0, 1\}$, we have, for any $i \in [n]$, $((Az)_i > b_i) \rightarrow (Az)_i \geq b_i + 1$. Hence, B essentially gets replaced by $B + 1$ in Lemma 3.3, leading to the strengthened bounds. (We have eliminated the constant K_2 here, since $K_2^{1/B} \leq K_2$; hence, this term can be absorbed into the constant K_1.) ∎

As remarked in the introduction, it can be seen that the bounds (on the integrality gap $(\alpha\beta)$) of Lemmas 3.3 and 3.4 significantly strengthen the corresponding bounds achievable by the standard analysis of randomized rounding. However, as will be seen in section 4, the probability of outputting such an integral solution through randomized rounding can be as small as $\exp(-\Omega(n + m))$; hence, showing a good bound on the integrality gap alone is not satisfactory from the algorithmic viewpoint. We now show that there is indeed a suitable pessimistic estimator; we first introduce some notation to avoid lengthy formulae.

NOTATION 1. *$\forall i \in [n]$, $j \in \{0\} \cup [m]$, and $w \in \{0, 1\}^j$, let*

$$h_i(j, w) \doteq \mathbf{E}[(1 + \delta_i)^{A_i \cdot X - \mu_i(1+\delta_i)} | X^{(j)} = w],$$

$$f_i(j,w) \doteq \mathbf{E}[(1+\delta_i)^{A_i \cdot X - \mu_i(1+\delta_i)} | X^{(j+1)} = w0], \text{ and}$$

$$g_i(j,w) \doteq \mathbf{E}[(1+\delta_i)^{A_i \cdot X - \mu_i(1+\delta_i)} | X^{(j+1)} = w1].$$

When j and w are clear from the context, we may refer to these as h_i, f_i, and g_i.

From Theorem 2.1 and Lemma 3.1, a natural guess for a pessimistic estimator, $U(w_1, \ldots, w_j, p_{j+1}, \ldots, p_m)$ $\forall j \in \{0\} \cup [m]$ $\forall w \in \{0,1\}^j$, might be

$$1 - \left(\prod_{i=1}^{n} (1 - h_i(j,w))\right) + \mathbf{E}[(1-\delta_{n+1})^{c^T \cdot X - \mu_{n+1}(1-\delta_{n+1})} | X^{(j)} = w].$$

This might complicate matters if $h_i(j,w) > 1$, however, so we first define $h'_i(j,w) = \min\{h_i(j,w), 1\}$, $f'_i(j,w) = \min\{f_i(j,w), 1\}$, and $g'_i(j,w) = \min\{g_i(j,w), 1\}$. We now let $U(w_1, \ldots, w_j, p_{j+1}, \ldots, p_m)$ $\forall j \in \{0\} \cup [m]$ $\forall w \in \{0,1\}^j$ be

$$1 - \left(\prod_{i=1}^{n} (1 - h'_i(j,w))\right) + \mathbf{E}[(1-\delta_{n+1})^{c^T \cdot X - \mu_{n+1}(1-\delta_{n+1})} | X^{(j)} = w]. \tag{3.3}$$

At this point, we have exhibited suitable α and β such that our function U satisfies properties (1) and 2(a) of Definition 2.3. We now turn to proving property 2(b), the proof of which is more interesting. Before showing Lemma 3.6, which proves this, we first establish a simple lemma which facilitates the proof of Lemma 3.6.

Lemma 3.5. *$\forall i \in [n]$, $j \in \{0\} \cup [m]$, and $w \in \{0,1\}^j$,*

(i) $0 \le f'_i(j,w) \le g'_i(j,w) \le 1$, *and*

(ii) $h'_i(j,w) \ge (1-p_{j+1}) f'_i(j,w) + p_{j+1} g'_i(j,w)$.

Proof. We drop the parameters j and w for the rest of the proof. Part (i) is easily seen. For part (ii), we first note that

$$0 \le f_i \le g_i \quad \text{and} \quad h_i = (1-p_{j+1}) f_i + p_{j+1} g_i \tag{3.4}$$

by the definition of these quantities. If $h_i < 1$ and $g_i \le 1$, then $f_i < 1$ by (3.4) and hence part (ii) above follows from (3.4), with equality. If $h_i < 1$ and $g_i > 1$, note again that $f_i < 1$ and furthermore that $g_i > g'_i = 1$; thus, part (ii) follows from (3.4). Finally if $h_i \ge 1$, note that $h'_i = 1$, $g'_i = 1$ and that $f'_i \le 1$, implying (ii) again. □

Remark. In many constructions of pessimistic estimators for various analyses, equality holds in part (ii) of Lemma 3.5 (as opposed to our "$\ge$"). This makes it easy to prove that the function on hand is a valid pessimistic estimator. Our task is made more challenging because of this change in our case.

Lemma 3.6. *$\forall j \in \{0\} \cup [m-1]$ $\forall w \in \{0,1\}^j$, $U(w_1, \ldots, w_j, p_{j+1}, p_{j+2}, \ldots, p_m) \ge (1-p_{j+1}) U(w_1, \ldots, w_j, 0, p_{j+2}, \ldots, p_m) + p_{j+1} U(w_1, \ldots, w_j, 1, p_{j+2}, \ldots, p_m)$. Thus, in particular, $U(w_1, \ldots, w_j, p_{j+1}, p_{j+2}, \ldots, p_m)$ is at least as high as*

$$\min\{U(w_1, \ldots, w_j, 0, p_{j+2}, \ldots, p_m), U(w_1, \ldots, w_j, 1, p_{j+2}, \ldots, p_m)\}.$$

Proof. Let $r \doteq p_{j+1}$, for convenience. Note that

$$\mathbf{E}[(1-\delta_{n+1})^{c^T \cdot X - \mu_{n+1}(1-\delta_{n+1})} | X^{(j)} = w]$$

equals the sum of two terms:

$$(1-r)\mathbf{E}[(1-\delta_{n+1})^{c^T \cdot X - \mu_{n+1}(1-\delta_{n+1})} | X^{(j+1)} = w0]$$

and

$$r\mathbf{E}[(1-\delta_{n+1})^{c^T\cdot X-\mu_{n+1}(1-\delta_{n+1})}|X^{(j+1)}=w1].$$

Omitting the parameters j and w in f_i, g_i, etc., it is thus sufficient to show that

$$\prod_{i=1}^{n}(1-h_i') \leq (1-r)\prod_{i=1}^{n}(1-f_i') + r\prod_{i=1}^{n}(1-g_i').$$

Thus from Lemma 3.5(ii) and since $h_i' \leq 1$, it suffices to show that

$$\prod_{i=1}^{n}(1-(1-r)f_i'-rg_i') \leq (1-r)\prod_{i=1}^{n}(1-f_i') + r\prod_{i=1}^{n}(1-g_i'), \tag{3.5}$$

which we now prove by induction on n.

Equality holds in (3.5) for the base case $n=1$. We now prove (3.5) by assuming its analogue for $n-1$, i.e., we show that

$$\left((1-r)\prod_{i=1}^{n-1}(1-f_i') + r\prod_{i=1}^{n-1}(1-g_i')\right)(1-(1-r)f_n'-rg_n') \leq (1-r)\prod_{i=1}^{n}(1-f_i')+r\prod_{i=1}^{n}(1-g_i').$$

Simplifying, we need to show that

$$r(1-r)(g_n'-f_n')\left(\prod_{i=1}^{n-1}(1-f_i') - \prod_{i=1}^{n-1}(1-g_i')\right) \geq 0, \tag{3.6}$$

which holds in view of Lemma 3.5(i). □

Since all the requirements of Definition 2.3 are satisfied, we have the following theorem.

THEOREM 3.7. *There exist constants $K_3, K_4 > 0$ such that given any PIP conforming to the notation of Definition* 1.1, *we can produce, in deterministic polynomial time, a feasible solution to it, of value at least*

$$K_3 \cdot \min\{y^*, (K_4 y^*/n^{1/B})^{B/(B-1)}\}.$$

If $A \in \{0,1\}^{n\times m}$, the guarantee on the solution value is at least

$$K_3 \cdot \min\{y^*, (y^*/n^{1/(B+1)})^{(B+1)/B}\}.$$

Proof. Lemmas 3.3 and 3.4 show property (1) of Definition 2.3. Properties 2(a) and 2(b) of Definition 2.3 are shown by Lemmas 3.1 and 3.6, respectively. Theorem 2.4 now completes the proof. □

4. The maximum independent set problem on graphs.

We consider the classical NP-hard problem of finding a maximum independent set (MIS) in a given undirected graph $G=(V,E)$ and formulate it as a packing problem. Although we do not get improved approximation algorithms for this problem, a few observations on this important problem are relevant, as we shall see shortly.

Turán's classical theorem [39] shows that G always has an independent set of size at least $|V|^2/(2|E|+|V|)$; such a set can also be found in polynomial time. The standard packing formulation described below, combined with our approach, shows the existence of an independent set of size $\Omega(|V|^2/|E|)$. The constant factor hidden in the $\Omega(\cdot)$ is weaker than that of Turán's theorem, however—we present this result to show that our approach proves a few other known results, too, in a unified way. We remark that we do not use the standard notation of graphs having n vertices and m edges, as it will go against our notation for PIPs and CIPs; the packing formulation has $|E|$ constraints and $|V|$ variables.

Define an indicator variable $x_i \in \{0,1\}$ for each vertex i, for the presence of vertex i in the independent set (IS). Subject to the constraint that $x_i + x_j \leq 1$ for every edge (i,j), we want to maximize $\sum_i x_i$. For specific problems like this, we can get better bounds than does the analysis for Theorem 3.7, which uses the general CH bounds. The fractional solution $x_i^* = 1/2$ for each i is optimal to within a factor of 2. Suppose we scale x^* down by some $\alpha > 1$ and do the randomized rounding as before. Then for any given edge (i,j), $\Pr(z_i + z_j > 1) \leq 1/(4\alpha^2)$, a bound much better than the CH bound. Analysis as above then shows that $\alpha = \Theta(|E|/|V|)$ and $\beta = \Theta(1)$ suffice, thus producing an IS of size $\Omega(y^*/(\alpha\beta)) = \Omega(|V|^2/|E|)$.

One reason for considering the MIS problem is to show that the failure probability given by (1.5) can be extremely close to (although strictly smaller than) 1. This would underscore the importance of the fact that a pessimistic estimator can be constructed for PIPs and CIPs. Suppose the graph $G = (V,E)$ is a line on the N vertices $1, 2, \ldots, N$ and that each vertex independently picks a random bit for itself with the bit being one with probability q, for some $q \in [0,1]$. Let p_N be the probability that no two adjacent vertices choose the bit 1. Setting $q = 1/(2\alpha) = \Theta(|V|/|E|) = \Theta(1)$ above, it is then clear that the probability that randomized rounding (with the above values for α and β) picks a valid IS in G equals p_N. We now proceed to show that p_N is exponentially small in N, validating our point.

Computing p_N by induction on N is standard. Let a_N (respectively, b_N) denote the probability that not only do no pair of adjacent vertices both choose 1 but also that vertex N chooses the bit 1 (respectively, 0). Note that $p_N = a_N + b_N$. The recurrences

$$\begin{aligned} a_{N+1} &= q b_N, \\ b_{N+1} &= (1-q)(a_N + b_N) \end{aligned}$$

are immediate. Letting $\nu \doteq \sqrt{(1-q)(1+3q)}$, it can then be seen that

$$a_N = \frac{q}{\nu}\left(\left(\frac{1-q+\nu}{2}\right)^N - \left(\frac{1-q-\nu}{2}\right)^N\right).$$

Using the facts $b_N = a_{N+1}/q$ and $p_N = a_N + b_N$, we then see that $p_N = \exp(-\Omega(N))$, i.e., extremely small. Thus, the success probability of randomized rounding with our chosen values for α and β can be (and usually is) extremely small, motivating the need for a good pessimistic estimator.

The MIS problem also illustrates the well-known fact that linear relaxations are not always tight. As seen above, this problem always has a fractional solution lying between $|V|/2$ and $|V|$. However, the graph G can have its independence number to be any integer in $[|V|]$ and hence, the integrality gap of this LP formulation can be quite bad. Furthermore, recent work of Håstad, building on earlier research, has shown that the MIS cannot be approximated in polynomial time to within any factor better than $|V|^{1-\epsilon}$ for any fixed $\epsilon > 0$, unless some unexpected containment results hold in computational complexity theory [17]. This shows that we cannot expect even reasonably good approximation algorithms for *all* PIPs.

5. Approximating CIPs. Given a CIP conforming to Definition 1.1, we show how to get a good approximation algorithm for it. Since the ideas here are very similar to those of section 3, we borrow a lot of notation from there, skim over most details, and present only the essential differences.

The idea here is to solve the LP relaxation and, for an $\alpha > 1$ to be fixed later, to set $x'_j = \alpha x^*_j$ for each $j \in [m]$. We then construct a random integral solution z by setting, *independently* for each $j \in [m]$, $z_j = \lfloor x'_j \rfloor + 1$ with probability $x'_j - \lfloor x'_j \rfloor$ and $z_j = \lfloor x'_j \rfloor$ with probability $1 - (x'_j - \lfloor x'_j \rfloor)$. Let A_i, s_i, and $X_1, X_2, \ldots, X_m$ be as in section 3. The bad events now are

$$E_i \equiv \text{“}A_i \cdot X < \mu_i(1-\delta_i)\text{”} \ \forall i \in [n] \text{ and}$$

$$E_{n+1} \equiv \text{“}c^T \cdot X > \mu_{n+1}(1+\delta_{n+1})\text{”},$$

where $\mu_i = \mathbf{E}[A_i \cdot X]$ and

$$\delta_i = 1 - (b_i - A_i \cdot s)/\mu_i$$

for $i \in [n]$, $\mu_{n+1} = \mathbf{E}[c^T \cdot X]$, and

$$\delta_{n+1} = (y^* \alpha\beta - \ c^T \cdot s)/\mu_{n+1} - 1.$$

Analogously to PIPs, $0 \le \delta_i < 1 \ \forall i \in [n]$ and $\delta_{n+1} \ge 0$.

For any $i \in [n]$, let

$$F_i = \{S \subseteq [m] : A_i \cdot \chi(S) \ge \mu_i(1-\delta_i)\}.$$

Each of these families is *monotone increasing* now, and thus Theorem 2.2 again guarantees Lemma 3.1, for the present definition of $E_1, E_2, \ldots, E_{n+1}$ also.

Suppose given some $j \in \{0\} \cup [m-1]$ and some $w \in \{0,1\}^j$ we define $h'_i(j,w)$, $f'_i(j,w)$, and $g'_i(j,w)$ for every $i \in [n]$ analogously as in Notation 1:

$$h'_i(j,w) \doteq \min\{1, \mathbf{E}[(1-\delta_i)^{A_i \cdot X - \mu_i(1-\delta_i)} | X^{(j)} = w]\},$$

$$f'_i(j,w) \doteq \min\{1, \mathbf{E}[(1-\delta_i)^{A_i \cdot X - \mu_i(1-\delta_i)} | X^{(j+1)} = w0]\}, \text{ and}$$

$$g'_i(j,w) \doteq \min\{1, \mathbf{E}[(1-\delta_i)^{A_i \cdot X - \mu_i(1-\delta_i)} | X^{(j+1)} = w1]\}.$$

As can be expected, the pessimistic estimator $U(w_1, \ldots, w_j, p_{j+1}, \ldots, p_m) \ \forall j \in \{0\} \cup [m] \ \forall w \in \{0,1\}^j$ is now

$$1 - \left(\prod_{i=1}^{n}(1 - h'_i(j,w))\right) + \mathbf{E}[(1+\delta_{n+1})^{c^T \cdot X - \mu_{n+1}(1+\delta_{n+1})} | X^{(j)} = w]. \tag{5.1}$$

Now we address the analogue of the important Lemma 3.6. It is easily checked that Lemma 3.5(ii) holds again and that instead of part (i) of Lemma 3.5, we have

$$0 \le g'_i \le f'_i \le 1. \tag{5.2}$$

Thus, (5.2) guarantees (3.6) even now! This shows that Lemma 3.6 holds for the current definition of U also.

Thus to establish that U is a pessimistic estimator, we only have to exhibit, as do Lemmas 3.3 and 3.4, $\alpha, \beta > 1$, which ensure that $U(p_1, \ldots, p_m) < 1$. We first present a lemma similar to Lemma 3.2, whose simple proof is omitted.

LEMMA 5.1. *For all $i \in [n]$, $\Pr(E_i) \leq H(B\alpha, 1 - 1/\alpha)$. Also, $\Pr(E_{n+1}) \leq G(y^*\alpha, \beta - 1)$.*

We now present the main theorem on covering problems. Since unweighted set cover is an important problem, we present the precise approximation bound for this problem as a distinct part of the theorem. For this problem, the term "$\ln\ln(n/y^*)$" appears in our approximation bound with the understanding that we are working in the nontrivial range where $y^* \leq n/e$, say. Since there is always a trivial integral solution of value at most n for this problem, the case $y^* > n/e$ admits an e-approximation.

THEOREM 5.2. (a) *Given a CIP conforming to the notation of Definition 1.1, we can produce, in deterministic polynomial time, a feasible solution to it with value at most*

$$y^*(1 + \mathrm{O}(\max\{\ln(nB/y^*)/B, \sqrt{\ln(2\lceil nB/y^*\rceil)/B}\})).$$

(b) *For unweighted set cover, we can improve this to $y^*(\ln(n/y^*) + \ln\ln(n/y^*) + \mathrm{O}(1))$.*

Proof. We start with the simple claim that for $t \in (0, 1/e)$, $1-t > \exp(-1.25t)$. To see this, let $u = e(1-\ln(e-1)) = 1.246\cdots$. Consider the function $t \mapsto 1-t-\exp(-ut)$; we now show that for $t \in (0, 1/e)$, this function is positive. The function vanishes at $t = 0$, and its derivative is $u\exp(-ut) - 1$. So as t increases from 0, the function increases initially and then decreases. Thus since the function vanishes at $t = 1/e$, it is positive for $t \in (0, 1/e)$.

Recall from the above discussion that $(1 - \max_{i\in[n]} \Pr(E_i))^n > G(y^*\alpha, \beta - 1)$ suffices. In all cases here, we will ensure that

$$\max_{i\in[n]} \Pr(E_i) < 1/e \text{ and } 1 < \beta \leq 2. \tag{5.3}$$

Thus, by the above claim and from Fact 1(b), it will suffice to show that

$$1.25n \max_{i\in[n]} \Pr(E_i) \leq y^*\alpha(\beta - 1)^2/3. \tag{5.4}$$

We now choose α and β appropriately for each of our cases.

(a) For general CIPs, we will use the bound $\max_{i\in[n]} \Pr(E_i) \leq H(B\alpha, 1 - 1/\alpha)$ of Lemma 5.1. So, assuming $H(B\alpha, 1 - 1/\alpha) < 1/e$ and $1 < \beta \leq 2$ hold, (5.4) shows that it will be enough to show

$$1.25n \exp(-B\alpha(1 - 1/\alpha)^2/2) \leq y^*\alpha(\beta - 1)^2/3. \tag{5.5}$$

For general CIPs, we will consider three cases: $\ln(nB/y^*) > B$, $3 \leq \ln(nB/y^*) \leq B$, and $\ln(nB/y^*) < 3$. (The first and third cases are not necessarily mutually exclusive.)

Suppose $\ln(nB/y^*) > B$. Since $B \geq 1$, this shows that $nB/y^* \geq e$. We take $\alpha = 4\ln(nB/y^*)/B$ and $\beta = 2$. Since $\alpha \geq 4$, $(1 - 1/\alpha) \geq 3/4$; thus it is easy to check that $H(B\alpha, 1 - 1/\alpha) < 1/e$. Therefore to prove (5.5), it is enough to show

$$1.25n(y^*/(nB))^{9/8} \leq 4y^*\ln(nB/y^*)/(3B), \quad \text{i.e.,} \quad 1.25(y^*/(nB))^{1/8} \leq 4\ln(nB/y^*)/3,$$

which is true since $nB/y^* \geq e$. Thus, the approximation bound $\alpha\beta$ in this case is $\mathrm{O}(\ln(nB/y^*)/B)$.

Next, suppose $3 \leq \ln(nB/y^*) \leq B$. Thus $nB/y^* \geq e^3$. We choose $\alpha = 1 + 3\sqrt{\ln(nB/y^*)/B}$ and $\beta = 1 + \sqrt{\ln(nB/y^*)/B} \leq 2$. We have

$$\begin{aligned}\exp(-B\alpha(1-1/\alpha)^2/2) &= \exp(-B(\alpha-1)^2/(2\alpha)) \\ &= \exp(-9\ln(nB/y^*)/(2\alpha)) \\ &\leq \exp(-9\ln(nB/y^*)/8),\end{aligned}$$

since $\alpha \leq 4$. We see that $\exp(-B\alpha(1-1/\alpha)^2/2) < 1/e$. Hence by (5.5), it is enough to show

$$1.25n(y^*/(nB))^{9/8} \leq y^*\alpha\ln(nB/y^*)/(3B),$$

which follows from the facts that $\alpha \geq 1$ and $nB/y^* \geq e^3$. The approximation bound $\alpha\beta$ is $1 + \mathrm{O}(\sqrt{\ln(nB/y^*)/B})$ here.

Now consider the third case $\ln(nB/y^*) < 3$. Choose $\alpha = 1 + 10/\sqrt{B}$ and $\beta = 1 + 1/\sqrt{B} \leq 2$. Now,

$$\exp(-B\alpha(1-1/\alpha)^2/2) = \exp(-50/\alpha) \leq \exp(-50/11),$$

since $\alpha \leq 11$. So $\exp(-B\alpha(1-1/\alpha)^2/2) < 1/e$. Hence, again by (5.5), it suffices to show that

$$1.25n\exp(-50/11) \leq y^*\alpha/(3B),$$

which follows from the facts that (i) $\alpha \geq 1$ and (ii) $nB/y^* \leq \exp(3) \leq \exp(50/11)/3.75$. Therefore $\alpha\beta = 1 + \mathrm{O}(1/\sqrt{B})$ in this case.

(b) Thus we have proved the theorem for general CIPs. For the important unweighted set cover problem (see section 1.3 for the definition), we observe that for any $i \in [n]$, E_i holds iff $A_i \cdot s = A_i \cdot X = 0$; this makes the calculations easier. If A_i has j nonzeroes (ones) in it, say in columns $\ell_1, \ell_2, \ldots, \ell_j$, then it is not hard to see that $\Pr(A_i \cdot X = 0)$ is maximized when $x'_{\ell_k} = \alpha/j \ \forall k \in [j]$. Thus, $\Pr(E_i) \leq (1-\alpha/j)^j < \exp(-\alpha)$ and hence, by (5.4), it suffices to pick $\alpha > 1$ and $1 < \beta \leq 2$ such that

$$1.25n\exp(-\alpha) \leq y^*\alpha(\beta-1)^2/3. \tag{5.6}$$

It can now be verified that by choosing

$$\alpha = \ln(n/y^*) + \ln\ln(n/y^*) + \mathrm{O}(1) \text{ and } \beta = 1 + (\ln(n/y^*))^{-1},$$

we will satisfy (5.6). Hence, the approximation guarantee $\alpha\beta$ can be made as small as $\ln(n/y^*) + \ln\ln(n/y^*) + \mathrm{O}(1)$. $\square$

It is also worth looking at some concrete improvements over existing algorithms brought about by Theorem 5.2. In the case of unweighted set cover, suppose $d \leq n$ is the maximum column sum—the maximum cardinality of any edge in the given hypergraph. Then, by summing up all the constraints, we can see that

$$y^*d \geq n. \tag{5.7}$$

Thus, our approximation bound for the set cover problem (see the second statement of Theorem 5.2) is never more by a multiplicative $(1 + o(1))$ factor above the classical bound of $\min\{n/y^*, \ln d + \mathrm{O}(1)\}$. On the other hand, $n/y^* \ll d$ is quite likely, and it is easy to construct set cover instances with

$$\min\{n/y^*, \ln d\} = \Theta(\log n/\log\log n)\ln(n/y^*).$$

For instance, we can arrange for just a few edges to have the maximum edge size of $n^{\Theta(1)}$, while keeping y^* as high as $n/\log^{\Theta(1)} n$. Thus in the best case, we get a $\Theta(\log n/\log\log n)$ factor improvement in the approximation ratio.

An important case of the unweighted set cover problem is the *dominating set* problem: given a (directed) graph G, the problem is to pick a minimum number of vertices such that for every one vertex v, at least one vertex in $v \cup \mathrm{Out}(v)$ is picked, where $\mathrm{Out}(v)$ denotes the out-neighborhood of v.

We next consider a more general domination-type problem on graphs, modeling a class of location problems. Given a (directed) graph G with n nodes and some integral parameter $B \geq 1$, we have to place the smallest possible number of facilities on the nodes of G, so that every node u has at least B facilities in $u \cup \mathrm{Out}(u)$; multiple facilities at the same node are allowed. Let us call this the B-domination problem on G.

For the case where G is *undirected* with maximum degree Δ, an approximation bound of $1 + \mathrm{O}(\max\{\ln(\Delta)/B, \sqrt{\ln(\Delta)/B}\})$ is presented in [26], improving on the $1+\mathrm{O}(\max\{\ln(n)/B, \sqrt{\ln(n)/B}\})$ bound given by the standard analysis of randomized rounding. Our Theorem 5.2 gives a bound of

$$1 + \mathrm{O}(\max\{\ln(nB/y^*)/B, \sqrt{\ln(2\lceil nB/y^*\rceil)/B}\}).$$

Even if G is directed, this new bound is as good as or better than

$$1 + \mathrm{O}(\max\{\ln(\Delta_{in})/B, \sqrt{\ln(\Delta_{in})/B}\}),$$

where Δ_{in} denotes the maximum in-degree of G; this is easily seen from the fact that $y^* \geq nB/\Delta_{in}$, which follows from the same reasoning as for (5.7). We thus get a generalization of the Naor–Roth result. In the case of undirected graphs, it is not hard to show families of graphs for which the present bound is better than that of Naor and Roth's by a factor of up to $\Theta(\log n/\log\log n)$.

In addition to its independent interest, the above B-domination problem is a crucial subproblem in the following file-sharing problem in distributed networks [26]. Given an undirected graph $G = (V, E)$ with n vertices and maximum degree Δ and a file F of k bits, F must be stored in some way at the nodes of G, such that every node can recover F by examining the contents of its and its neighbor's memories; the aim is to minimize the total amount of memory used. (Note that solving the above domination problem is not sufficient for this task.) Letting y^* be the optimum of the (obvious) linear relaxation of the k-domination problem on G, a memory allocation that uses a total of

$$1 + \mathrm{O}(\max\{\ln(\Delta)/k, \sqrt{\ln(\Delta)/k}\}) \tag{5.8}$$

bits is presented in [26]. (Note that y^* is a lower bound on the total amount of memory needed.) Briefly, the approach of [26] works as follows. Let $B = k + 3\lceil\log_2 \Delta\rceil + 1$. Given a feasible integral solution (vector) $x = [x_v]_{v \in V}$ to the B-domination problem on G, the work of [26] shows how to come up with a feasible memory allocation for

F, using a total of $t = \sum_{v \in V} x_v$ bits. Recall that our bound for CIPs shows a way of achieving

$$t = y^*(B/k)(1 + \mathrm{O}(\max\{\ln(nB/y^*)/B, \sqrt{\ln(2\lceil nB/y^* \rceil)/B}\})),$$

which equals

$$y^*(1 + \mathrm{O}(\max\{\ln(\Delta)/k, \sqrt{\ln(2\lceil nk/y^* \rceil)/k}\})).$$

This is always as good as (5.8) and better if $k \gg \ln(\Delta)$.

5.1. Applications to a facility location problem. We now consider another type of canonical facility location problem: *uncapacitated facility location*. Given a directed graph $G = (V, E)$ with nonnegative vertex-costs $\{d(j)\}$ and nonnegative edge-lengths $\{c(i,j)\}$, let $\mathrm{Out}(v)$ denote the out-neighborhood of vertex v. Suppose it costs $d(j)$ units to locate a facility at vertex j. The problem is to place facilities on the nodes of G such that for every vertex v, at least one vertex in $\mathrm{Out}(v)$ houses a facility; the aim is to minimize the sum of the total weight of the sites of the facilities and the total distance (in terms of edge-lengths) traveled by all the vertices to their closest facilities. By scaling, we assume w.l.o.g. that all the $d(j)$ and $c(i,j)$ lie in $[0,1]$. $\mathrm{O}(\log n)$ approximation bounds are known for this problem (Hochbaum [18], [5]). There is a known LP relaxation for the problem to be described below. Letting $n = |V|$ and y^* be the optimal value of this relaxation, we now show how to obtain an integral solution of value $\mathrm{O}(y^*(1 + \log(\lceil n/y^* \rceil)))$.

We start with a known ILP formulation of the problem (Kolen and Tamir [22]). Let $\Delta(i) \doteq |\mathrm{Out}(i)|$. For each $i \in V$, sort the set $\{c(i,j) : j \in \mathrm{Out}(i)\}$ in nondecreasing order to obtain a sequence $c'(i,1), c'(i,2), \ldots, c'(i,\Delta(i))$. Following [22], consider the following variables. For all $i \in V$ and all $j \leq \Delta(i) - 1$, let

- $x_i \in \{0,1\}$ be the indicator variable for a facility being placed at i, and
- $z_{i,j} \in \{0,1\}$ be the indicator variable for no facility being at a distance of at most $c'(i,j)$ from i.

Then it is not hard to see that the following ILP formulation of [22] is valid:

$$\text{minimize } \sum_{j \in V} d_j x_j + \sum_{i \in V} \left(c'(i,1) + \sum_{j=1}^{\Delta(i)-1} (c'(i,j+1) - c'(i,j)) z_{i,j} \right) \text{ subject to}$$

$$\forall i \in V\ \forall k \leq \Delta(i) - 1, \quad z_{i,k} + \sum_{j:\ (i,j) \in E,\ c(i,j) \leq c'(i,k)} x_j \geq 1; \tag{5.9}$$

$$\forall i \in V, \quad \sum_{j:\ (i,j) \in E} x_j \geq 1; \tag{5.10}$$

$$\forall i \in V\ \forall k \leq \Delta(i) - 1, \quad x_i \in \{0,1\}, \qquad z_{i,k} \in \{0,1\}. \tag{5.11}$$

Relaxing each x_i and $z_{i,j}$ to be a real in $[0,1]$, we get an LP relaxation. Let y^* denote the optimum value of this relaxation and $\{x^*, z^*\}$ be an optimal solution to the relaxation. We now show how to round this to a feasible solution for the above ILP. Let $m = |E|$. Note that a direct application of our covering results will lead to an integral solution of value $\mathrm{O}(y^*(1 + \log(\lceil m/y^* \rceil)))$, since there are $\mathrm{O}(m)$ constraints above. We now improve this to $\mathrm{O}(y^*(1 + \log(\lceil n/y^* \rceil)))$.

For each (i,j), set $z'_{i,j} := 0$ if $z^*_{i,j} < 1/2$ and $z'_{i,j} := 1$ if $z^*_{i,j} \geq 1/2$. For each i, set $x'_i = \min\{2x^*_i, 1\}$. It is not hard to check that this is a feasible fractional solution to the LP with objective function value at most $2y^*$; crucially, we have rounded all the $z_{i,j}$ to 0 and 1. Let us see how to round the vector x' to a $\{0,1\}$-vector. Fix $i \in V$. It is easily seen that $z^*_{i,1} \geq z^*_{i,2} \geq \cdots$. Let $k(i)$ be the maximum k for which $z^*_{i,k} \geq 1/2$ (if no such k exists, $k(i) = 0$); for all $k \leq k(i)$, (5.9) will be satisfied irrespective of our rounding of x', since $z'_{i,k} = 1$. It is not hard to check that to satisfy (5.9) and (5.10), it suffices to have

$$\sum_{j:\ (i,j)\in E,\ c(i,j)\leq c'(i,k(i)+1)} x_j \geq 1.$$

Thus, we are left with at most one constraint per vertex i. Moreover, the fractional vector x' satisfies these constraints as seen above. Thus, we can invoke our approach for CIPs to efficiently construct a feasible integral solution of value $\mathrm{O}(y^*(1 + \log(\lceil n/y^* \rceil)))$.

The reader is referred to [18], [5] for other interesting approaches to the problem. We also mention that for the important special case where the distances $c(i,j)$ are symmetric (i.e., $c(i,j) = c(j,i)$) and satisfy the triangle inequality, work of Shmoys, Tardos, and Aardal [33] and Guha and Khuller [16] has led to good *constant-factor* approximations for this problem.

6. Concluding remarks.

We have presented a simple but very useful property of all packing and covering integer programs—positive correlation. This naturally suggests a better way of analyzing the performance of randomized rounding on PIPs and CIPs. However, the provable probability of success—of satisfying all the constraints *and* delivering a very good approximation—can be extremely low; therefore, in itself, this approach may just prove an existential result. Fortunately, the structure of PIPs and CIPs in fact suggests a pessimistic estimator, thus converting this existence proof into a (deterministic) polynomial-time algorithm. In our view, this is very interesting and gives evidence of the utility of derandomization techniques. A common objection to derandomization is that often it converts a fast randomized algorithm that has a good probability of success to a somewhat slower deterministic algorithm. In our case, the opposite is true! The randomized algorithm suggested by the existence proof has an extremely low probability of success; second, solving the LP relaxation heavily dominates the running time, and the time for running the derandomization is comparatively negligible. (This observation about running the LP relaxation also suggests that, in practice, it would be better to get quickly an approximately optimal solution to the LP relaxation, since we are dealing with approximate solutions anyway.)

The basic idea here has been that by using the correlations, we could improve on the Boole–Bonferroni inequality for our applications. There is much interesting work on approximating the probability of a union using the structure of the underlying events: see, e.g., Hunter [20] and Aldous [2]. In the case of PIPs and CIPs, we have benefited from the fact that the constraints "help each other," by being positively correlated. The precise reasons for such a correlation are spelled out in section 1.2. It is a challenging open question to use the structure of correlations in more complicated scenarios; one such problem is the *set discrepancy problem* [35], [3]. Given a system of n subsets $S_1, S_2, \ldots, S_n$ of a ground set A with n elements, the problem is to come up with a function $\psi : A \rightarrow \{-1, 1\}$, such that the *discrepancy* $\mathrm{disc}(\psi) \doteq \max_{i\in[n]} |\psi(S_i)|$ is small, where $\psi(S_i) = \sum_{j\in S_i} \psi(j)$. While randomized rounding and the method of conditional probabilities can be used to produce a ψ with discrepancy $\mathrm{O}(\sqrt{n \log n})$ [35], [3], a classical *nonconstructive* result of Spencer shows the existence of a ψ with $\mathrm{disc}(\psi) = \mathrm{O}(\sqrt{n})$ [36]. This is best possible, and it is an important open problem to make this constructive. If we write down the natural integer programming formulation for this problem, we can see that each constraint is positively correlated with some subsets of the constraints and negatively correlated with others. (There is the associated observation that in several IPs with both $\leq$ and $\geq$ constraints, the $\leq$ constraints are often positively correlated amongst each other; this is similar for the $\geq$ constraints. This idea potentially could bring improvements in some cases.) It would be very interesting if such more complicated forms of correlation can be used to get a constructive result here.

The set discrepancy problem, an undirected multicommodity flow problem presented in [28], and certain other NP-hard problems in very large scale integration routing, are all modeled by a class of integer programs that are referred to as *minimax integer programs* (MIPs) in recent work of this author [37]. These are different from PIPs and CIPs; an improved integrality gap for sparse instances of such problems is proved in [37].

Yet another potential room for improvement lies in lower-bounding, in the context of (1.4), the ratio

$$\Pr\left(\bigwedge_{i=1}^{n} \overline{E_i}\right) / \prod_{i=1}^{n} \Pr(\overline{E_i}),$$

at least for some particular classes of PIPs/CIPs. We know this ratio to be at least one, by (1.4); a better lower bound (at least for particular problems) will lead to better bounds on the integrality gap. Roughly speaking, such better lower bounds seem plausible especially for PIPs/CIPs wherein several columns have several nonzero entries, i.e., in situations where there is heavy (positive) correlation among the constraints of the IP. This could be a difficult problem, however.

How far can such ideas be pushed? In the general setting of all PIPs and CIPs, not much progress seems to be possible along these lines, as shown in section 4. Also, for some important covering problems such as various "cut" problems on graphs and some network design problems, the number of constraints is *exponential* in the number of variables. Our approach will not give good approximation algorithms for such problems, although some of these problems do admit good approximations via other methods [5]. It would be interesting to apply our approach to get improved approximations for some specific important packing and covering problems; one such recent result is for edge-disjoint paths and related problems [38]. Furthermore, it will be interesting to study the correlations involved in other relaxation approaches such as semidefinite programming relaxations.

For the unweighted set cover problem, recent work of Slavík improves upon our result, showing that a natural greedy algorithm outputs a solution of value at most $(y^* - 1/2)\ln(n/y^*) + y^*$ [34]. Furthermore, improved approximation algorithms for the unweighted set cover problem in many geometric settings have been obtained in [6]. If the dual set-system of a given set cover instance has Vapnik–Chervonenkis (VC) dimension d and if c is the size of the optimal set cover, then the work of [6] presents an algorithm that delivers a solution of value at most $\mathrm{O}(c\log(cd))$. Please refer to [6] for the definitions of dual set systems and VC dimension. We just mention here that $d = \mathrm{O}(1)$ in many geometric settings; thus, the work of [6] leads to an approximation guarantee of $\mathrm{O}(\log c)$ in such cases.

As we had seen before, our bounds are incomparable with known results for some *weighted* CIPs, e.g., those considered in [8], [5]. For instance, our approximation ratio of $O(\ln(n/y^*))$ for the weighted set cover problem is incomparable with the ratio of $O(\ln d)$ obtained in [8], where d denotes the maximum column sum in A; this is because unlike the unweighted case, n/y^* is not necessarily at most d, in the weighted case. Recent work of the author improves on many current results, including some of the results on weighted PIPs and CIPs from the present paper [37]. The work of [37] presents improved approximation guarantees for PIPs, CIPs, and MIPs, if the coefficient matrix A does not have many nonzero entries in any column; in particular, it generalizes the result of [8] shown above.

Appendix. Standard analysis of randomized rounding for PIPs. Choose $\alpha = e(5n)^{1/B}$. We first handle an easy case. If $y^* \leq 3\alpha$, we just choose any j such that $c_j = 1$ and set $x_j = 1$ and $x_k = 0\ \forall k \neq j$. This is clearly an $O(n^{1/B})$-approximation algorithm if $y^* \leq 3\alpha$.

Suppose $y^* > 3\alpha$. Let the bad events $E_1, E_2, \ldots, E_{n+1}$ be as in (1.1) with $\beta = 3$. If we avoid all these $(n+1)$ events, we would have a feasible solution with objective function value at least $y^*/(3\alpha)$. The goal now is to show

$$\sum_{i\in[n+1]} \Pr(E_i) \leq K_4 \tag{A.1}$$

for say some constant $K_4 < 1$.

For the purpose of analysis, view each z_j as a sum of $\lceil x'_j \rceil$ independent $\{0,1\}$ r.v.'s, where the first $\lceil x'_j \rceil - 1$ are 1 with probability 1 and the last is 1 with probability $1 - (\lceil x'_j \rceil - x'_j)$. Recalling that $A_{i,j} \in [0,1]\ \forall i,j$, we see, for each $i \in [n]$, that $(Az)_i$ is a sum of independent r.v.'s, each of which takes on values in $[0,1]$. Also, $\mathbf{E}[(Az)_i] = (Ax')_i/\alpha \leq b_i/\alpha$.

Thus, for $i \in [n]$, $\Pr(E_i)$ can be upper-bounded using part (a) of Fact 1,

$$\Pr((Az)_i > b_i) \leq (e/\alpha)^{b_i} = (5n)^{-b_i/B} \leq 1/(5n), \tag{A.2}$$

since $b_i \geq B$. To upper-bound $\Pr(E_{n+1})$, even Chebyshev's inequality will suffice here. If Y is a sum of independent r.v.'s, each taking values in $[0,1]$, it is easy to check that the variance of Y is at most $\mathbf{E}[Y]$. Thus, Chebyshev's inequality shows that for such a Y and any $a > 0$, $\Pr(|Y - \mathbf{E}[Y]| \geq a) \leq \mathbf{E}[Y]/a^2$. Since $y^* > 3\alpha$ now by assumption, we have $\mu \doteq \mathbf{E}[c^T \cdot z] = y^*/\alpha \geq 3$. Thus by Chebyshev's inequality

$$\Pr(c^T \cdot z < \mu/3) \leq \Pr(|c^T \cdot z - \mu| > 2\mu/3] \leq 9/(4\mu) \leq 3/4. \tag{A.3}$$

Bounds (A.2) and (A.3) show that

$$\sum_{i\in[n+1]} \Pr[E_i] \leq 1/5 + 3/4 = 0.95;$$

by repeating this algorithm an appropriate constant number of times, the probability of failure can be reduced to any desired positive constant.

Next suppose $A \in \{0,1\}^{n\times m}$. Recall from Definition 1.1 that we take each b_i to be an integer here. Thus, $E_i \equiv ((Az)_i > b_i)$ is equivalent to $((Az)_i \geq b_i + 1)$ now; thus B essentially gets replaced by $B+1$ in (A.2). We may thus take $\alpha = \Theta(n^{1/(B+1)})$ and get an $O(\alpha)$ approximation as above.

Acknowledgments. We thank Moni Naor, Babu Narayanan, and David Shmoys for their valuable comments. Thanks in particular to Prabhakar Raghavan for his insightful suggestions and pointers to the literature. We acknowledge the valuable feedback of the STOC 1995 program committee and referees. We also thank the journal referees for their very helpful comments and suggestions, which have substantially improved the quality of this paper.

REFERENCES

[1] R. Aharoni, P. Erdős, and N. Linial, *Optima of dual integer linear programs*, Combinatorica, 8 (1988), pp. 13–20.

[2] D. Aldous, *The Harmonic Mean Formula for probabilities of unions: Applications to sparse random graphs*, Discrete Math., 76 (1989), pp. 167–176.

[3] N. Alon, J. H. Spencer, and P. Erdős, *The Probabilistic Method*, Wiley-Interscience, New York, 1992.

[4] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, *Efficient probabilistically checkable proofs and applications to approximation*, in Proceedings if the ACM Symposium on Theory of Computing, San Diego, CA, 1993, pp. 294–304.

[5] D. Bertsimas and R. Vohra, *Linear programming relaxations, approximation algorithms and randomization; A unified view of covering problems*, Technical Report OR 285–94, Massachusetts Institute of Technology, Cambridge, MA, 1994.

[6] H. Brönnimann and M. T. Goodrich, *Almost optimal set covers in finite VC-dimensions*, Discrete Comput. Geom., 14 (1995), pp. 463–479.

[7] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493–509.

[8] V. Chvátal, *A greedy heuristic for the set covering problem*, Math. Oper. Res., 4 (1979), pp. 233–235.

[9] P. Crescenzi and V. Kann, *A compendium of NP optimization problems*, Technical Report SI/RR-95/02, Department of Computer Science, University of Rome "La Sapienza," 1995.

[10] G. Dobson, *Worst-case analysis of greedy heuristics for integer programming with nonnegative data*, Math. Oper. Res., 7 (1982), pp. 515–531.

[11] J. D. Esary and F. Proschan, *Coherent structures of non-identical components*, Technometrics, 5 (1963), pp. 191–209.

[12] U. Feige, *A threshold of* $\ln n$ *for approximating set cover*, in Proceedings of the ACM Symposium on Theory of Computing, Philadelphia, PA, 1996, pp. 314–318.

[13] M. L. Fisher and L. A. Wolsey, *On the greedy heuristic for continuous covering and packing problems*, SIAM J. Alg. Discrete Methods, 3 (1982), pp. 584–591.

[14] C. M. Fortuin, P. W. Kasteleyn, and J. Ginibre, *Correlational inequalities on some partially ordered sets*, Comm. Math. Phys., 22 (1971), pp. 89–103.

[15] R. L. Graham, *Application of the FKG inequality and its relatives*, in Mathematical Programming: The State of the Art, A. Bachem, M. Grötschel, and B. Korte, eds., Springer-Verlag, Berlin, New York, 1983.

[16] S. Guha and S. Khuller, *Greedy strikes back: Improved facility location algorithms*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 649–657.

[17] J. Håstad, *Clique is hard to approximate within* $n^{1-\epsilon}$, in Proceedings of the IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 627–636.

[18] D. S. Hochbaum, *Heuristics for the fixed cost median problem*, Math. Programming, 22 (1982), pp. 148–162.

[19] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, Amer. Statist. Assoc. J., 58 (1963), pp. 13–30.

[20] D. Hunter, *An upper bound for the probability of a union*, J. Appl. Probab., 13 (1976), pp. 597–603.

[21] D. S. Johnson, *Approximation algorithms for combinatorial problems*, J. Comput. System Sci., 9 (1974), pp. 256–278.

[22] A. Kolen and A. Tamir, *Covering problems*, in Discrete Location Theory, P. B. Mirchandani and R. L. Francis, eds., Wiley-Interscience, New York, 1990, pp. 263–304.

[23] L. Lovász, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383–390.

[24] C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, J. ACM, 41 (1994), pp. 960–981.

[25] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, London, 1995.

[26] M. Naor and R. M. Roth, *Optimal file sharing in distributed networks*, SIAM J. Comput., 24 (1995), pp. 158–183.

[27] S. A. Plotkin, D. B. Shmoys, and É. Tardos, *Fast approximation algorithms for fractional packing and covering problems*, Math. Oper. Res., 20 (1995), pp. 257–301.

[28] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.

[29] P. Raghavan, *Randomized approximation algorithms in combinatorial optimization*, in Proceedings of the FST and TCS Conference, 1994, Lecture Notes in Comput. Sci. 880, Springer-Verlag, Berlin, pp. 300–317.

[30] P. Raghavan and C. D. Thompson, *Randomized rounding: A technique for provably good algorithms and algorithmic proofs*, Combinatorica, 7 (1987), pp. 365–374.

[31] T. K. Sarkar, *Some lower bounds of reliability*, Technical Report 124, Department of Operations Research and Statistics, Stanford University, Stanford, CA, 1969.

[32] D. R. Shier, *Network Reliability and Algebraic Structures*, Oxford Science Publications, Oxford University Press, London, 1991.

[33] D. B. Shmoys, É. Tardos, and K. Aardal, *Approximation algorithms for facility location problems*, in Proceedings of the ACM Symposium on Theory of Computing, El Paso, TX, 1997, pp. 265–274.

[34] P. Slavík, *A tight analysis of the greedy algorithm for set cover*, J. Algorithms, 25 (1997), pp. 237–254.

[35] J. Spencer, *Ten Lectures on the Probabilistic Method*, SIAM, Philadelphia, 1987.

[36] J. H. Spencer, *Six standard deviations suffice*, Trans. Amer. Math. Soc., 289 (1985), pp. 679–706.

[37] A. Srinivasan, *An extension of the Lovász local lemma, and its applications to integer programming*, in Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms, Atlanta, GA, 1996, pp. 6–15.

[38] A. Srinivasan, *Improved approximations for edge-disjoint paths, unsplittable flow, and related routing problems*, in Proceedings of the IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, 1997, pp. 416–425.

[39] P. Turán, *On an extremal problem in graph theory*, Mat. Fiz. Lapok, 48 (1941), pp. 436–452.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 671–684

AN EFFICIENT ALGORITHM FOR GENERATING NECKLACES WITH FIXED DENSITY[*]

FRANK RUSKEY[†] AND JOE SAWADA[†]

Abstract. A k-ary necklace is an equivalence class of k-ary strings under rotation. A necklace of fixed density is a necklace where the number of zeros is fixed. We present a fast, simple, recursive algorithm for generating (i.e., listing) fixed-density k-ary necklaces or aperiodic necklaces. The algorithm is optimal in the sense that it runs in time proportional to the number of necklaces produced.

Key words. necklaces, Lyndon words, fixed density, CAT algorithm, generate, difference covers

AMS subject classifications. 05-04, 68R05, 68R15

PII. S0097539798344112

1. Introduction.

There are many reasons to develop algorithms for producing lists of basic combinatorial objects. First, the algorithms are truly useful and find many applications in diverse areas such as hardware and software testing, nonparametric statistics, and combinatorial chemistry. Second, the development of these algorithms can lead to mathematical discoveries about the objects themselves, either experimentally or through insights gained in the development of the algorithms.

The primary performance goal in an algorithm for listing a combinatorial family is an algorithm whose running time is proportional to the number of objects produced. In this paper an *efficient* algorithm is one that uses only a constant amount of computation per object, in an amortized sense. Such algorithms are also said to be constant amortized time (CAT) algorithms.

Necklaces are a fundamental type of combinatorial object, arising naturally, for example, in the construction of single-track Gray codes, in the enumeration of irreducible polynomials over finite fields, and in the theory of free Lie algebras. Efficient algorithms for exhaustively generating necklaces were first developed by Fredricksen and Kessler [4] and Fredricksen and Maiorana [5], although they did not prove that they were efficient. They were proven to be efficient by Ruskey, Savage, and Wang [8]. Closely related algorithms for generating Lyndon words (aperiodic necklaces) were developed by Duval [3] and shown to be efficient by Berstel and Pocchiola [1]. Subsequently, a recursive algorithm was developed that was more flexible and easier to analyze than the earlier algorithms, which were all iterative [2]. In many applications not all necklaces are required, but rather only those of fixed density (the number of zeros is fixed). Previous to this paper, no efficient generation algorithm for fixed-density necklaces was known.

Previous fixed-density necklace algorithms had running times of $O(n \cdot N(n,d))$ (Wang and Savage [9]) and $O(N(n))$ (Fredricksen and Kessler [4]), where $N(n,d)$ denotes the number of necklaces with length n and density d and $N(n)$ denotes the number of necklaces with length n. Wang and Savage base their algorithm on finding a Hamilton cycle in a graph related to a tree of necklaces. The main feature

[*]Received by the editors August 31, 1998; accepted for publication (in revised form) December 23, 1998; published electronically November 23, 1999. This research was supported by the NSERC.
http://www.siam.org/journals/sicomp/29-2/34411.html

[†] Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada (fruskey@csr.uvic.ca, jsawada@csr.uvic.ca).

of their algorithm is that it also generates the strings in Gray code order. The basis of Fredricksen and Kessler's algorithm is a mapping of lexicographically ordered compositions to necklaces. Both algorithms consider only binary necklaces, but our results apply over a general alphabet. We take a new approach by first modifying Ruskey's recursive algorithm for generating necklaces [2] and then optimizing it for the fixed-density case. Recursive algorithms have several advantages over their iterative counterparts. They are generally simpler and easier to analyze. They are more suitable to conversion to backtracking algorithms, since subtrees are easily pruned from the computation tree. In fact, we have used just such a backtracking to discover new minimal difference covers (sets of numbers achieving all possible differences, mod n).

In the following section we will give some definitions related to necklaces. In section 3 we will introduce a fast algorithm for generating fixed-density k-ary necklaces. In section 4 we analyze the algorithm, proving the algorithm is CAT for any density. In section 5 we conclude by outlining an application and some future work.

2. Background and definitions. A k-ary *necklace* is an equivalence class of k-ary strings under rotation. We identify each necklace with the lexicographically least representative in its equivalence class. The set of all k-ary necklaces with length n is denoted $\mathbf{N}_k(n)$. For example, $\mathbf{N}_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$.

An important class of necklaces are those that are aperiodic. An aperiodic necklace is called a *Lyndon word*. Let $\mathbf{L}_k(n)$ denote the set of all k-ary Lyndon words with length n. For example, $\mathbf{L}_2(4) = \{0001, 0011, 0111\}$. The cardinality of $\mathbf{L}_k(n)$ is denoted $L_k(n)$.

A string α is a *prenecklace* if it is a prefix of some necklace. The set of all k-ary prenecklaces with length n is denoted $\mathbf{P}_k(n)$. For example, $\mathbf{P}_2(4) = \mathbf{N}_2(4) \cup \{0010, 0110\}$. The cardinality of $\mathbf{P}_k(n)$ is $P_k(n)$.

We denote fixed-density necklaces, Lyndon words, and prenecklaces in a similar manner by adding the additional parameter d to represent the number of nonzero characters in the strings. We refer to the number d as the density of the string. Thus the set of k-ary necklaces with density d is represented by $\mathbf{N}_k(n,d)$ and has cardinality $N_k(n,d)$. For example, $\mathbf{N}_3(4,2) = \{0011, 0012, 0021, 0022, 0101, 0102, 0202\}$. Similarly, the set of fixed-density Lyndon words is represented by $\mathbf{L}_k(n,d)$ with cardinality $L_k(n,d)$. The set of fixed-density prenecklaces is denoted by $\mathbf{P}_k(n,d)$ and has cardinality $P_k(n,d)$. In addition to these familiar terms we introduce the set $\mathbf{P}'_k(n,d)$, which is the elements of $\mathbf{P}_k(n,d)$ whose last character is nonzero. Its cardinality is denoted $P'_k(n,d)$.

To count fixed-density necklaces we let $N(n_0, n_1, \ldots, n_{k-1})$ denote the number of necklaces composed of n_i occurrences of the symbol i for $i = 0, 1, \ldots, k-1$. Let the density of the necklace $d = n_1 + \cdots + n_{k-1}$ and $n_0 = n - d$. It is known from Gilbert and Riordan [6] that

$$N(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \backslash gcd(n_0, \ldots, n_{k-1})} \phi(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!}. \tag{2.1}$$

To get the number of fixed-density necklaces with length n and density d, we sum over all possible values of $n_1, n_2, \ldots, n_{k-1}$:

$$N_k(n,d) = \sum_{n_1 + \cdots + n_{k-1} = d} N(n-d, n_1, \ldots, n_{k-1}).$$

The number of fixed-density Lyndon words is defined similarly:

$$L(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \backslash gcd(n_0, n_1, \ldots, n_{k-1})} \mu(j) \frac{(n/j)!}{(n_0/j)!(n_1/j)! \cdots (n_{k-1}/j)!},$$

$$L_k(n,d) = \sum_{n_1 + \cdots + n_{k-1} = d} L(n-d, n_1, \ldots, n_{k-1}).$$

In the binary case these expressions simplify as follows:

$$N_2(n,d) = \frac{1}{n} \sum_{j \backslash gcd(n,d)} \phi(j) \binom{n/j}{d/j},$$

$$L_2(n,d) = \frac{1}{n} \sum_{j \backslash gcd(n,d)} \mu(j) \binom{n/j}{d/j}.$$

Currently, it is not known how to count fixed-density prenecklaces.

In the following section we will introduce a CAT algorithm to generate fixed-density necklaces. When analyzing the performance of our algorithm we make use of the following lemmas about prenecklaces and Lyndon words.

Cattell et al. [2] give a lemma that characterizes prenecklaces by making use of a function lyn on strings, which is the length of the longest Lyndon prefix of the string:

$$lyn(a_1 a_2 \cdots a_n) = \max\{1 \le p \le n | a_1 a_2 \cdots a_p \in \mathbf{L}_k(p)\}.$$

LEMMA 2.1. *Let k-ary string $\alpha = a_1 \cdots a_n$ and $p = lyn(\alpha)$. Then $\alpha \in \mathbf{P}_k(n)$ if and only if $a_{j-p} = a_j$ for $j = p+1, \ldots, n$.*

Reutenauer [7] gives a useful lemma about Lyndon words. Inequalities between words are always with respect to lexicographic order.

LEMMA 2.2. *If α and β are Lyndon words with $\alpha < \beta$, then $\alpha\beta$ is a Lyndon word.*

3. Generating fixed-density necklaces. We use a two-step approach to develop a fast algorithm for generating fixed-density necklaces. First we create a new necklace algorithm based on the recursive necklace-generation algorithm $\mathsf{Gen}(t,p)$ (Figure 3.1) [2]. We then optimize this new necklace algorithm for the fixed-density case by making a few key observations about fixed-density necklaces.

To begin we give a brief overview of $\mathsf{Gen}(t,p)$. The general approach of this algorithm is to generate all length n prenecklaces. The prenecklace being generated is stored in the array a with one position for each character. We assume that $a_0 = 0$. The initial call is $\mathsf{Gen}(1,1)$ and each recursive call appends a character to the prenecklace to get a new prenecklace. At the beginning of each recursive call to $\mathsf{Gen}(t,p)$, the length of the prenecklace being generated is $t-1$ and the length of the longest Lyndon prefix is p. As long as the length of the current prenecklace is less than n, each call to $\mathsf{Gen}(t,p)$ makes one recursive call for each valid value for the next character in the string, updating the values of both t and p in the process. This algorithm can generate necklaces, Lyndon words, or prenecklaces of length n in lexicographic order by specifying which object we want to generate. The function $\mathsf{PrintIt}(p)$ allows us to differentiate between these various objects as shown in Figure 3.2.

The computation tree for $\mathsf{Gen}(t,p)$ consists of all prenecklaces of length less than or equal to n. As an example, we show a computation tree for $N_2(4)$ in Figure 3.3. By comparing the number of nodes in the computation tree to the number of objects generated it was shown that this algorithm is CAT [2].

```
procedure Gen ( $t$, $p$ : integer );
local $j$ : integer;
begin
    if $t > n$ then PrintIt( $p$ )
    else begin
        $a_t := a_{t-p}$;    Gen( $t+1$, $p$ );
        for $j \in \{a_{t-p}+1, \ldots, k-2, k-1\}$ do begin
            $a_t := j$;    Gen( $t+1$, $t$ );
        end;
    end;
end {of Gen};
```

FIG. 3.1. *The recursive necklace algorithm.*

Sequence type	PrintIt(p)
Prenecklaces ($\mathbf{P}_k(n)$)	Println($a[1..n]$)
Lyndon words ($\mathbf{L}_k(n)$)	**if** $p = n$ **then** Println($a[1..n]$)
Necklaces ($\mathbf{N}_k(n)$)	**if** n mod $p = 0$ **then** Println($a[1..n]$)

FIG. 3.2. *Different objects output by different versions of* PrintIt(p).

3.1. Modified necklace algorithm. For every necklace of positive density, the last character of the string must be nonzero. Thus, if we are concerned only with generating necklaces or Lyndon words we can reduce the size of the computation tree by compressing all the prenecklaces whose last character is 0. Looking at Figure 3.3, we want to generate only the nodes in bold. This results in the modified computation tree shown in Figure 3.4. Notice that at each successive level in this tree we are incrementing the density of the prenecklace rather than the length. To generate this modified tree we create a recursive routine based on the original necklace algorithm in Figure 3.1; however, rather than determining the valid values for the next position in the string, we need to determine both the valid positions and the values for the next nonzero character.

To make this change we use the array a to hold the positions of the nonzero characters and maintain another array b to indicate the values of the nonzero characters. The ith element of the array a represents the position of the ith nonzero character, and the ith element of the array b represents the value of the ith nonzero character. Thus if we generate a necklace with length 7 with $a = [3, 4, 5, 7]$ and $b = [1, 3, 2, 1]$, the corresponding necklace is 0013201. (We can also maintain the original necklace structure by performing some extra constant-time operations.) Note that in the binary case, the second array b is not necessary since all nonzero characters must be 1. We use the parameter t to indicate the current density of the string. The length of the current string is a_t. Since all Lyndon prefixes end in a nonzero character, we let a_p indicate the length of the longest Lyndon prefix. Using these two parameters, we can compute all valid positions and values for the next nonzero character.

To determine the valid positions and values for the next nonzero character and to maintain the lexicographic ordering we compute the maximum position and the minimum value for that position so that the new string still has the prenecklace property. We compute this maximal position for the next character using the following

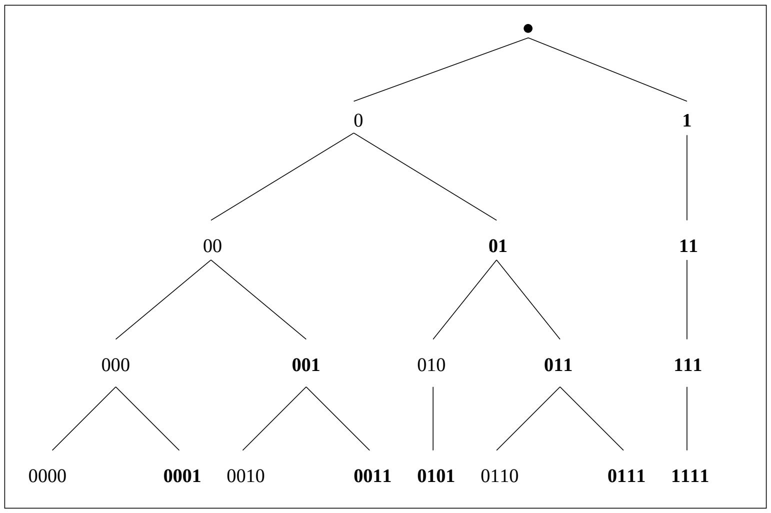


FIG. 3.3. *Computation tree for $N_2(4)$ from* $\mathsf{Gen}(t,p)$.

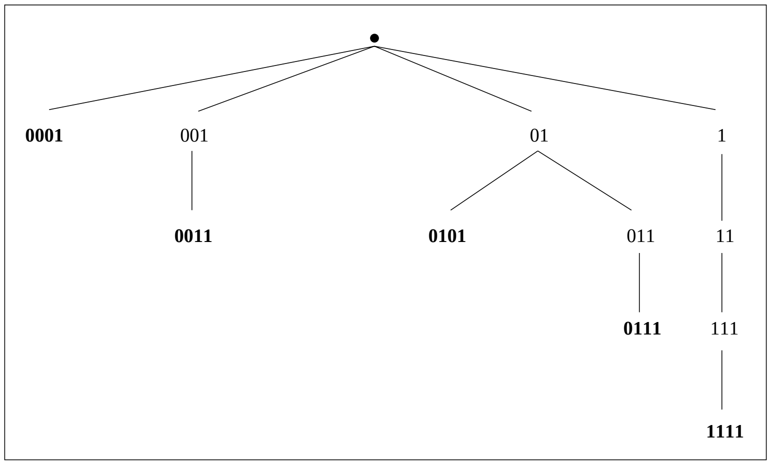


FIG. 3.4. *Computation tree for $N_2(4)$ from* $\mathsf{Gen2}(t,p)$.

expression:

$$\lfloor (t+1)/p \rfloor a_p + a_{(t+1) \bmod p}.$$

The minimal value for this position is b_{t+1-p}. By the properties of prenecklaces all larger values at the maximal position are also valid [8]. Also, all positions before the maximum position and greater than the position of the last assigned nonzero character (a_t) can hold all values ranging from 1 to $k-1$. (Note that since we want to generate all necklaces with length n, we restrict the position to be less than or equal to n.) For each of these valid combinations of position and value, we lexicographically assign the position to a_{t+1} and the value to b_{t+1}, followed by a recursive call updating both t and p. Finally, if the position of the last nonzero element is greater than or equal to n, we call the $\mathsf{PrintIt}(p)$ function to print out either the Lyndon words or necklaces in a similar manner to the original algorithm $\mathsf{Gen}(t,p)$.

```
procedure Gen2 ( $t$, $p$ : integer );
local $i, j, max$ : integer;
begin
    if $a_t \geq n$ then PrintIt( $p$ )
    else begin
        $max = a_{t+1-p} + a_p$;
        if $max \leq n$ then begin
            $a_{t+1} := max$;
            $b_{t+1} := b_{t+1-p}$;
            Gen2 ( $t+1$, $p$ );
        end else begin
            $max := n$;    $a_{t+1} := n$;    $b_{t+1} := 1$;
            Gen2 ( $t+1$, $t+1$ );
        end;
        for $i \in \{b_{t+1}+1, \ldots, k-2, k-1\}$ do begin
            $b_{t+1} := i$;
            Gen2 ( $t+1$, $t+1$ );
        end;
        for $j \in \{max-1, max-2, \ldots, a_t+1\}$ do begin
            $a_{t+1} := j$;
            for $i \in \{1, \ldots, k-2, k-1\}$ do begin
                $b_{t+1} := i$;
                Gen2 ( $t+1$, $t+1$ );
    end; end; end;
end {of Gen2};
```

FIG. 3.5. *Modified recursive necklace algorithm.*

This modified algorithm, Gen2(t, p), for generating necklaces is given in Figure 3.5. Each initial branch of the computation tree is a result of a separate call to Gen2(t, p), each call specifying a different combination for the position and value of the first nonzero character. Note that the zero string is not generated by Gen2(t,p) and must be generated separately. The nodes of the resulting computation tree for Gen2(t, p) are all prenecklaces with length less than or equal to n whose last character is nonzero.

A complete C program for this modified necklace algorithm is available from the authors. A simplified program for the binary case is also available. Observe that we are not restricted to generating the necklaces in lexicographic order. Many orders are possible by reordering the recursive calls.

3.2. Fixed-density necklace algorithm. We now optimize our modified algorithm for the fixed-density case by making several observations. First, we restrict the position of the first nonzero character depending on the density. In particular, there are no necklaces with density d that can have the first nonzero character in a position after $n - d + 1$ or before $\lfloor (n-1)/d + 1 \rfloor$. Also, if we are generating a string with length n and density d and have just placed the ith nonzero character, then the $(i+1)$st nonzero character must come before the position $n - (d - i) + 2$. If we place the next character at or after this position, then any resulting string with length n will have density less than d. Also, because the last nonzero character must be in the nth position, we stop the string generation after placing the $(d-1)$st nonzero character.

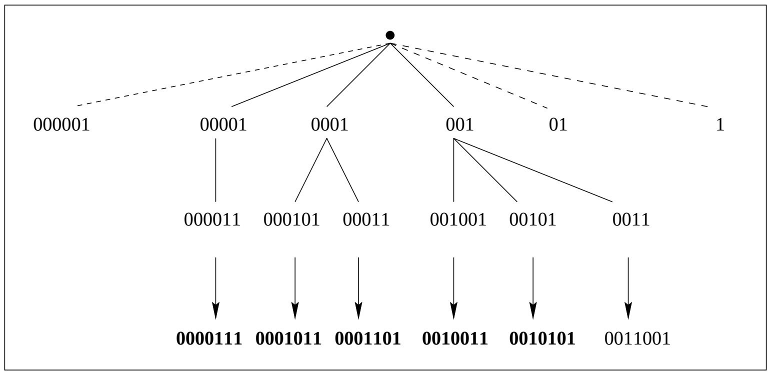


FIG. 3.6. *Computation tree (solid edges only) for $N_2(7,3)$ from* GenFix(t,p).

Thus, the strings generated by following this last restriction are strings with length less than n and density $d-1$. By following this approach, we may generate up to $k-1$ strings for each call to PrintIt(p), since we can place up to $k-1$ characters in the nth position. However, it is not always the case that we will generate all $k-1$ strings or even any strings with each call to PrintIt(p). Thus we add an additional constant-time test to see which values can be placed in the nth position. This test is similar to the test for finding the maximal valid position and minimum value for the next nonzero character as outlined in the previous subsection. Once a minimum value is determined (if there is one at all), we perform the usual tests to determine if the string is a necklace or a Lyndon word. All larger values for the nth position will result in a string that is a Lyndon word [8]. Thus the overall work done in the PrintIt(p) function to determine the valid strings remains constant for each string generated.

In summary, we use our modified necklace algorithm outlined in Figure 3.5 with the following optimizations:

1. The first nonzero character must be between $n-d+1$ and $(n-1)/d+1$ inclusive.
2. The ith nonzero character must be placed at or before the $(n-d+i)$th position.
3. Stop generating when we have assigned $d-1$ nonzero characters.
4. Determine valid values for the nth position in the PrintIt(p) function.

The computation tree for generating $N_2(7,3)$ is given in Figure 3.6. The dotted lines indicate the initial branches we do not need to follow by modification 1. The arrows indicate the strings produced by adding the final character to the nth position. The bold strings indicate the actual necklaces produced by the PrintIt(p) function. The remaining string (0011001) is rejected since it is not a necklace.

The algorithm for generating fixed-density necklaces and Lyndon words in lexicographic order is given in Figure 3.7. To generate fixed-density prenecklaces, we generate $N(n+1,d+1)$ and print out only the first n characters, making sure we do not print the same string twice. A complete C program for this fixed-density necklace algorithm is available from the authors. A simplified program for the binary case is also available. In the latter program we make use of the fact that we can generate binary necklaces with density $d > n/2$ by complementing the output from generating necklaces with density $n-d$. In this case, however, the strings generated are not in lexicographic order and are not necessarily the lexicographic representatives for their respective equivalence classes.

```
procedure GenFix ( $t$, $p$ : integer );
local $i, j, max, tail$ : integer;
begin
    if $t \geq d-1$ then PrintIt($p$);
    else begin
        $tail := n - (d-t) + 1$;
        $max := a_{t+1-p} + a_p$;
        if $max \leq tail$ then begin
            $a_{t+1} := max$;
            $b_{t+1} := b_{t+1-p}$;
            GenFix( $t+1$, $p$ );
            for $i \in \{b_{t+1}+1, \ldots, k-2, k-1\}$ do begin
                $b_{t+1} := i$;
                GenFix( $t+1$, $t+1$ );
            end;
            $tail := max - 1$;
        end;
        for $j \in \{tail, tail-1, \ldots, a_t+1\}$ do begin
            $a_{t+1} := j$;
            for $i \in \{1, \ldots, k-2, k-1\}$ do begin
                $b_{t+1} := i$;
                GenFix( $t+1$, $t+1$ );
    end; end; end;
end {of GenFix};
```

FIG. 3.7. *Fixed-density necklace algorithm.*

4. Analysis of algorithm. In this section we show that GenFix(t,p) is CAT. We start the analysis by analyzing several trivial cases. When the desired density of the string is n the computation tree and strings produced are equivalent to the generation of $N_{k-1}(n)$, which we already know is CAT. When the density is zero we simply generate the zero string, and when $d=1$ we generate the $k-1$ strings where the last bit ranges from 1 to $k-1$ and the rest of the string is all zeros. In each case where the density is greater than zero the resulting strings are generated in CAT.

For the nontrivial cases we examine the number of nodes in the computation tree, noting that the amount of work to generate each node is constant. When $1 < d < n$, the nodes in the computation tree consist only of prenecklaces that end in a nonzero bit with density i ranging from 1 to $d-1$ and length ranging from $(n-1)/d+i$ to $n-d+i$. Recall that $\mathbf{P}'_k(n,d)$ is the set of prenecklaces with length n and density d, where the last bit is nonzero. Thus, the size of the computation tree for our fixed-density algorithm $(1 < d < n)$ is bounded by the expression

$$CompTree_k(n,d) \leq \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} P'_k(j,i).$$

Recall that we generate binary fixed-density necklaces with density greater than $n/2$ by generating $N(n, n-d)$ and complementing the output. Therefore, in the case where $k=2$ (and only in this case), we have the restriction that d is less than or equal to $n/2$.

To prove that our algorithm is efficient we will show that the ratio between the size of the computation tree and the number of strings produced is bounded by a constant. Since there does not appear to be a simple explicit formula for $P'_k(n,d)$, our approach will be to derive an upper bound in terms of $N_k(n,d)$ and $L_k(n,d)$.

LEMMA 4.1. $P'_k(n,d) \le N_k(n,d) + L_k(n,d)$.

Proof. We partition $\mathbf{P}'_k(n,d)$ into two categories: necklaces and nonnecklaces. Let the elements of $\mathbf{P}'_k(n,d)$ that are not necklaces be $\mathbf{Q}'_k(n,d)$.

We show that $Q'_k(n,d) \le L_k(n,d)$ by providing an injective mapping of $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$. By Lemma 2.1 each element of the set $\mathbf{Q}'_k(n,d)$ must have the form $\alpha = (a_1 \cdots a_p)^j a_1 \cdots a_m$, where $p = lyn(\alpha)$, $j \ge 1$, and $0 < m < p$. Let n_i be the number of occurrences of the symbol i in $a_1 \cdots a_m$ and define the string $\gamma = 0^{n_0}1^{n_1} \cdots (k-1)^{n_{k-1}}$. We define a function f on the set $\mathbf{Q}'_k(n,d)$ as follows:

$$f(\alpha) = \gamma(a_1 \cdots a_p)^j.$$

For example, $f((002101303)^7 0021013) = 0001123(002101303)^7$. This mapping preserves both length and density. Since γ and $a_1 \cdots a_p$ are both Lyndon words and $\gamma < a_1 \cdots a_p$, it follows from repeated use of Lemma 2.2 that $f(\alpha) \in \mathbf{L}_k(n,d)$.

To show that f is injective consider two unique elements of $\mathbf{Q}'_k(n,d)$: $\alpha = (a_1 \cdots a_p)^s a_1 \cdots a_i$ and $\beta = (b_1 \cdots b_q)^t b_1 \cdots b_j$. If $i = j$, then $f(\alpha) \ne f(\beta)$, since $a_1 \cdots a_p$ and $b_1 \cdots b_q$ are both Lyndon words and $a_1 \cdots a_p \ne b_1 \cdots b_q$. Otherwise assume that $i < j$. Since a_i and b_j are both nonzero, the ith element of $f(\alpha)$ is nonzero and the jth element of $f(\beta)$ is nonzero. Now if the ith element of $f(\beta)$ is nonzero then the $(i+1)$st element must also be nonzero if $f(\alpha) = f(\beta)$. However the $(i+1)$st element of $f(\alpha) = a_1$, which is 0. Thus $f(\alpha) \ne f(\beta)$ for unique $\alpha, \beta \in \mathbf{Q}'_k(n,d)$. Thus f is an injection from $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$.

Now since there exists an injective mapping from $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$ we have $Q'_k(n,d) \le L_k(n,d)$. From earlier discussion we know that $P'_k(n,d) = N_k(n,d) + Q'_k(n,d)$ and thus $P'_k(n,d) \le N_k(n,d) + L_k(n,d)$. ◻

We observe in the binary case that by taking each element from $\mathbf{P}_2(n,d)$ and adding a 1 to the end of the string we get the set $\mathbf{P}'_2(n+1,d+1)$. Thus from the previous lemma we also get an upper bound on $P_2(n,d)$.

COROLLARY 4.2. $P_2(n,d) \le N_2(n+1,d+1) + L_2(n+1,d+1)$.

We can now bound our computation tree as the sum of fixed-density necklaces and fixed-density Lyndon words:

$$CompTree_k(n,d) \le \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} N_k(j,i) + L_k(j,i).$$

However, by plugging the formulas for fixed-density necklaces and Lyndon words into the above expression we end up with a complicated quadruple sum. Therefore we will prove two lemmas, which give simple bounds for fixed-density Lyndon words and necklaces.

LEMMA 4.3. *The following inequality is valid for all* $0 \le d \le n$*:*

$$L_k(n,d) \le \frac{1}{n}\binom{n}{d}(k-1)^d.$$

Proof. Each element of $\mathbf{L}_k(n,d)$ is a representative of an equivalence class of k-ary strings, each with n elements. If we add up the elements from each equivalence class we will get $nL_k(n,d)$ unique strings each of length n and density d. The expression

$\binom{n}{d}(k-1)^d$ counts the total number of k-ary strings with length n and density d. Therefore $L_k(n,d) \leq \frac{1}{n}\binom{n}{d}(k-1)^d$. ☐

A similar bound for $N_k(n,d)$ is more difficult to obtain. Here we bound $N_k(n,d)$ by $L_k(n,d)$.

LEMMA 4.4. *The following inequality is valid for all* $0 < d < n$*:*

$$\frac{1}{n}\binom{n}{d}(k-1)^d \leq N_k(n,d) \leq 2L_k(n,d).$$

Proof. By considering the case when $j = 1$ in (2.1) and noting that the remaining terms are all nonnegative, we have

$$\begin{aligned} N_k(n,d) &\geq \frac{1}{n}\sum_{n_1+\cdots+n_{k-1}=d} \frac{n!}{(n_0!)(n_1!)\cdots(n_{k-1}!)} \\ &= \frac{1}{n}\binom{n}{d}\sum_{n_1+\cdots+n_{k-1}=d} \frac{d!}{(n_1!)\cdots(n_{k-1}!)} \\ &= \frac{1}{n}\binom{n}{d}(k-1)^d. \end{aligned}$$

The final equality is a result of the basic multinomial expansion.

To show that $N_k(n,d) \leq 2L_k(n,d)$, we provide an injective mapping of the periodic necklaces to Lyndon words. If α is a periodic necklace, then $\alpha = (a_1 \cdots a_p)^j$, where $p = lyn(\alpha)$ and $j > 1$. Since $d < n$ we know that $a_1 = 0$. We define a function g on all periodic necklaces with length n and density d as follows:

$$g(\alpha) = 0(a_1 \cdots a_p)^{j-1} a_2 \cdots a_p.$$

This function simply moves the bit $a_{p(j-1)+1} = a_1 = 0$ to the front of the string. This operation preserves both length and density. Since $(a_1 \cdots a_p)^{j-1} a_2 \cdots a_p$ is a Lyndon word, by Lemma 2.2 $g(\alpha)$ is a Lyndon word.

To show that g is an injection we consider two unique periodic necklaces: $\alpha = (a_1 \cdots a_p)^i$ and $\beta = (b_1 \cdots b_q)^j$. If $p = q$ and $g(\alpha) = g(\beta)$, then $a_1 \cdots a_p = b_1 \cdots b_q$, contradicting the fact that $\alpha \neq \beta$. If $p \neq q$, then assume that $p < q$. This implies that $i > j > 1$. Now comparing the characters in positions $2, 3, \ldots, q+1$ of $g(\alpha)$ and $g(\beta)$ we observe that if $g(\alpha) = g(\beta)$ then $b_1 \cdots b_q = (a_1 \cdots a_p)^t a_1 \cdots a_s$ for some $t \geq 1$ and $1 \leq s \leq p$. However, since $a_1 \cdots a_p$ is a Lyndon word, then $(a_1 \cdots a_p)^t a_1 \cdots a_s$ is periodic if $s = p$ and is not a necklace if $s < p$. This contradicts the fact that $b_1 \cdots b_q$ is a Lyndon word. Thus $g(\alpha) \neq g(\beta)$ for unique periodic necklaces α and β. Therefore g is an injective mapping from the periodic necklaces to Lyndon words.

Since there exists an injective mapping from the periodic strings of $\mathbf{N}_k(n,d)$ to $\mathbf{L}_k(n,d)$ we get the result $N_k(n,d) \leq 2L_k(n,d)$. ☐

Using the previous lemmas we can simplify our upper bound on the size of the computation tree:

$$\begin{aligned} CompTree_k(n,d) &= \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} P'_k(j,i) \\ &\leq \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} N_k(j,i) + L_k(j,i) \end{aligned}$$

$$
\begin{aligned}
&\le 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} L_k(j,i)\\
&\le 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} \frac{1}{j}\binom{j}{i}(k-1)^i\\
&= 3\sum_{i=1}^{d-1}\frac{1}{i}(k-1)^i\sum_{j=1}^{n-d+i}\binom{j-1}{i-1}\\
&= 3\sum_{i=1}^{d-1}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i. \qquad (4.1)
\end{aligned}
$$

To get the last two equalities we use some basic binomial coefficient identities.

To simplify this bound for the computation tree even more, we inductively prove yet another upper bound for the remaining sum in (4.1). We first prove an upper bound for the case when $k > 2$ and $1 < d < n$. We then provide a similar proof for the case when $k = 2$. In the latter case we take advantage of the fact that we can generate binary necklaces with $d > n/2$ by generating necklaces with density $n-d$ and then complementing the output of each generated necklace to get all necklaces with density d. Once again, this is the only situation where the strings are not generated in lexicographic order. Thus when $k = 2$, we only consider the case when $1 < d \le n/2$.

LEMMA 4.5. *For $2 \le d < n$ and $k > 2$,*

$$\sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i \;<\; \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}.$$

Proof. We prove the lemma by induction on d. Let

$$S_k(n,d) = \sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i.$$

Basis: $d = 2$ or 3, $n \ge 3$. Observe that this covers all cases for $n = 3, 4$:

$$
\begin{aligned}
&d = 2\colon S_k(n,2) = 0 < 2(n-1)(k-1),\\
&d = 3\colon S_k(n,3) = (n-2)(k-1) < \binom{n-1}{2}(k-1)^2.
\end{aligned}
$$

Assume: $S_k(n,d) < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}$ for $1 < d < n-1$, $k > 2$, and $n \ge 5$. Consider $S_k(n,d+1)$:

$$
\begin{aligned}
S_k(n,d+1) &= \sum_{i=1}^{d-1}\frac{1}{i}\binom{n-d-1+i}{i}(k-1)^i\\
&= \sum_{i=1}^{d-2}\frac{1}{i}\binom{(n-1)-d+i}{i}(k-1)^i + \frac{1}{d-1}\binom{n-2}{d-1}(k-1)^{d-1}\\
&< \frac{2}{d-1}\binom{(n-1)-1}{d-1}(k-1)^{d-1} + \frac{1}{d-1}\binom{n-2}{d-1}(k-1)^{d-1}\\
&= \frac{3}{d-1}\binom{n-2}{d-1}(k-1)^{d-1}
\end{aligned}
$$

$$= \frac{3d}{(d-1)(n-1)}\binom{n-1}{d}(k-1)^{d-1}$$
$$\leq \frac{2}{d}\binom{n-1}{d}(k-1)^{d}.$$

To show that the last inequality is correct we prove that $\frac{3d}{(d-1)(n-1)} \leq \frac{2}{d}(k-1)$ for $n \geq 5$. By multiplying both sides by $\frac{d}{k-1}$ we get $\frac{3d^2}{(d-1)(n-1)(k-1)} \leq 2$. The LHS of this inequality is maximized when we maximize $d = n-2$ and minimize $k = 3$. By substituting these values and rearranging we get

$$\begin{aligned} 3(n-2)(n-2) &\leq 4(n-1)(n-3), \\ 0 &\leq 4(n^2-4n+3) - 3(n^2-4n+4), \\ 0 &\leq n(n-4). \end{aligned}$$

This equality is true for $n \geq 4$. ☐

LEMMA 4.6. *For $2 \leq d \leq n/2$ and $k = 2$,*

$$\sum_{i=1}^{d-2} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i \; < \; \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}.$$

Proof. We prove the lemma by induction on d. Let

$$S_k(n,d) = \sum_{i=1}^{d-2} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i.$$

Basis: $d = 2$ or 3, $n \geq 3$. Observe that this covers all cases for $n = 3,4,5,6,7$:

$$\begin{aligned} &d = 2\colon S_k(n,2) = 0 < 2(n-1)(k-1), \\ &d = 3\colon S_k(n,3) = (n-2)(k-1) < \binom{n-1}{2}(k-1)^2. \end{aligned}$$

Assume: $S_k(n,d) < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}$ for $1 < d < n/2$ and $n \geq 5$. From the proof of the previous lemma we know

$$\begin{aligned} S_k(n,d+1) &< \frac{3d}{(d-1)(n-1)}\binom{n-1}{d}(k-1)^{d-1} \\ &\leq \frac{2}{d}\binom{n-1}{d}(k-1)^{d}. \end{aligned}$$

To show that the last inequality is correct we prove that $\frac{3d}{(d-1)(n-1)} \leq \frac{2}{d}(k-1)$ for $n \geq 8$. By substituting the value 2 for k and multiplying both sides by d we get $\frac{3d^2}{(d-1)(n-1)} \leq 2$. The LHS of this inequality is maximized when we maximize $d = \frac{n}{2} - 1$. By substituting this value for d and rearranging the terms we get

$$\begin{aligned} 3\left(\frac{n}{2}-1\right)^2 &\leq 2\left(\frac{n}{2}-2\right)(n-1), \\ 0 &\leq 2\left(\frac{n}{2}-2\right)(n-1) - 3\left(\frac{n}{2}-1\right)^2, \\ 0 &\leq 2\left(\frac{n^2}{2} - \frac{5n}{2} + 2\right) - 3\left(\frac{n^2}{4} - n + 1\right), \\ 0 &\leq \frac{n^2}{4} - 2n + 1. \end{aligned}$$

By solving this quadratic we see that the inequality holds for $n \geq 8$. ◻

We now use the previous lemmas to get a simple upper bound on the size of the computation tree:

$$\begin{aligned} CompTree_k(n,d) &\leq 3\sum_{i=1}^{d-1} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i \\ &= 3\sum_{i=1}^{d-2} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \\ &< \frac{6}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \\ &= \frac{9}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}. \end{aligned}$$

Recall that our goal is to prove that the ratio of nodes in the computation tree to the number of strings produced is bounded by a constant. From Lemma 4.4 we have a lower bound on the number of strings produced:

$$N_k(n,d) > \frac{1}{n}\binom{n}{d}(k-1)^d = \frac{1}{d}\binom{n-1}{d-1}(k-1)^d.$$

Thus the ratio of our computation tree to necklaces produced is

$$\frac{CompTree_k(n,d)}{N_k(n,d)} < 9\frac{d}{(d-1)(k-1)} \leq 18.$$

Experimentally, this constant is less than 3.

THEOREM 4.7. *Algorithm* GenFix *for generating fixed-density k-ary necklaces is CAT.*

5. Future work and an application. In this paper we have presented a CAT algorithm for generating fixed-density k-ary necklaces. This algorithm is used when we want to generate necklaces where the number of zeros is fixed; however, if we want all necklaces where the number of occurrences for every character is fixed, then our algorithm works only for the binary case. An efficient algorithm for the k-ary case would be very interesting, but currently does not exist. Another open problem is to count the number of fixed-density prenecklaces; the number of fixed-density necklaces and Lyndon words is known and was given in this paper.

5.1. Generating difference covers. As an application, we embed our fixed-density necklace algorithm into a program that generates difference covers. A set $D = \{a_1, \ldots, a_k\}$, $1 < a_i < n$, is called an (n,k) difference cover if for every $d \neq 0 \bmod n$ there exists an ordered pair (a_i, a_j) in D such that $a_i - a_j = d \bmod n$. For example, the set $\{1, 2, 3, 6\}$ is a $(10,4)$ difference cover. An (n,k) difference cover is minimal if an $(n,k-1)$ difference cover does not exist.

To generate all difference covers (n,k) we generate all fixed-density necklaces $\mathbf{N}_2(n,k)$ where the position of each one in the necklace represents a number in the set D. To determine whether the necklace represents a difference cover, we keep track of information about each ordered pair. This additional work takes at worst case $O(k)$ time for every node in the computation tree. Thus the overall running time for generating all the (n,k) difference covers is $O(kN_2(n,k))$.

In practice, it is useful to know whether or not an (n, k) difference cover exists. When n gets large the search space may become infeasible to work with; however, if we have some intuition about what the first few numbers may be in the set D, we can customize our algorithm to drastically reduce the search space. Using this strategy we were able to prove the existence of a (131, 13) difference cover, namely

$$\{1, 8, 27, 33, 34, 44, 57, 71, 73, 79, 88, 91\}.$$

A complete C program for generating difference covers with equivalence under rotation is available from the authors.

REFERENCES

[1] J. Berstel and M. Pocchiola, *Average cost of Duval's algorithm for generating Lyndon words*, Theoret. Comput. Sci., 132 (1994), pp. 415–425.

[2] K. Cattell, F. Ruskey, J. Sawada, C. R. Miers, and M. Serra, *Fast Algorithms to Generate Unlabeled Necklaces and Irreducible Polynomials over GF*(2), manuscript, 1998.

[3] J.-P. Duval, *Génération d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée*, Theoret. Comput. Sci., 60 (1988), pp. 255–283.

[4] H. Fredricksen and I. J. Kessler, *An algorithm for generating necklaces of beads in two colors*, Discrete Math., 61 (1986), pp. 181–188.

[5] H. Fredricksen and J. Maiorana, *Necklaces of beads in k colors and k-ary de Bruijn sequences*, Discrete Math., 23 (1978), pp. 207–210.

[6] E. N. Gilbert and J. Riordan, *Symmetry types of periodic sequences*, Illinois J. Math., 5 (1961), pp. 657–665.

[7] C. Reutenauer, *Free Lie Algebras*, Clarendon Press, Oxford, England, 1993.

[8] F. Ruskey, C. D. Savage, and T. Wang, *Generating necklaces*, J. Algorithms, 13 (1992), pp. 414–430.

[9] T. M. Y. Wang and C. D. Savage, *A Gray code for necklaces of fixed density*, SIAM J. Discrete Math., 9 (1996), pp. 654–673.

SIAM J. COMPUT.
Vol. 29, No. 2, pp. 685–695

FINDING AN EVEN SIMPLE PATH IN A DIRECTED PLANAR GRAPH[*]

ZHIVKO PRODANOV NEDEV[†]

Abstract. In this paper we show that the following problem, the *even simple path* (ESP) *problem for directed planar graphs*, is solvable in polynomial time:

Given: a directed planar graph $G = (V, E)$ and two nodes $s\,(startingnode)$, $t\,(targetnode) \in V$;

Find: a simple path (i.e., without repeated nodes) from s to t of even length. (The length of the path is the number of edges it contains.)

Key words. labeled directed graphs, NP-completeness, polynomial-time algorithms, regular expressions, simple paths, planar graphs

AMS subject classifications. 68P, 68Q, (68Q25), 68R10

PII. S0097539797330343

1. Introduction.

A linear-time algorithm solving this problem for general undirected graphs is given in [5]; it is also shown there that the same problem (ESP) is NP-complete [3] for directed graphs. This is achieved by polynomially reducing the *path via node problem for directed graphs* to the ESP problem. The former problem is one of the many subgraph homeomorphism problems for *directed graphs* that have been shown to be NP-complete in [2].

In [6], the regular simple path (RSP) problem is investigated:

Given: a finite alphabet $\sum$,
a database in the form of a edge-labeled directed graph G, with labels taken from $\sum$,
a query R in the form of a regular expression over the same alphabet $\sum$,
and a pair of vertices (x, y) from G;

Find: a simple path p in G from x to y such that the concatenation of labels along p satisfies R.

In [6], it is also shown that RSP is NP-hard in general. One of the cases used to prove this statement is the query $R = (\_\ \_)*$. (Here the symbol "_" stands for any letter of the alphabet.) This is just the ESP problem stated in another way—in this case the labels play no role. As has already been pointed out [5], this problem is NP-complete for a general digraph. But in [6] several algorithms are presented which run in polynomial time in the size of the graph for certain classes of graphs and queries.

As a result of our paper we can state that the same query, $R = (\_\ \_)*$, is polynomially solvable if G is a planar graph. Thus, our paper broadens the knowledge of the RSP problem—expanding the number of situations where this NP-complete problem can be polynomially solved.

In our paper we show that the ESP problem is solvable in polynomial time by polynomially reducing it to the following *k disjoint paths problem for directed planar graphs*:

[*]Received by the editors November 20, 1997; accepted for publication (in revised form) October 8, 1998; published electronically December 3, 1999.
http://www.siam.org/journals/sicomp/29-2/33034.html

[†]Department of Computer Science, University of Cape Town, Rondebosch 7700, South Africa (nedev@cs.uct.ac.za).

Given: a directed planar graph $G = (V, A)$ and k pairs $(r_1, s_1), \ldots, (r_k, s_k)$ of vertices of G;

Find: k pairwise vertex-disjoint paths $P_1, \ldots, P_k$ in G, where P_i runs from r_i to s_i $(i = 1, \ldots, k)$.

for the simplest case $k = 2$. Recently it was shown that this problem is polynomially solvable for any fixed k [7].

In our solution we introduce two new data structures: The first one is a plane region with oriented boundaries, called "list," which has theoretical value only, as the number of list structures in a planar graph might be exponential (we present such an example in section 5). The second data structure is a special case of the first structure; we call it "simple list." We prove that the number of simple lists in any planar graph is polynomial and we give a polynomial algorithm to generate all of them. In this algorithm we also invent a special modification of depth-first search (DFS) [8] for the case of planar graphs.

The rest of the paper is organized as follows. Section 2 gives preliminary definitions. Section 3 gives the general idea of the main algorithm. Section 4 contains the basic lemmas about list superfaces. Section 5 introduces the new modification of the DFS algorithm for planar digraphs and gives polynomial algorithm for finding all its simple lists. Section 6 presents the final algorithm for the ESP problem and section 7 gives possible future extensions.

2. Preliminaries.

The following "basics for planar graphs" have been taken almost directly from [4].

Consider a directed graph $G = (V, E)$ with edge set E and vertex set V. Let $n = |V|$ and $m = |E|$. We can draw a picture G' of G in the plane as follows: For each vertex $v \in V$, we draw a distinct point v'; for each edge $(u, v) \in E$, we draw a simple arc connecting the two points u' and v'. We call this arc an *embedding* of the edge (u, v). For brevity, we will sometimes identify graphs with their pictures thus drawn in the plane. If no arcs of G' cross each other, we call G' a *planar embedding* or *embedding* of G. If G has a planar embedding, then we say that G is *planar*.

Let G be a connected embedded planar digraph. If the edges and vertices of G are deleted from the plane in which G is embedded, the plane is divided into disconnected regions. Exactly one of the regions is infinite; all others are finite. Each region is called a *face* of G—in this paper we will call it an *elementary face* to distinguish it from a superface (defined below). The infinite region is also called the *external* face; the finite regions are called *internal* faces. The *boundary* of a face f is the sequence of edges and vertices surrounding f. Any face boundary is a simple cycle.

DEFINITION 1. *If we have a digraph G, then we will call any set of edges that forms a simple cycle (possibly disregarding the directions of the edges), together with its interior or exterior, a* superface. *We will call that simple cycle the* boundary *of the superface. This interior or exterior part of the plane that is associated with the boundary is the* inside *part of the superface; further, if a vertex or edge lies there, we will say that it is inside even if it is from the exterior part of the plane.*

Please note that for any given simple cycle, we have exactly two superfaces with boundary this cycle: one with finite and one with infinite face.

DEFINITION 2. *A superface is called a* list superface *LS or* list *if its boundary contains two special nodes b and e, the first called the beginning and the second called the end, such that all the edges from the boundary moving clockwise from the beginning b to the end e are in one direction—from the beginning to the end—and all the edges from the boundary moving counterclockwise from the beginning b to the end e are in*

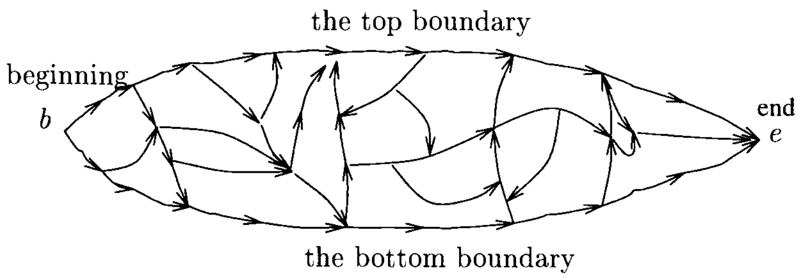


FIG. 1. *An example of a list superface.*

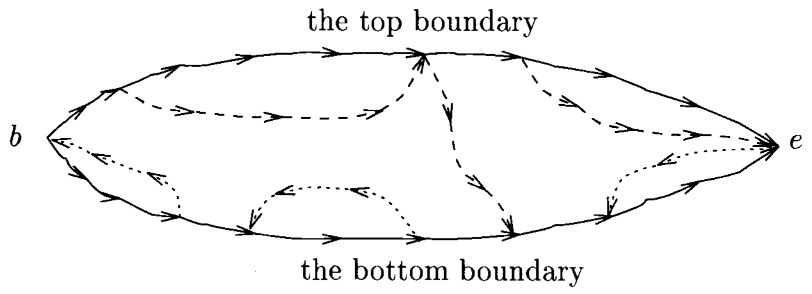


FIG. 2. *An example of forbidden (dashed) and permitted (dotted) paths in a simple list superface.*

one direction—from the beginning to the end (see Figure 1*).*

DEFINITION 3. *A list superface LS is called a* simple *list superface if for all directed paths $P = p_0, p_1, \dots, p_n$ where p_0 and p_n lie on the boundary, $p_1, \dots, p_{n-1}$ are strictly from the inside, and p_n is a predecessor of p_0 (meaning that from p_n we can reach p_0 using only the directed edges from the boundary of the LS; see Figure* 2*).*

From now on we will assume that our graph at hand has been preprocessed so that the following will hold:

- All vertices in the graph are reachable from s (the starting node), and t (the target node) can be reached from any vertex. In doing this we discard all the nodes that are irrelevant to the ESP problem between the nodes s and t. This can be done if we run BFS(s), reverse the direction of each edge, run BFS(t) and delete each node that hasn't been marked by both runs of breadth-first search algorithms.
- The graph has been tested for planarity and embedded if found planar.

3. Outline of the main algorithm. First we find one simple path from s to t. If it has even length, we stop and print it as a final solution. From now on we will assume that we are working on the nontrivial case—as a result of the first step, we have a path of odd length.

We prove that if there is at least one solution—an even path between s and t—in addition to the odd path found previously, then there exists a list superface LS with one even and one odd border and two simple node-disjoint paths P_1 and P_2, the first from s to the beginning of the list b, and the second from the end of the list e to t, such that P_1 and P_2 are also node-disjoint from the boundary of LS less $\{b,e\}$.

Next we prove that if there exists a list superface LS and two simple node-disjoint paths P_1 and P_2 as above, then there exists a *simple* list superface LS' and P_1' and P_2' with the same property. We need this because a planar graph can have an exponential number of list superfaces, but the number of simple list superfaces is polynomial in the size of the graph.

Based on these facts, we employ a modified DFS to construct all simple list superfaces in G. For each found simple list superface F we check if it has one odd and one even border; if this is the case, first we delete all the nodes on the boundary of F (except b and e) with their associated edges. Then we employ the algorithm from [7] to find node-disjoint paths between s and the beginning b of the simple list superface and between the end e of the simple list superface and t. If we find at least one such complete construct, then one solution—an even path—can be constructed as a concatenation of these two paths and one of the two borders of the simple list at hand. If in searching through all simple lists we fail to find such a construct, then it will mean that we have proved that a solution doesn't exist.

4. Basic theorem.

LEMMA 1. *Let F be a nonsimple list superface with one odd and one even boundary. Also let there be two simple node-disjoint paths, P_1 and P_2, the first from s to b (the beginning of F) and the second from e (the end of F) to t, such that P_1 and P_2 are also node-disjoint from the boundary of F less $\{b,e\}$. Then using a path P which makes F nonsimple, we can find a list superface $F' \subsetneqq F$, with one odd and one even boundary and two simple node-disjoint paths P_1' and P_2', such that P_1' and P_2' are also node-disjoint from the boundary of F' less $\{b',e'\}$, one from s to the beginning of F' and one from the end of F' to t.*

Proof. There are two general cases for the paths P_1 and P_2:

1. The two paths are from the same side with respect to the boundary of F, either outside or inside. If they are inside we can map all the nodes and edges from the outside of F inside, and vice versa. Therefore we can treat these two subcases as one—we will assume that the two paths are from the outside of F.
2. The two paths are from different sides with respect to the boundary of F, one is outside and the other inside. Again using remapping of all the nodes and edges from the outside of F inside and vice versa, we can treat these two subcases as one—we will assume that the two paths are P_1 outside and P_2 inside F.

A. Let us consider the first case for P_1 and P_2: the two paths are from the outside of F. Let the path P make F nonsimple. There are two cases for this path P with beginning p_0 and end p_n. In the first case, suppose p_0 lies on one boundary and p_n lies on the other (see Figure 3 (a)). Here there are two possible sublist superfaces of F. The two boundaries are divided into four pieces, of which one is even and three are odd, or three even and one odd. In either case there will be one sublist with subboundaries of equal parity and one with subboundaries of different parities. If P is even, we take as F' the sublist with subboundaries of different parities; if P is odd, we take the sublist with subboundaries of equal parity. One of the two paths from s to the beginning of F' and from the end of F' to t will stay the same; the other will be a continuation of the path for F with one piece of the boundary (see Figure 3). The second case where p_0 and p_n lie on the same boundary of F is similar (see Figure 3 (b)).

B. Let us consider the second case for P_1 and P_2: P_1 outside and P_2 inside F. Let path P with beginning p_0 and end p_n make F nonsimple. In this case, P and P_2

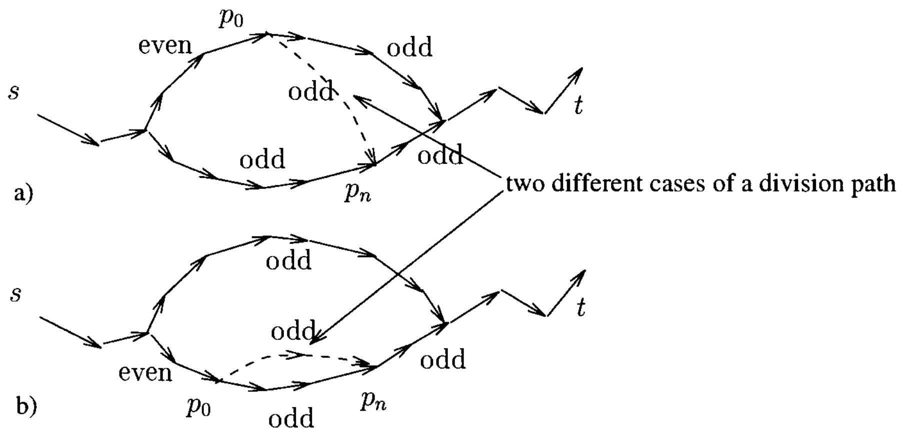


FIG. 3. *Nonsimple list and the division path.*

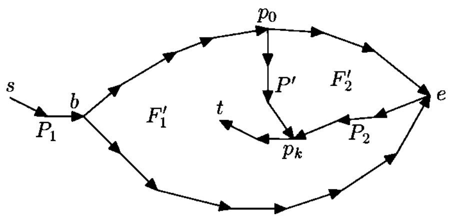


FIG. 4. *Nonsimple list with P_2 inside; a division path intersects P_2.*

are inside F. If P is node-disjoint with P_2, we repeat the logic from case A to derive P'_1, F', and P'_2. Let us assume, then, that P intersect P_2 and let us denote by P' this part of P that starts from its beginning p_0 until the first common node with P_2, p_k (see Figure 4).

Let a = *length* (the path of the upper boundary from b up to p_0). Let b = *length* (the path of the upper boundary from p_0 up to e). Let c = *length* (the lower boundary). Let x = *length* (the subpath P'). And let y = *length* (the subpath of P_2 from e to p_k). There are two possible sublists: F'_1 and F'_2 of the list F, which possibly can lead to the solution of our lemma. Let us presume for the moment that their boundaries are of equal parity. Then the following equations are correct:

$$\begin{aligned} a + b + c &= 2k_1 + 1, \\ x + b + y &= 2k_2, \\ a + x + c + y &= 2k_3. \end{aligned}$$

Adding the above three equations we get the following:

$$2(a + b + c + x + y) = 2(k_1 + k_2 + k3) + 1.$$

But as a, b, c, x, and y are whole numbers, this is impossible. It means that one of the two possible subfaces of F has boundaries with different parities. If it is F'_1, taking $P'_1 = P_1$ and $P'_2 = \{$the subpath of P_2 from p_k to $t\}$, we have a solution of the lemma. If F'_2 has boundaries with different parities, taking $P'_1 = \{P_1$ concatenated with the subpath of the upper boundary from b to $p_0\}$ and $P'_2 = \{$the subpath of P_2 from p_k to $t\}$, we have a correct solution of the lemma. $\square$

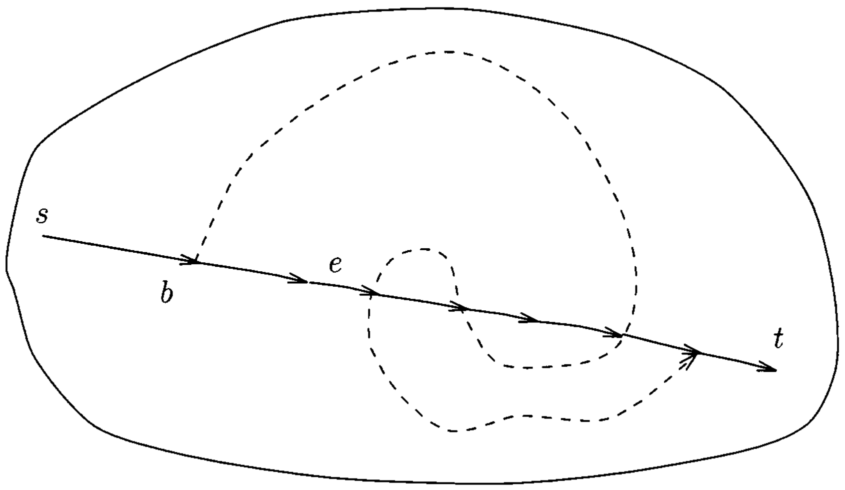


FIG. 5. *Two simple paths, one even and one odd, between s and t.*

COROLLARY. *If we have the situation of Lemma 1, we can derive $F' \subsetneq F$ with the same property as F. We can repeat this only a finite number of times, so we must eventually reach a simple list superface with the same property as F.*

LEMMA 2. *Suppose G is planar graph embedded in the plane, and s and t are two vertices from G. If there are two simple paths from s to t, one of which is even and the other odd, then there exists a simple list superface F with one odd and one even boundary, and two simple paths P_1 and P_2, one from s to the beginning of F and one from the end of F to t, such that F, P_1, and P_2 are mutually node disjoint.*

Proof. Let the even path be $s, v_1, v_2, \dots, v_k, t$ and the odd path be $s, u_1, u_2, \dots, u_l, t$. Let b be the first point lying on both paths, such that P_1 and P_2 coincide from s up to b and separate at b. Let e be the first vertex of P_1 after b which is also on P_2 (see Figure 5 ($P_1 \equiv$ solid line, $P_2 \equiv$ dashed line)).

Let us consider the length of the subpaths of P_1 and P_2 between b and e. Suppose they are of equal parity; then, because P_2 does not have any point in common with the subpath of P_1 between b and e, we can replace the subpath of P_2 between b and e with the subpath of P_1 between b and e. In other words, we can make the two paths P_1 and P_2 coincide from s to e and still have different parities.

We cannot repeat the previous step indefinitely or we will reach two paths of equal parity. Eventually we will reach a point where b and e are as above, but the subpaths of P_1 and P_2 between b and e have different parity.

Now consider the list which starts at b and ends at e, with boundaries the subpaths of P_1 and P_2 between b and e. We also have two node-disjoint paths, the coinciding part of P_1 and P_2 from s to b, and the subpath of P_2 from e to t. Now we have the situation of Lemma 1. It follows that we must have a simple list superface F with one odd and one even boundary, and two simple node-disjoint paths P_1' and P_2', one from s to b', the beginning of that simple list superface, and one from e', the end of the list, to t. It is also true that these two paths are node-disjoint from $F - b' - e'$. □

5. Finding all simple lists. The DFS algorithm [8], [1] does not specify the order in which the neighbors of a given vertex are to be visited. For a planar em-

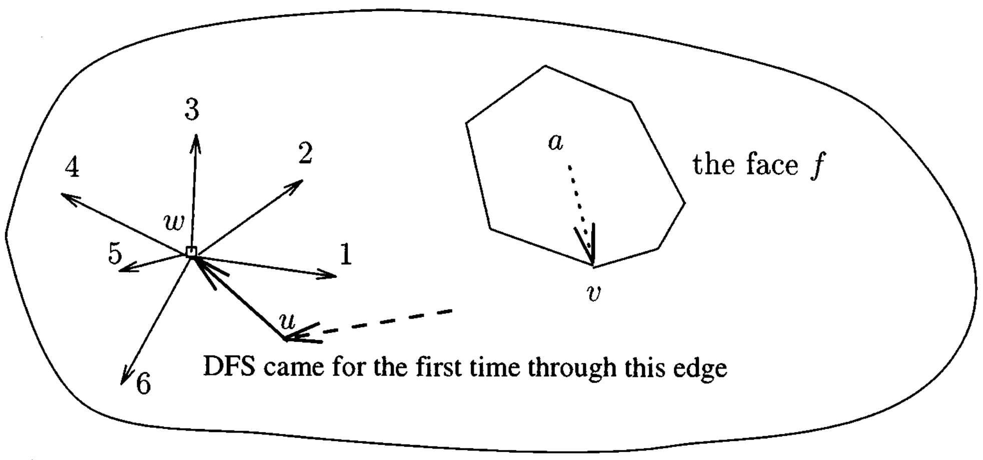


FIG. 6. *A dynamically introduced order during the DFS of all outgoing edges.*

bedding of a planar graph G, we introduce a *right depth-first search* (RDFS) and *left depth-first search* (LDFS), depending initially on a starting vertex v and an elementary face f which must have v on its boundary.

DEFINITION 4. *First we pick any point a from the interior of f and connect it to v through a new edge $a \mapsto v$. Since a planar embedding means that the edges incident with every vertex can be sorted clockwise or counterclockwise, we reorder all outgoing edges from v in counterclockwise manner starting with the first outgoing edge counterclockwise from $a \mapsto v$. Now we start a DFS of G from v using the new order from above. Whenever the DFS visits a new vertex w for the first time through the edge $u \mapsto w$, we reorder all outgoing edges from w counterclockwise from this edge (see Figure 6). After this reordering is done, we continue the DFS in a normal way using this new order. This modification of DFS we will call LDFS (v,f).*

RDFS (v, f) is defined similarly, using the clockwise order.

It is easy to see that the price paid for the dynamic reordering of the outgoing edges in each vertex is not significant. Let $m_1, m_2, \ldots, m_n$ (note that $\sum_{i=1}^{n} m_i = m$, the number of all edges in G) be the number of edges incident with every node of G. Then if we perform only a binary search and the "reordering" is done by showing only the position of the first clockwise outgoing edge in the old double-linked adjacency list at each vertex of G, the running time of the modified DFS is

$$T \le T_{DFS} + \log_2 m_1 + \log_2 m_2 + \cdots + \log_2 m_n \le T_{DFS} + \log_2 \left(\prod_{i=1}^{n} m_i \right)$$

$$\le T_{DFS} + \log_2 \left(\left(\sum_{i=1}^{n} m_i \right) / n \right)^n \le T_{DFS} + n \log_2((3n-6)/n) \le O(n).$$

Here we used the inequality $\prod_{i=1}^{n} m_i \le (\sum_{i=1}^{n} m_i)/n)^n$ and the fact that the sum of all edges in a planar graph is less than or equal to $3n - 6$.

We now apply the modified DFS in the following algorithm.

```
1.    ALGORITHM A: Finding_all_simple_list_superfaces (G); /*G is planar digraph*/
2.    begin  for each edge $v \mapsto u$ in the graph $G$ do
```

```
                /* There are two elementary faces having $v \mapsto u$ on their
                        boundaries.*/
                /* We use the face $f$ to guarantee the uniqueness of the simple face
                        output in line 12*/
3.              choose the face $f$ that contains the edge $v \mapsto u$ in the clockwise
                        direction;
4.              build $LDFSTree(v, f)$ with the condition that $v \mapsto u$ is the only
                        outgoing edge from $v$;
                  /* all other outgoing edges are temporarily deleted until the end
                        of this line */
5.              build $RDFSTree(v, f)$;
6.              for each vertex $w \in G$, $w \neq v$ do
7.                  if $w \in$ both of the above trees then
8.                      find the path $P_1$ from $v$ to $w$ made up of vertices in the first
                              tree;
9.                      find the path $P_2$ from $v$ to $w$ made up of vertices in the
                              second tree;
10.                     if $P_1 \cap P_2 = \{v, w\}$ && the face $f \in$ the finite region between
                                    $P_1$ and $P_2$ then
11.                         print $P_1 \cup P_2$ as a simple list superface;
                        end if ;
                    end if ;
                end for ;
            end for ;
      end;
```

LEMMA 3. *Algorithm* A *gives us all simple list superfaces in G, and the number of simple list superfaces is polynomial in the number of vertices of G.*

Proof. First we will prove that the output of Algorithm A is a simple list superface. Consider the output of line 12. From line 11 we have $P_1 \cap P_2 = \{v, w\}$ and face $f \in$ the finite region between P_1 and P_2. Let us assume that this list superface is not simple. But then there must be a directed path $P = p_0, p_1, \ldots, p_n$ where p_0 and p_n lie on the boundary and $p_1, \ldots, p_{n-1}$ lie inside, such that p_n is not a predecessor of p_0. There are two possible cases (see Figure 7):

a. p_0 and end p_n lie on opposite boundaries of F;

b. p_0 is a predecessor of p_n on the same boundary.

Without loss of generality, we suppose that p_0 lies on P_1. Let the first vertex after p_0 be p_1 on P and p' on P_1. In Algorithm A, the path P_1 consists of tree arcs formed during $LDFS(v, f)$, so the arc $p_0 \mapsto p_1$ will be traversed before the arc $p_0 \mapsto p'$ because of the counterclockwise order induced on the outgoing arcs from p_0. There is now a path from p_1 to w in the first case and from p_1 to p_n in the second case. But then the DFS must have visited w or p_n already when it finished with p_1 and before it returned to p_0 (see [1]). Therefore, when the arc $p_0 \mapsto p'$ was traversed later, the algorithm would not have repeated its visit to w (in the first case) or p_n (in the second case).

This contradiction means that the output of line 12 of Algorithm A is a simple list superface.

Second, let's take any simple list superface F in G and prove that it will be found by Algorithm A.

Let F begin at b and end at e, and let b_1 and b_2 be the neighbors of b on each

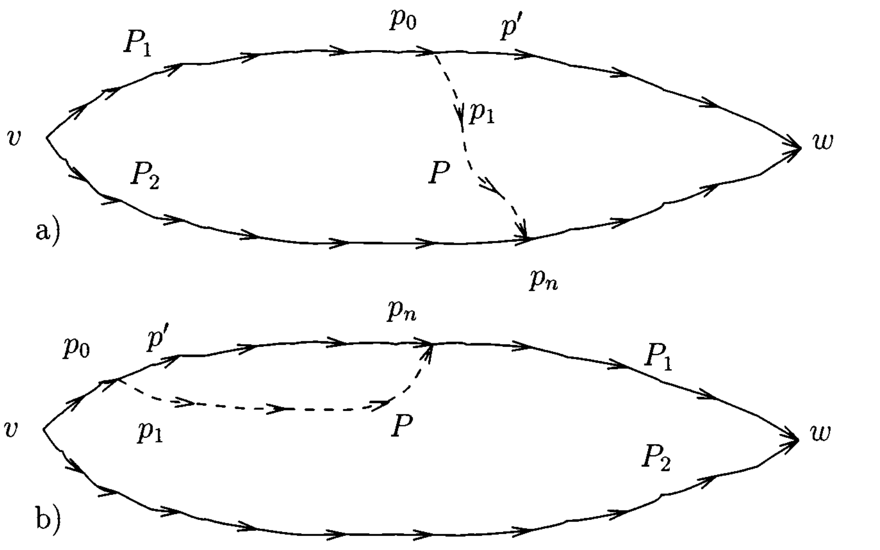


FIG. 7. *If we assume that F is not simple, there are two possible paths which split F.*

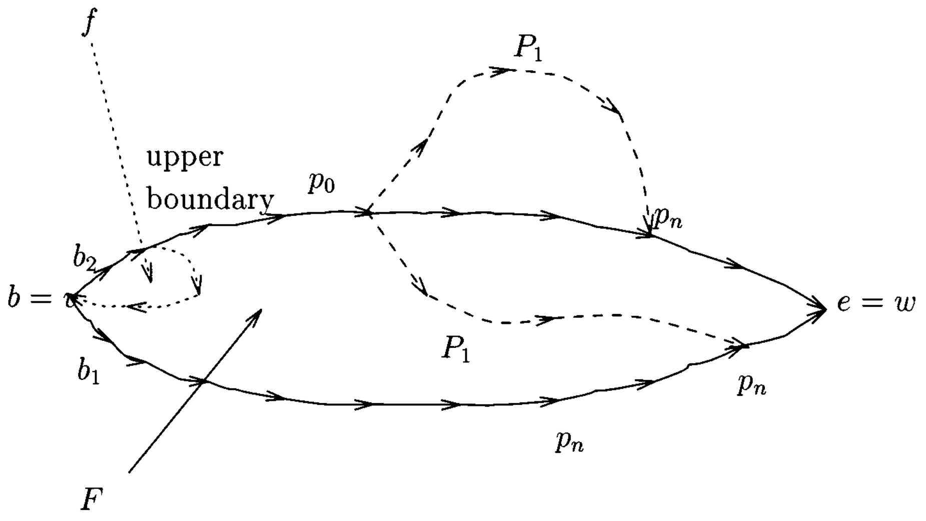


FIG. 8. *We assume that the simple list superface F is not generated by Algorithm* A.

of the two boundaries of F, ordered counterclockwise. Let f be the elementary face inside F which has $b \mapsto b_2$ on its boundary (see Figure 8).

Suppose that Algorithm A reaches b in line 2, $b \mapsto b_2$ in line 3, and e in line 7 and that the paths found in lines 9 and 10 are such that P_1 is not the upper boundary of F (the boundary which contains b_2) or P_2 is not the lower boundary of F.

Let p_0 be the first vertex where the upper boundary separates from the path from $LDFSTree(b, f)$ from b to e. There are two possibilities for P_1:

1. to go counterclockwise from the upper boundary,
2. to go clockwise from the upper boundary.

In the first case let p_n be the vertex where P_1 and the upper boundary meet again.

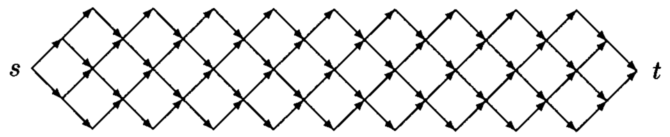


FIG. 9. *An example of exponential number of list superfaces ($l = 11$).*

As in the first part of the proof, p_n will be traversed by LDFS before considering the subpath of P_1 after p_0—which is a contradiction. In the second case, p_n can't be a predecessor of p_0 because P_1 is a path from a tree and therefore simple. Thus p_n might be an ancestor of p_0 on the same boundary or lie on the opposite boundary, but both of these cases violate the fact that F is simple.

We have proved that the upper boundary and P_1 must be identical. The proof that the lower boundary of F is P_2, the path found by $RDFSTree(b, f)$ between b and e, is similar. ∎

LEMMA 4. *The number of simple list superfaces is polynomial.*

Proof. The number of iterations of the outmost loop (line 2) will not be more that the number of arcs in G. The innermost loop (line 6) will not execute more than n times on each round of the outmost loop, where n is the number of vertices. Inside the innermost loop each line will take at most $O(n)$ time. So Algorithm A will be $O(m * n * n)$, which is $O(n^3)$ for planar graphs. ∎

Example. It is not easy to find a case where there is an exponential number of list superfaces with start vertex s and end vertex t. In Figure 9 the graph has the number of vertices $\geq 5l - 6$, where l is a parameter. In a list superface from s to t, either the top or the bottom square or both (3 variant) from each vertical pair (one top and one bottom square form one vertical pair) must be present. Thus there are at least 3^{l-2} list superfaces, none of which is simple, starting at s and ending at t. If Algorithm A is run with this example, all these superfaces will be discarded because of the violation of the first condition in line 10 of Algorithm A.

6. Final algorithm.

We are ready to formulate and prove an algorithm to find an even simple path between vertices s and t in a planar digraph G.

```
1.  ALGORITHM B: Find_even_simple_path_in_planar_digraph(G);
2.  Find one simple directed path from s to t; if there is none, go to line 12;
3.  If the path found at the previous step is even, then print the solution and exit;
4.  Use Algorithm A(G) to generate all simple list superfaces and feed them one
        by one to the next line;
5.  for each simple list superface F with start b and end e in G do
6.      if F has one even and one odd boundary then
7.          delete temporarily until end if all the nodes from the boundary of F,
                except b and e;
8.          run the polynomial algorithm from [7] to find two node-disjoint paths:
                one from s to b, and one from e to t;
9.          if there is a solution for the latest task then
10.             there is final solution—concatenation of these three pieces:
                    a) the path, found in line 8, from s to b
                    b) the subboundary of F, that is, of the same parity as
                       the sum of the two paths from line 8
                    c) the path from e to t, found in line 8
11.             print this solution and exit;
```

```
            end if ;
        end if ;
    end for ;
12. print THERE IS NO SOLUTION!;
    end;
```

LEMMA 5. *Algorithm* B *is correct and is polynomial in the size of the graph G.*

Proof. Step 2 is no more than an algorithm for the shortest distance between s and t. For step 4, we use the result from Lemma 3 that Algorithm A is polynomial and that the number of all simple list superfaces in a planar graph is polynomial. This means that the **for** cycle starting in line 5 executes a polynomial number of times. Since all steps in the body of the **for** loop are polynomial (for line 8, see [7]), and there is a polynomial number of iterations, the whole of Algorithm B is polynomial. □

We haven't tried to calculate any estimation for the running time of Algorithm B because it refers to the algorithm from [7], which has had only theoretical value up to now.

7. Conclusion.

As a result of Lemma 5 we can state that the query $R = (\_ \, \_)*$ from RSP problem (described in the introduction) is polynomially solvable for planar graphs. It is an open problem to find which other queries or classes of queries allow a polynomial solution to the RSP problem in planar digraphs and digraphs embedded onto orientable surfaces. Also there is a need for efficient polynomial algorithms for them. It would also be interesting to investigate which classes of queries remain NP-complete for planar graphs. Furthermore, it would be interesting if there exists a more efficient algorithm than that presented here for finding all simple list superfaces in a directed planar graph.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. FORTUNE, J. HOPCROFT, AND J. WYLLIE, *The directed subgraph homeomorphism problem*, Theoret. Comput. Sci., 10 (1980), pp. 111–121.

[3] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[4] M.-Y. KAO AND P. N. KLEIN, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, J. Comput. System Sci., 47 (1993), pp. 459–500.

[5] A. LAPAUGH AND C. PAPADIMITRIOU, *The even-path problem for graphs and digraphs*, Networks, 14 (1984), pp. 507–513.

[6] A. O. MENDELZON AND P. T. WOOD, *Finding regular simple paths in graph databases*, in Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, The Netherlands, Morgan Kaufmann, Palo Alto, 1989, pp. 185–193.

[7] A. SCHRIJVER, *Finding k disjoint paths in a directed planar graph*, SIAM J. Comput., 23 (1994), pp. 780–788.

[8] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 3, pp. 697–711

THE ANGULAR-METRIC TRAVELING SALESMAN PROBLEM*

ALOK AGGARWAL†, DON COPPERSMITH†, SANJEEV KHANNA‡,
RAJEEV MOTWANI§, AND BARUCH SCHIEBER†

Abstract. Motivated by applications in robotics, we formulate the problem of minimizing the total angle cost of a TSP tour for a set of points in Euclidean space, where the angle cost of a tour is the sum of the direction changes at the points. We establish the NP-hardness of both this problem and its relaxation to the cycle cover problem. We then consider the issue of designing approximation algorithms for these problems and show that both problems can be approximated to within a ratio of $O(\log n)$ in polynomial time. We also consider the problem of simultaneously approximating both the angle and the length measure for a TSP tour. In studying the resulting tradeoff, we choose to focus on the sum of the two performance ratios and provide tight bounds on the sum. Finally, we consider the extremal value of the angle measure and obtain essentially tight bounds for it. In this paper we restrict our attention to the *planar* setting, but all our results are easily extended to higher dimensions.

Key words. angle metric, approximation algorithms, combinatorial optimization, computational complexity, extremal point sets, robotics, traveling salesman problem

AMS subject classification. 68Q25

PII. S0097539796312721

1. Introduction.

The traveling salesman problem (TSP) is one of the most widely studied combinatorial optimization problem, to the extent that there is an entire book [16] devoted to it. The importance of this problem is in large part due to the richness of its structure, which has led to the development of a variety of general paradigms and techniques for dealing with optimization problems. We further explore this structure by formulating a natural version of the TSP problem that does not seem to have been studied earlier in the literature [16]. The new problem, called the Angle-TSP problem, seeks to minimize the total angle of a TSP tour for a set of points in Euclidean space, where the angle of a tour is the sum of the direction changes at the points. We establish the NP-hardness of this problem and the related cycle-cover problem, provide approximation algorithms for both problems, study their extremal properties, and give tight bounds on the trade-off between the angular and the length performance ratio achievable by a TSP tour.

Formally, we define the optimization problem *angle-TSP* as follows: given a collection $P = \{p_1, \ldots, p_n\}$ of n points in Euclidean space, find a tour of the points in

*Received by the editors November 27, 1996; accepted for publication (in revised form) November 9, 1998; published electronically December 7, 1999. A preliminary version of this paper appeared in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, 1997, pp. 221–229.

http://www.siam.org/journals/sicomp/29-3/31272.html

†IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598 (aggarwa@watson.ibm.com, copper@watson.ibm.com, sbar@watson.ibm.com).

‡Fundamental Mathematics Research Department, Bell Labs, 700 Mountain Avenue, Murray Hill, NJ 07974 (sanjeev@research.bell-labs.com). This work was performed while the author was at the Department of Computer Science, Stanford University, Stanford, CA and was supported by a Schlumberger Foundation Fellowship, an OTL grant, and NSF grant CCR-9357849.

§Department of Computer Science, Stanford University, Stanford, CA 94305 (rajeev@cs.stanford.edu). This author was supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI grant DAAH04-96-1-0007, and an NSF Young Investigator Award CCR-9357849, with matching funds from IBM, the Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

P with minimum total angle. For two edges (u, v) and (v, w) incident on a vertex v, we define the angle subtended by them as the angle between the vectors $\vec{uv}$ and $\vec{vw}$. Equivalently, this angle is the absolute value of the change in the direction of motion when traveling from u to w via v, along these edges. Note that the angle always lies in the interval $[0, \pi]$. For a path or a cycle on a subset of P, the total angle subtended is defined as the sum over all pairs of adjacent edges of the angle subtended by the pair. We also define an angular-metric version of the cycle cover problem (CCP) called **Angle-CCP:** given a collection $P = \{p_1, \ldots, p_n\}$ of n points in the Euclidean plane, find a cycle cover of the points in P subtending the minimum possible total angle. As will become clear shortly, unlike the length-metric cycle cover problem that is relatively easy, the Angle-CCP problem is as hard as the angle-TSP problem itself.

Our main results are as follows. We show that the Angle-TSP and the Angle-CCP problem are both NP-hard. Given this, we turn our attention to designing approximation algorithms for these problems and show that both problems can be approximated to within a ratio of $O(\log n)$ in polynomial time. It should be noted that the Angle-TSP appears easier than Angle-CCP since an approximation algorithm for Angle-CCP would imply a similar approximation for Angle-TSP but the reverse is not clear. We also consider the problem of simultaneously approximating both the angle and the length measure for a TSP tour. In studying the tradeoff between the angle and the length metric, we choose to focus on the sum of the two performance ratios and provide tight bounds on the sum. Finally, we consider the extremal value of the angle measure and obtain essentially tight bounds for this too. While we state our results in context of *planar* angle-TSP, all our results are easily extended to higher dimensions.

Our original motivation was a problem posed by Guibas, namely, to find a "smooth" tour that visits a set of prescribed viewpoints of an object [12]. The smoothness criteria arises naturally in the context of nonholonomic robots [2, 3, 14]. Of late, there has been considerable interest in motion planning with "curvature constraints" (for example, see Agarwal, Raghavan, and Tamaki [1]). This is motivated by nonholonomic motion planning for steering-constrained robots that have stops on the steering mechanisms limiting the rate of change of direction [4, 5, 10, 13, 15]. Our problem involves a different notion of smoothness, but it could be used as a preprocessing step for planning curvature-constrained paths as suggested by Fraichard [10]. Our notion is of direct relevance in situations where a rotation by a robot is significantly more expensive than a translation. For instance, this would be the case for robots with high inertia, or where relatively large mechanical errors during rotation would necessitate error-detection and correction process via localization [14]. Also, there are applications in planning the trajectories of high-speed aircraft [8].

The angle-TSP is a special case of TSP problems where the cost of an edge depends on the edges traversed earlier in the tour. Several applications can be modeled by such TSP problems. For example, the problem of scheduling orders in a production line can be modeled as a TSP problem where the vertices represent orders that have to be scheduled and edges represent set-up costs in moving from one order to another. In applications such as VLSI fabrication, steel mills and chemical plants scheduling, the set-up cost of chemical cleansing and temperature control depend not only on the previous order, but on several previous orders. This translates to a TSP problem where the cost of an edge depends on previous edges.

Our work suggests many interesting and challenging directions for future work.

Can our approximation ratio of $O(\log n)$ be improved to a constant, or be matched by a complementary hardness result? What can be said about other forms of trade-off such as the product of the angle and the length measures? It is also interesting to consider other natural variants of the basic angle-TSP problem; for example, the variant where the maximum angle change at a vertex is bounded and the goal is to minimize the length measure.

The rest of this paper is organized as follows. We begin in Section 2 by showing that there are polynomial time $O(\log n)$-approximation algorithms for both angle-TSP and Angle-CCP. In Section 3, we establish essentially tight bounds on the extremal cost of Angle-TSP. Building on the latter result, in Section 4, we establish tight bounds on the ability of a polynomial time algorithm to simultaneously approximate both the optimal angle-TSP and the optimal Length-TSP. Finally, in Section 5, we present the proof of NP-hardness of angle-TSP and Angle-CCP.

2. Approximation algorithms for angle-TSP and angle-CCP.

In this section, we describe an $O(\log n)$-approximation algorithm for the angle-TSP problem on an instance P containing n points, with a running time polynomial in n. But first we need the following lemmas.

LEMMA 2.1. *Suppose there is a cycle C on a set of points P in the plane that subtends a total angle of α. Then, for any subset $P' \subset P$, there exists a cycle C' subtending a total angle at most α.*

The proof of Lemma 2.1 is straightforward: using short-cuts to eliminate the points of $P \setminus P'$ from C, we obtain a cycle C' of no larger total angle by the triangle inequality for angles.

LEMMA 2.2. *There is a polynomial-time algorithm that finds a maximum cardinality path on the points in P that subtends a total angle at most π.*

Proof. We use dynamic programming similar to Boyce *et al.* [6]. Fix some directed edge s in P and define the array A_s as follows: for each directed edge e in P and for $1 \leq k \leq n$, the array entry $A_s[e, k]$ contains the minimum angle subtended by a path of cardinality k starting with the edge s and ending at edge e; by convention, the entry is ∞ if there is no such path of cost at most π. We claim that the array entries for paths of cardinality k can be computed from the array entries for paths of cardinality $k-1$ in polynomial time, as follows: for any directed edge e and any edge f directed into the source point of e, let $\theta_{f,e}$ be the angle between f and e; then, $A_s[e,k] \leftarrow \min_f \{A_s[f,k-1] + \theta_{f,e}\}$, and $A_s[e,k]$ is set to ∞ if it exceeds π. To verify the correctness of this computation, observe that in a path with total angle at most π, neither vertices nor edges can repeat.

It is fairly easy to modify the preceding construction to store actual paths, rather than the angle subtended by them. The maximum cardinality path starting from the edge s with total subtended angle at most π can be determined by finding the largest k for which there exists an edge e such that $A_s[e,k]$ is finite. Repeating this for all choices of the starting edge s gives the desired result. It is easy to verify that this computation can be performed in polynomial time. □

We now establish the desired result.

THEOREM 2.3. *There is a polynomial-time algorithm which gives $O(\log n)$-approximation for angle-TSP.*

Proof. Suppose that we are given an instance P consisting of n points $\{p_1, \ldots, p_n\}$ in the plane. Let τ^* be the minimum angle tour for P, where $\theta = \theta_{opt}(P)$ is the total angle change in the optimal cycle. Define $K = \lceil \theta/\pi \rceil$, clearly $K \geq 2$. It follows that the tour τ^* contains a subpath of cardinality at least n/K which subtends a net angle change at most π.

We now claim that for any instance P with n points with $\theta_{opt}(P) \leq K\pi$, it is possible to compute in polynomial time, a cycle cover $\mathcal{C} = \{C_1, \ldots, C_r\}$ for P with the following properties: $r = K \log n$; and, each cycle C_i subtends a total angle at most 3π. It can be verified that patching together any two cycles C and C' subtending total angles α and α' gives a single cycle C'' subtending an angle $\alpha'' \leq \alpha + \alpha' + 2\pi$. From this, it follows that repeatedly patching together the cycles in $\mathcal{C}$ gives a tour τ with $\theta(\tau, P) = O(\pi K \log n)$. Since $\theta_{opt}(P) = \Omega(\pi K)$, it follows that $R_\theta(\tau, P) = O(\log n)$, giving the desired result.

It remains to validate the polynomial-time construction of the claimed cycle cover. We do so by induction on n. The base case is trivial and is omitted. For the induction step, first observe that by Lemma 2.2, we can find a path of cardinality at least n/K in P with total angle at most π. Joining together the two endpoints of this path gives a cycle C_1 subtending a total angle at most 3π. Consider the instance P' with n' points obtained by deleting the points in C_1 from P. From Lemma 2.1, it follows that $\theta_{opt}(P') \leq \theta_{opt}(P) \leq K\pi$. Since $n' \leq n(1 - 1/K)$ and $K \log n' \leq K \log n - 1$, we obtain from the induction hypothesis that it is possible to recursively compute in polynomial time a suitable cycle cover for P' consisting at most $K \log n - 1$ cycles. Adding C_1 to this collection of cycles gives the desired cycle cover for P. $\square$

We note that the same proof applies also to the Angle-CCP problem.

COROLLARY 2.4. *There is a polynomial-time algorithm which gives $O(\log n)$-approximation for* Angle-CCP.

3. The extremal cost of angle-TSP.

In this section, we address the question: What is the extremal cost of an optimal angle-TSP as a function of n? While this question is of intrinsic interest in its own right, we will see in Section 4 that it also enables us to analyze the trade-off between the performance ratio for the angle and the length metrics.

Note that for any n points in the plane there is a solution for the angle-TSP of cost $O(n)$. We improve this upper bound and show that for any n points in the plane there is a solution for the angle-TSP of cost $O(n/\log n)$. On the other hand, we are able to match this bound by showing the existence of an instance for which the optimal cost is $\Omega(n/\log n)$.

THEOREM 3.1. *Given a set of n points in the plane, there exists a solution for the angle-TSP problem of cost $O(n/\log n)$.*

Proof. Our proof relies on the following classic result of Erdős and Szekeres [9]: Given a set of n points in general position, there exists a subset of size $\Omega(\log n)$ which induces a convex polygon. This means that the entire point set can be covered by $O(n/\log n)$ convex polygons. Since traversing each polygon has a cost of 2π and linking two polygons costs at most another 2π, we can construct a tour that visits all the n points at a total cost of $O(n/\log n)$. This establishes the claimed result. As an aside, one may note that the claimed $\Omega(\log n)$ size convex polygon can be found in polynomial time by a simple dynamic programming. $\square$

We now establish a matching lower bound on the extremal cost.

THEOREM 3.2. *There exists an instance P of angle-TSP with n points in the plane such that the optimal cost for this instance is $\Omega(n/\log n)$.*

Proof. We describe a recursive construction that achieves this lower bound. Assume that $n = 2^k$, for some integer k. In the construction, we have $k = \log n$ levels of recursion. Define an instance of level i to be the instance constructed after i levels of recursion, and note that the final instance is the instance of level k. The instance of level 0 simply contains one point. The instance of level $i+1$ consists of two instances of level i at distance 4^i from each other such that the line connecting the centroids[1] of these two instances makes an angle of $i\pi/k$ with the X axis. We say that these two instances are *aligned* at angle $i\pi/k$. Note that the instance of level k has $2^k = n$ instances of level 0, and in general 2^{k-i} instances of level i.

Consider any solution to angle-TSP on this instance. To prove the lower bound we show that the cost of traversing any three consecutive points in this solution is $\Omega(\pi/k)$. This readily implies the desired bound.

Fix any three consecutive points, and let i be the maximum level such that these three points do not belong to the same instance of level i. Clearly, such an i exists since the instance of level one consists of only two points. Note that these three points belong to the same single instance of level $i+1$.

It is not difficult to verify that the cost of traversing points in the two instances is $\Omega(\pi/k)$, since: (i) the two instances of level i are aligned at angle $i\pi/k$; (ii) all the instances of lower levels are aligned at an angle that is at most $(i-1)\pi/k$; and, (iii) the distance between points in different instances of level i is at least twice the distance between points in the same instance of level i. □

4. Simultaneously approximating length and angle. In this section we discuss the possibility of *simultaneously* obtaining a good approximation to both the angle measure and the length measure for planar TSP. Fix an instance P and consider any tour τ for it. Denote by $\theta(\tau, P)$ the total angle subtended by τ, and by $\ell(\tau, P)$ the total length of τ. Further, let $\theta_{opt}(P)$ be the angle subtended by the optimal angle-TSP solution for P, and $\ell_{opt}(P)$ be the total length of the optimal Length-TSP solution for P. Note that the optimal angle-TSP and the optimal Length-TSP solutions for P need not be the same. We define the performance ratio of τ with respect to the two measures as:

$$R_\theta(\tau, P) = \frac{\theta(\tau, P)}{\theta_{opt}(P)} \qquad \text{and} \qquad R_\ell(\tau, P) = \frac{\ell(\tau, P)}{\ell_{opt}(P)}.$$

Our goal is to study the tradeoff between the two ratios $R_\theta(\tau, P)$ and $R_\ell(\tau, P)$ over all possible TSP solutions τ.

THEOREM 4.1. *There exists an instance P consisting of n points in the plane, such that for any tour τ on P,*

$$R_\theta(\tau, P) + R_\ell(\tau, P) = \Omega\left(\sqrt{n/\log n}\right).$$

Proof. Consider the following scenario in the plane. Fix $t = \sqrt{n \log n}$ and let $P_1, \ldots, P_t$ be a collection of identical instances described in the proof of Theorem 3.2 each with $\sqrt{n/\log n}$ points. Scale these instances so that the length of the path connecting the points in each instance is of unit length. We arrange these instances in a vertical stack so that they are perfectly aligned, and two successive instances in the vertical sequence are at a unit distance from each other. We refer to this entire set of points as the instance P.

We claim that: (a) $\ell_{opt}(P) = O(\sqrt{n \log n})$; and, (b) $\theta_{opt}(P) = O(\sqrt{n/\log n})$. To see (a), observe that by our scaling, there exists a Hamiltonian path of unit length in each of the instances P_i; also, since the optimal Length-TSP tour can visit the

[1] The centroid of a finite set of points is their arithmetic mean.

instances in the vertical order, the total cost of traveling between the series of $\sqrt{n \log n}$ instances is $O(\sqrt{n \log n})$. To see (b), observe that a set of $\sqrt{n/\log n}$ straight lines cover all points in P and these can be stitched together into a tour using $O(\sqrt{n/\log n})$ changes in the direction of travel.

Fix any tour τ for P. Consider any three consecutive points in τ. If these points are in the same instance P_i then, by the proof of Theorem 3.2, the *angular* cost of traversing them is $\Omega(1/\log n)$. If these points belong to at least two different instances then, because of the spacing between instances, the *length* cost of traversing them is $\Omega(1)$. Consider the $\lfloor n/2 \rfloor$ consecutive triples. We either incur the angular cost for at least half of them ($\geq \lfloor n/4 \rfloor$), or we incur the length cost for at least half of them. We conclude that $\theta(\tau, P) = \Omega(n/\log n)$ or $\ell(\tau, P) = \Omega(n)$. Combining this with the upper bounds on $\ell_{opt}(P)$ and $\theta_{opt}(P)$, we conclude that $R_\theta(\tau, P) + R_\ell(\tau, P) = \Omega(\sqrt{n/\log n})$. $\square$

We can show that there is a polynomial-time algorithm that matches the lower bound to within a factor of $\sqrt{\log n}$. We first show a marginally weaker upper bound that is considerably easier to obtain.

THEOREM 4.2. *Given a set P of n points in the plane, there is a polynomial-time algorithm for constructing a tour τ on P such that: $R_\theta(\tau, P) + R_\ell(\tau, P) = O(\sqrt{n \log n})$.*

Proof. To find the desired tour we first compute an $O(\log n)$-approximation to the optimal angle-TSP tour by using the approximation algorithm described in Section 2. Denote the resulting tour by τ. We distinguish between two cases.

Case 1. $\theta(\tau, P) = \Omega(\sqrt{n \log n})$. In this case compute a constant approximation to the Length-TSP using, e.g., Christofides' approximation [7] for TSP. Denote the resulting tour by τ'. We claim that $R_\theta(\tau', P) + R_\ell(\tau', P) = O(\sqrt{n \log n})$. Clearly, $R_\ell(\tau', P) = O(1)$. Since $\theta(\tau, P) = \Omega(\sqrt{n \log n})$, we know that $\theta_{opt}(P) = \Omega(\sqrt{n/\log n})$. Also $\theta(\tau', P) = O(n)$. Hence $R_\theta(\tau', P) = O(\sqrt{n \log n})$.

Case 2. $\theta(\tau, P) = O(\sqrt{n \log n})$. In this case $R_\theta(\tau, P) = O(\log n)$. We claim that in $R_\ell(\tau, P) = O(\sqrt{n \log n})$. To see this we re-examine the approximation algorithm for the angle-TSP problem. Recall that in this algorithm we find K' paths each of which subtends an angle of at most π that cover all the n points. Note that in our case $K' = O(\sqrt{n \log n})$. Below, we show that the length of each such path is a lower bound on $\ell_{opt}(P)$. Since there are $O(\sqrt{n \log n})$ such paths this implies that $R_\ell(\tau, P) = O(\sqrt{n \log n})$. (Note that the stitching of these paths costs $O(\sqrt{n \log n}\ell_{opt}(P))$.) Consider such a path Q with endpoints u and v. Clearly, the distance between u and v is a lower bound on $\ell_{opt}(P)$. We show that the length of Q is at most $4\sqrt{2}$ times this length. Partition Q into four segments, such that each segment subtends an angle of at most $\pi/4$. We claim that the length of each such segment is $\sqrt{2}$ times the distance from u to v. Consider such a segment. The length of it is just the sum of the lengths of its edges. Consider the projections of all these edges on the straight line connecting u and v. The sum of the lengths of these projections is the distance between u and v. Since each edge has an angle of at most $\pi/4$ with this straight line, the projection of each edge is at least $\sqrt{2}/2$ of its length. $\square$

The proof of the following tighter upper bounds is much more involved.

THEOREM 4.3. *Given a set P of n points in the plane, there is a polynomial-time algorithm for constructing a tour τ on P for which $R_\theta(\tau, P) + R_\ell(\tau, P)$ is bounded as follows:*

(1) *if* $\theta_{opt}(P) \leq \frac{\sqrt{n}}{\log n}$, *then*

$$R_\theta(\tau, P) + R_\ell(\tau, P) = O\left(\theta_{opt}(P) \log\left(\frac{n}{\theta_{opt}(P)}\right)\right)$$

(2) *if* $\frac{\sqrt{n}}{\log n} \leq \theta_{opt}(P) \leq \sqrt{n \log n}$, *then*

$$R_\theta(\tau, P) + R_\ell(\tau, P) = O\left(\sqrt{\frac{n}{\log n}}\right)$$

(3) *if* $\theta_{opt}(P) \geq \sqrt{n \log n}$, *then*

$$R_\theta(\tau, P) + R_\ell(\tau, P) = O\left(\frac{n}{\theta_{opt}(P)}\right).$$

Thus the worst cast sum of the ratios is $O(\sqrt{n})$.

Proof. To find the desired tour, τ, we use two approximation algorithms: (a) the approximation algorithm for the angle-TSP problem given in Section 2; and, (b) a constant-factor approximation to the Length-TSP in the plane, e.g., Christofides' approximation [7] for TSP.

We begin by finding a constant-factor approximation to the Length-TSP. Let L be the resulting tour. Fix a parameter k; the value of k will be determined later.

The next stage consists of $m = c \max\{\lceil \log(k/\theta_{opt}(P)) \rceil, 0\}$ iterations, for some constant $c > 1$. Note that since we can approximate $\theta_{opt}(P)$ up to a logarithmic factor, we can compute the number of iterations.

In each iteration, we find sub-paths that cover at least 3/4 of the points that still need to be covered and concatenate them in to the desired tour τ. Let P_i be the set of uncovered points at the beginning of iteration i, for $i = 0, ..., m-1$. Let L_i be the sub-tour of L induced by P_i. In iteration i we break L_i into a minimal number of sub-paths each containing at most $k/2^i$ points. Note that the number of such sub-paths is at most $n/(k2^i)$ (since the $|P_i| \leq n/4^i$). Denote these sub-paths by $L_{i,j}$, and the points on $L_{i,j}$ by $P_{i,j}$, for $1 \leq j \leq n/(k2^i)$. In each subset $P_{i,j}$ we use the approximation algorithm of Section 2 to find a maximal path which subtends an angle of at most π. We stitch all these paths in the order of their appearance in L, and concatenate the resulting path to the desired path τ. We repeat this step until at least 3/4 of the points of P_i are covered.

After terminating the iterations, we concatenate the sub-tour of L induced by the points that are still uncovered to τ.

We now bound $\theta(\tau, P)$ and $\ell(\tau, P)$; we begin with $\theta(\tau, P)$. The desired path τ is the concatenation of the paths computed in the iterations and the final path. The final path consists $O(n/4^m) = O(n/(k2^m)\theta_{opt}(P))$ points and hence its angular cost is $O(n/(k2^m)\theta_{opt}(P))$. Consider a path concatenated to τ in iteration i. Recall that this path is given by stitching the maximal paths computed in each of $P_{i,j}$. Since the angular cost of each maximal path is at most π, the angular cost of the path concatenated to τ is $O(n/(k2^i))$. Similar to the proof given in Section 2 it can be shown that the number of maximal paths which subtends an angle of at most π required to cover at least 3/4 of the points in the set $P_{i,j}$ is $O(\theta_{opt}(P_{i,j}))$. Hence the number of paths (which are given by stitching the maximal paths computed in each of $P_{i,j}$) required to cover at least 3/4 of the points in the set P_i is $O(\max_j\{\theta_{opt}(P_{i,j})\}) = O(\theta_{opt}(P))$. Summing over the m iterations and adding the angular cost of the final path, we conclude that $R_\theta(\tau, P) = O(\sum_{i=0}^{m} n/(k2^i)) = O(n/k)$. Since $\theta(\tau, P) = O(n)$, we have $R_\theta(\tau, P) = O(\min\{n/k, n/\theta_{opt}(P)\})$.

Next, we bound $\ell(\tau, P)$. The desired path τ is the concatenation of the paths computed in the iterations and the final path. We claim that the length of each such path is $O(\ell_{opt}(P))$. This is clearly true for the last path concatenated to τ since this path is a sub-tour of L and L is a constant-factor approximation to the Length-TSP. Consider a path concatenated to τ in iteration i. Recall that this path is given by stitching the maximal paths computed in each subset $P_{i,j}$. To show that the length of this path is $O(\ell_{opt}(P))$, it suffices to show that: (1) the cost of the stitching is $O(\ell_{opt}(P))$, and (2) the total length of the sub-paths is $O(\ell_{opt}(P))$. Statement (1) follows since the subpaths are stitched in the order of their appearance on L. To prove (2) consider such a sub-path Q with endpoints u and v computed in $P_{i,j}$. Similar to the proof of Theorem 4.2 it can be shown that the length of Q is at most $4\sqrt{2}$ times the distance between u and v. This clearly implies that the length of Q is proportional to the length of $L_{i,j}$. Since each sub-path is computed in a different $L_{i,j}$, the total length of the sub-paths is proportional to the length of L, and hence, is $O(\ell_{opt}(P))$.

It follows that $R_\ell(\tau, P)$ is proportional to the *number* of subpaths concatenated to τ. Consider an iteration i. As above it can be shown that the number of paths required to cover at least $3/4$ of the points is $O(\max_j\{\theta_{opt}(P_{i,j})\})$. There are two ways to bound this maximum. First, $\theta_{opt}(P_{i,j}) \le \theta_{opt}(P)$. Second, since the cardinality of each $P_{i,j}$ is at most $k/2^i$, by Theorem 3.1, we get that $\max_j\{\theta_{opt}(P_{i,j})\} = O((k/2^i)/\log k)$. Summing over the m iterations, we conclude that

$$R_\ell(\tau, P) = O\left(\min\left\{\frac{k}{\log k}, \theta_{opt}(P) \cdot \log\frac{k}{\theta_{opt}(P)}\right\}\right).$$

We distinguish between three possible ranges of θ_{opt}.

1. In the range $\theta_{opt}(P) \le \frac{\sqrt{n}}{\log n}$, we set
$$k = \frac{n}{\theta_{opt}(P) \cdot \log(\frac{n}{\theta_{opt}(P)})},$$
and thus get
$$R_\theta(\tau, P) + R_\ell(\tau, P) = O\left(\frac{n}{k} + \theta_{opt}(P)\log(\frac{k}{\theta_{opt}(P)})\right)$$
$$= O\left(\theta_{opt}(P)\log(\frac{n}{\theta_{opt}(P)})\right).$$
2. In the range $\frac{\sqrt{n}}{\log n} \le \theta_{opt}(P) \le \sqrt{n \log n}$, we set $k = \sqrt{n \log n}$, and get that
$$R_\theta(\tau, P) + R_\ell(\tau, P) = O(\frac{n}{k} + \frac{k}{\log k}) = O(\sqrt{\frac{n}{\log n}})$$
3. In the range $\theta_{opt}(P) \ge \sqrt{n \log n}$, we set $m = 0$, and thus τ is set to be L. We get that
$$R_\theta(\tau, P) + R_\ell(\tau, P) = O\left(\frac{n}{\theta_{opt}(P)}\right).$$

$\square$

5. The intractability of angle-TSP and angle-CCP. In this section we prove that finding an optimal Angle-CCP is NP-hard. A similar reduction shows that the Angle-TSP problem is also NP-hard. We give the proof for the Angle-CCP problem, and then describe briefly how this proof can be modified to show the NP-hardness of the Angle-TSP problem.

THEOREM 5.1. *The* Angle-CCP *and* Angle-TSP *problems are NP-hard.*

We provide a reduction to Angle-CCP from the one-in-three 3SAT problem that is known to be NP-hard (Problem LO4 of Garey and Johnson [11]). The input to one-in-three 3SAT is a set of m clauses $C_1, \dots, C_m$, with each clause consisting of three literals from the set $\{u_1, \dots, u_n, \overline{u}_1, \dots, \overline{u}_n\}$, where $\{u_1, \dots, u_n\}$ are boolean variables and $\{\overline{u}_1, \dots, \overline{u}_n\}$ are their negations. The problem is to decide whether there exists a truth assignment to the variables such that each clause contains exactly one true literal. From now on we use the notation N to denote $n + m$.

For an instance ϕ of the one-in-three 3SAT problem, we construct a corresponding instance P of the Angle-CCP problem such that there exists a one-in-three 3SAT solution to ϕ if and only if the corresponding instance P has a cycle cover of total angle at most T, for a value T to be specified later. We index the points in P by their coordinates along the X and Y axes.

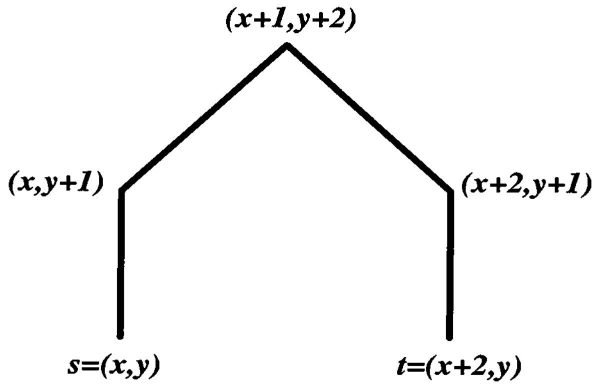


FIG. 5.1. *A U-turn gadget in direction* $+Y$.

For the reduction we need a gadget that we call a "U-turn gadget," although "V-turn gadget" would perhaps be a more appropriate name for it. This consists of points on two straight lines and on two diagonals connecting those lines as shown in Figure 5.1. Each such gadget is characterized by its *endpoints* and its *direction.* Intuitively, a gadget pointing in the direction $+Y$ with endpoints $s = (x, y)$ and $t = (x + 2, y)$ consists of a set of points in the plane such that in order to cover them efficiently we have to make a U-turn after entering at s (t) in the direction $+Y$ and exiting at t (respectively, s) in the direction $-Y$. To achieve this goal, the gadget contains $12N^5$ points where $N = n + m$. Of these, $10N^5$ points are spread uniformly on the two unit-length intervals from (x, y) to $(x, y+1)$, and from $(x+2, y)$ to $(x + 2, y + 1)$. The remaining $2N^5$ points are spread uniformly along the two diagonals from $(x, y + 1)$ to $(x + 1, y + 2)$ and from $(x + 2, y + 1)$ to $(x + 1, y + 2)$.

We are now ready to define the instance P of Angle-CCP. In P, we have nine points corresponding to each clause C_i. Suppose that clause C_i consists of the literals

u_a,$\overline{u}_b$, and u_c. The points corresponding to this clause are: (N^2i, N^2a), (N^2i, N^2b), (N^2i, N^2c+1), (N^2i+1, N^2a+1), (N^2i+1, N^2b+1), (N^2i+1, N^2c+1), (N^2i+2, N^2a+1), (N^2i+2, N^2b), and (N^2i+2, N^2c); see Figure 5.2.

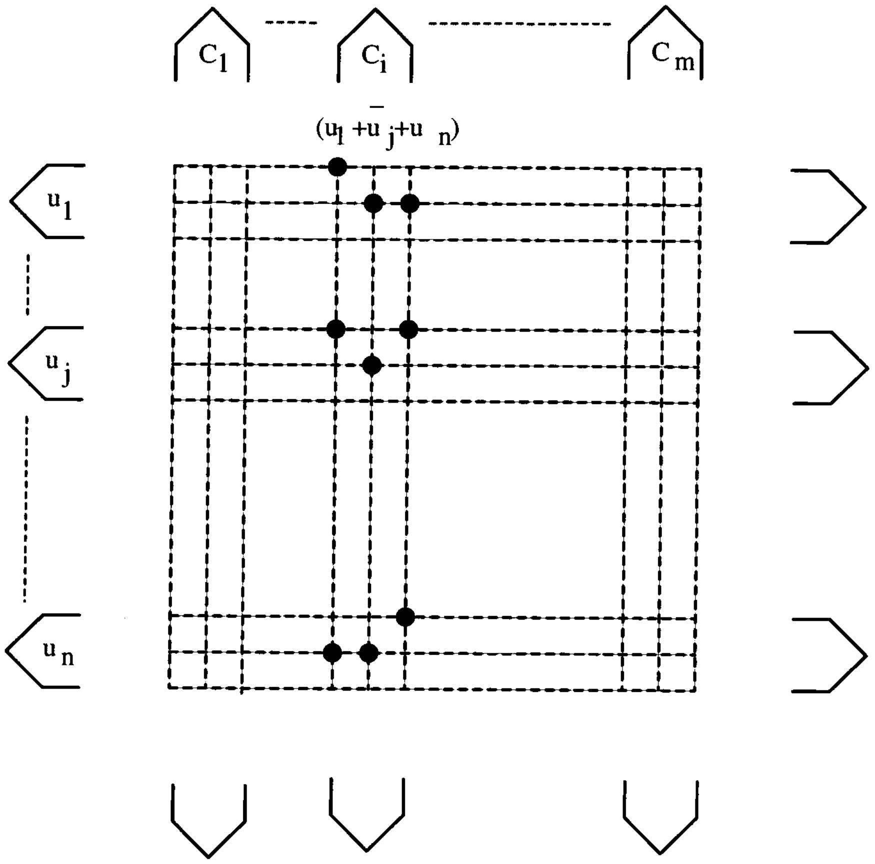


FIG. 5.2. *Illustrating the reduction.*

Let us understand the reasoning behind this assignment of points. We allocate three "columns" to each clause, where the clause C_i is associated with the columns with X-coordinates N^2i, N^2i+1, and N^2i+2. Similarly, we allocate three "rows" to each variable, where the variable u_j is associated with the rows with Y-coordinates N^2j, N^2j+1, and N^2j+2. In addition, we allocate the row N^2j to the literal u_j, and the row N^2j+1 to the literal $\overline{u}_j$. (We assume that $N = (n+m)$ is large enough, so that $N^2i + 3 < N^2(i+1)$.)

Let us fix our attention on a clause C_i and the three columns associated with it. For $j \in \{1, 2, 3\}$, in the jth column associated with this clause, we have the points in its intersection with the rows corresponding to the jth literal in the clause and the complements of the other two literals. In addition, we have the points (N^2i+3, N^2j+2), for $j = 1, \ldots, n$, and $i = 1, \ldots, m$. We call all the points defined so far *grid points.*

Finally, we add $2N$ U-turn gadgets. For each variable u_j, $1 \le j \le n$, we have two U-turn gadgets – one with endpoints $(-N^5, N^2j)$ and $(-N^5, N^2j+2)$ pointing in the direction $-X$ (the "left" gadget), and another with endpoints (N^5+N^2m, N^2j) and (N^5+N^2m, N^2j+2) pointing in the direction $+X$ (the "right" gadget). Similarly, for each clause C_i, $1 \leq i \leq m$, we have two U-turn gadgets – one with endpoints $(N^2i, -N^5)$ and $(N^2i+2, -N^5)$ pointing in the direction $-Y$ (the "bottom" gadget), and another with endpoints (N^2i, N^5+N^2n) and (N^2i+2, N^5+N^2n) pointing in the direction $+Y$ (the "top" gadget).

Our goal now is to show that the instance of the one-in-three 3SAT has a satisfying assignment if and only if the corresponding instance of Angle-CCP has a cycle cover of total angle at most $T = 2\pi N + 4N^{-4}$.

LEMMA 5.2. *If there exists a one-in-three truth assignment satisfying ϕ, then there is a cycle cover for P of total angle at most T.*

Proof. We will construct a cycle cover that has N cycles, one per clause and one per variable. Consider a clause C_i and suppose that the literal that satisfies C_i is the first literal; the other cases are similar. The cycle corresponding to C_i covers the points on the two U-turn gadgets of C_i and also the points in two of the columns assigned to C_i. These are the columns that do not contain the point intersecting the row assigned to the literal satisfying C_i, i.e., the cycle covers the columns with X-coordinates N^2i+1 and N^2i+2.

We bound the angle of the cycle starting from point $(N^2i, -N^5)$, going counterclockwise. The angle cost of covering the points of the bottom gadget exiting at $(N^2i+2, -N^5)$ is π. From here, we can cover all the points in column N^2i+2 without incurring any angle cost. Then, the angle cost of covering the top U-turn gadget entering at (N^2i+2, N^5+N^2n) and exiting at (N^2i, N^5+N^2n) is again π. From this point we traverse to point (N^2i+1, N^2n) at an angle cost of (approximately) $2N^{-5}$ and then, without incurring any further angle cost, cover all the points in column N^2i+1. Finally, we traverse from $(N^2i+1, 0)$ to $(N^2i, -N^5)$ at an angle cost of (approximately) $2N^{-5}$. Overall, the angle of the cycle is $2\pi + 4N^{-5}$.

The remaining points are covered by the cycles corresponding to the variables. Consider a variable u_j, and suppose that the assignment of u_j is TRUE; the other case is similar. The cycle corresponding to u_j covers the points on the two U-turn gadgets of u_j. It also covers the points in two of the rows assigned to u_j, viz., the row N^2j corresponding to the literal u_j and row N^2j+2.

We now bound the angle of the cycle starting from point $(-N^5, N^2j+2)$, going counterclockwise. The angle cost of covering the points of the left gadget exiting at $(-N^5, N^2j)$ is π. From here, we cover all the points in row N^2j at no additional angle cost. Then, the angle cost of covering the right U-turn gadget entering at (N^5+N^2m, N^2j) and exiting at (N^5+N^2m, N^2j+2) is again π. From this point, we cover all the points in row N^2j+2 at no additional angle cost. Overall, the angle cost is 2π. Note that in case u_j is assigned FALSE, the total angle is $2\pi + 4N^{-5}$.

Summing over all the $N = (n+m)$ cycles, we obtain that total angle is bounded by $T = 2\pi N + 4N^{-4}$. ◻

The following lemmas help us to establish the reverse direction.

LEMMA 5.3. *The angle cost of covering only the U-turn gadgets in P is $2\pi N$. The cover that achieves this has N cycles, each of angle 2π and each of which covers a pair of gadgets corresponding to either a clause or a variable. Moreover, any other cover that traverses the points in a different order has an angle cost at least $2\pi N + \Omega(N^{-3})$.*

Proof. It can be verified that the cover with N cycles in which each pair of gadgets corresponding to either a clause or a variable is covered by one cycle indeed achieves an angle cost of $2\pi N$. We now prove that this is the only cover of angle cost $2\pi N + o(N^{-3})$. Clearly, any cycle cover of this angle cost has at most N cycles. Consider such a cover $\mathcal{C}$.

Recall that each gadget contains two straight line intervals. We call these lines the *spokes* of the gadget. We have horizontal and vertical spokes. Each spoke has two endpoints: *internal* and *external*. The internal endpoint is the one closer to the grid points. For simplicity, consider first only cycle covers that cover all the points in each such spoke consecutively; i.e., when we follow a cycle in $\mathcal{C}$, the first point of a spoke that we encounter is an endpoint, and starting from it we cover all the spoke points ending at the other endpoint.

We now prove that any cycle cover $\mathcal{C}$ with the above property that covers only the spoke points must have an angle cost of at least $2\pi N$. Clearly, the cover described in the lemma achieves this angle cost. To show this we prove that the "average" angle cost of covering each spoke is at least $\pi/2$, since there are $4N$ spokes this implies the bound. Consider a cycle $C \in \mathcal{C}$. Note that by our assumption, whenever C covers a horizontal (vertical) spoke it subtends an angle 0 or π (respectively, $\pi/2$ or $3\pi/2$) with the $+X$ direction. There are two cases: either C covers only spokes of the same orientation (horizontal or vertical), or C covers both horizontal and vertical spokes. In the first case it is easy to see that the angle cost of covering k spokes of the same orientation is at least $\lceil k/2 \rceil \pi$. Thus the average angle cost is at least $\pi/2$.

Consider a cycle C that covers alternate runs of horizontal and vertical spokes. For each such run we distinguish whether if starts (ends) in an internal or external endpoint. The following properties are easy to verify.

1. The angular cost subtended when moving from an external endpoint of a vertical (horizontal) spoke to an external endpoint of a horizontal (respectively, vertical) spoke is at least $3\pi/2 - o(N^{-2})$.
2. The angular cost subtended when moving from an internal endpoint of a vertical (horizontal) spoke to an internal endpoint of a horizontal (respectively, vertical) spoke is at least $\pi/2 - o(N^{-2})$.
3. The angular cost subtended when moving from an internal endpoint of a vertical (horizontal) spoke to an external endpoint of a horizontal (respectively, vertical) spoke, or vice versa, is at least $\pi - o(N^{-2})$.
4. The angular cost subtended while covering k spokes of the same orientation by a *path* that starts and ends in external endpoints is at least $\lceil (k-2)/2 \rceil \pi$. (Note that this is only the angle cost of the path without including the angle cost of reaching the first endpoint and leaving the last.)
5. The angular cost subtended while covering k spokes of the same orientation by a *path* that starts and ends in internal endpoints is at least $\lceil k/2 \rceil \pi$.
6. The angular cost subtended while covering k spokes of the same orientation by a *path* that starts in an internal endpoint, and ends in an external endpoint, or vice versa, is at least $\lceil (k-1)/2 \rceil \pi$.

These observations imply that the average angular cost of a spoke in such a cover is more than $\pi/2$. We conclude that to achieve a cover $\mathcal{C}$ of angle cost at most $2\pi N$ all the cycles must cover only spokes of the same orientation and in this case the cost of the cover is at least $2\pi N$.

We now show that the same holds even if we remove the assumption that spokes are traversed consecutively. For each spoke ℓ consider the interval of length N^{-3} starting at its external endpoint. This interval consists of $5N^2$ points. Hence, there is a cycle $C \in \mathcal{C}$ that covers at least $5N$ of these points; we say that this C is the *main cycle* for the spoke ℓ. We claim that C must traverse at least two points of the interval one after the other. This is because each time a point not in the interval is traversed between two points in the interval, the cycle C subtends an angle cost of $\pi - o(N^{-2})$. Note that in case ℓ is horizontal (vertical), C subtends an angle 0 or π (respectively, $\pi/2$ or $3\pi/2$) with the $+X$ direction. Among the points traversed consecutively, let the external point be the one closest to the external endpoint of ℓ, and the internal point be the one closest to the internal endpoint. It is not difficult to verify that the same arguments we used above would hold if we let the main cycle of ℓ play the role of the cycle that traverses ℓ (consecutively), and the internal and external points of the interval play the role of the internal and external endpoints.

Recall that each gadget contains also two diagonal lines. To achieve the desired angle cost, it must be the case that all points on a diagonal are covered by the main cycle of the spoke touching it, and that these points are traversed either just before or just after the points of the touching spoke are traversed. To see this, recall that every cycle covers at least two spokes and that all its covered spokes are of the same orientation. Consider a such a cycle, and assume that it covers some points on a diagonal line, between the traversal of two spokes not touching it. It is easy to see that in this case the additional angular cost subtended is at least $\pi/2$. This would make the average angular cost of a spoke $\pi/2 + O(N^{-1})$; a contradiction. Now, consider the leftmost gadget on the top. The cycle that covers the points on the right diagonal of this gadget must be the main cycle of its left spoke, otherwise its angle cost would be too large, following from arguments similar to the ones given above. Hence, we have that both spokes of this gadget have the same main cycle, which covers also the points on its two diagonals. Similarly, both spokes of the bottom leftmost gadget have the same main cycle, which covers also the points on its two diagonals. Moreover, it must be the case that the same cycle covers both gadgets. Otherwise, the cost of the cycle would be $k\pi + \Omega(N^{-3})$.

In the same way we can show that both gadgets corresponding to the second clause are covered by the same cycle, and so on. It follows that in order to achieve the desired angle cost, no cycle covers gadgets corresponding to more than one clause. In a similar fashion, it can be shown that the gadgets corresponding to a variable must be covered by the same cycle, and that no cycle covers the gadgets corresponding to more than one variable. We conclude that the only cycle cover that achieves the desired cost is the one with N cycles, each of which covers a pair of gadgets corresponding to either a clause or a variable. $\square$

LEMMA 5.4. *If there exists a cycle cover P of total angle at most T, then there is a one-in-three truth assignment satisfying ϕ.*

Proof. Suppose that there exists a cover of the Angle-CCP instance of angle cost at most $T = 2\pi N + 4N^{-4}$. Since it covers all the U-turn gadgets and its cost is bounded by $2\pi N + o(N^{-3})$, it must be that this cover consists of N cycles each of which covers the two gadgets corresponding to either a clause or a variable. Consider the cycle that covers the U-turn gadgets of a clause C_i. In addition to the points of the two gadgets, this cycle can cover at most two of the columns corresponding to C_i. Otherwise, the cost of the cycle would be $2\pi + \Omega(N^{-3})$. (Note that covering more than one column in a path connecting the two gadgets would incur a cost of $\Omega(N^{-3})$.) Similarly, the cycle that covers the U-turn gadgets of a variable u_j can additionally cover at most two of the rows corresponding to u_j. One of these rows must be row $N^2 j + 2$, since none of the other cycles can cover the points on this row within the angle bounds.

It follows that for each clause C_i, all the grid points in one of its columns are covered by cycles corresponding to variables. Recall that such a column consists of points in the rows corresponding to one literal $l_j \in \{u_j, \overline{u}_j\}$ in the clause, and to the complements of the other two. Assign a TRUE value to l_j, and assign a FALSE value to the other two literals in C_i. Repeating this for all clauses implies that in the resulting truth assignment, for each clause we have exactly one literal that is TRUE. We claim that this assignment is consistent, i.e., no variable is assigned both TRUE and FALSE. This is because a literal is assigned the value TRUE only if its corresponding row is covered by the cycle corresponding to its originating variable. However, a cycle corresponding to a variable u_j can cover either the row corresponding to the literal u_j or the row corresponding to the literal $\overline{u}_j$, but not both. This implies that ϕ has a one-in-three truth assignment. □

We briefly sketch the modifications required to prove the NP-hardness of the Angle-TSP problem. Again, the reduction is quite similar and is from the one-in-three 3SAT problem. Given an instance ϕ of the one-in-three 3SAT problem, the corresponding instance of the Angle-TSP problem would include the same grid points and the left/top U-turn gadgets as before. In addition, in the "right" and the "bottom" we add U-turn gadgets of a different size that are used to "stitch" the rows (columns) corresponding to adjacent variables (respectively, clauses). Finally, we add two other gadgets: one to connect the bottom end of the rightmost column with the right end of the bottom row, and another to connect the bottom end of the leftmost column with the right end of the top row. It can be shown that there exists a tour of total angle cost at most T if and only if there is a one-in-three truth assignment satisfying ϕ.

Acknowledgments. We would like to thank Leonidas Guibas for suggesting the problem. We would like to thank the anonymous referees for helpful comments and especially for pointing out reference [9] that enabled us to improve Theorems 3.1 and 4.3.

REFERENCES

[1] P.K. Agarwal, P. Raghavan, and H. Tamaki, *Motion planning for a steering-constrained robot through moderate obstacles*, in Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing, ACM, New York, 1995, pp. 343–352.

[2] J. Barraquand and J-C. Latombe, *Nonholonomic mobile robots and optimal maneuvering*, Revue d'Intelligence Artificelle, 3 (1989), pp. 77–103.

[3] J. Barraquand and J-C. Latombe, *Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles*, Algorithmica, 10 (1993), pp. 121–155.

[4] J. Boissonnat, J. Czyzowicza, O. Devillers, J-M. Robert, and M. Yvinec, *Convex tours of bounded curvature*, in Proceedings of the Second Annual European Symposium on Algorithms (ESA94), Springer, Utrecht, The Netherlands, 1994, pp. 254–265.

[5] J. Boissonnat, A. Cérézo, and J. Leblond, *Shortest paths of bounded curvature in the plane*, J. Intelligent and Robotic Systems: Theory and Appl., 11 (1994), pp. 5–20.

[6] J.E. Boyce, D.P. Dobkin, R.L. Drysdale III, and L.J. Guibas, *Finding extremal polygons*, SIAM J. Comput., 14 (1985), pp. 134–147.

[7] N. Christofides, *Worst-Case Analysis for a New Heuristic for the Traveling Salesman Problem*, Report 388, Graduate School of Industrial Administration, Carnegie–Mellon University, Pittsburgh, PA, 1976.

[8] J.C. Clements, *Minimum-time turn trajectories to fly-to points*, Optimal Control Appl. Methods, 11 (1990), pp. 39–50.

[9] P. Erdős and G. Szekeres, *A combinatorial problem in geometry*, Composito Math., 2 (1935), pp. 463–470.

[10] T. Fraichard, *Smooth trajectory planning for a car in a structured world*, in Proceedings of the IEEE International Conference on Robotics, IEEE Press, Piscataway, NJ, 1991, pp. 318–323.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, CA, 1979.

[12] L. Guibas, *private communication*, Stanford Univeristy, Palo Alto, CA, 1995.

[13] P. Jacobs and J. Canny, *Planning smooth paths for mobile robots*, in Proceedings of the IEEE International Conference on Robotics, IEEE Press, Piscataway, NJ, 1989, pp. 2–7.

[14] J-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA, 1991.

[15] J-P. Laumond, *Finding collision-free smooth trajectories for a non-holonomic mobile robot*, in Proceedings of the IEEE International Conference on Robotics, IEEE Press, Piscataway, NJ, 1987, pp. 1120–1123.

[16] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley, New York, 1985.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 712–727

ON LEARNING FUNCTIONS FROM NOISE-FREE AND NOISY SAMPLES VIA OCCAM'S RAZOR[*]

B. NATARAJAN[†]

Abstract. An Occam approximation is an algorithm that takes as input a set of samples of a function and a tolerance ϵ and produces as output a compact representation of a function that is within ϵ of the given samples. We show that the existence of an Occam approximation is sufficient to guarantee the probably approximate learnability of classes of functions on the reals even in the presence of arbitrarily large but random additive noise. One consequence of our results is a general technique for the design and analysis of nonlinear filters in digital signal processing.

Key words. probably approximate learning, noisy data

AMS subject classifications. 68T05

PII. S0097539794277111

1. Introduction.

We begin with an overview of our main result. Suppose we are allowed to randomly sample a function f on the reals, with the sample values of f being corrupted by additive random noise ν of strength b. Let g be a sparse but approximate interpolant of the random samples of f—sparse in the sense that it can be compactly represented, and approximate in the sense that the interpolation error in g with respect to the samples is at most b. Intuitively, we would expect that the noise and the interpolation error add in their contribution to the error in g with respect to f, i.e., $||g - f|| \approx 2b$. However, our results show that the noise and the interpolation error tend to cancel, i.e., $||g-f|| \approx 0$, with the extent of the cancellation depending on the sparseness of g and how often f is sampled.

We consider the paradigm of probably approximately correct learning of Valiant, as reviewed in Natarajan [21] and Anthony and Biggs [3]. Broadly speaking, the paradigm requires an algorithm to identify an approximation to an unknown target set or function, given random samples of the set or function. Of central interest in the paradigm is the relationship between the complexity of the algorithm, the a priori information available about the target set or function, and the goodness of the approximation. When learning sets, if the target set is known to belong to a specific target class, Blumer et al. [9] establish that the above quantities are related via the Vapnik–Chervonenkis dimension of the target class, and if this dimension is finite, then learning is possible. They also show that even if the Vapnik–Chervonenkis dimension is infinite, learning may still be possible in a generalized sense, via the use of Occam's Razor. Specifically, if it is possible to compress collections of examples for the target set, then a learning algorithm exists. For instance, the class of sets composed of finitely many intervals on the reals satisfies this condition and hence can be learned. Kearns and Schapire [18] extend the above results to the case of probabilistic concepts and offer an Occam's Razor result in that setting. For the case of functions, Natarajan [20] examines conditions under which learning is possible and shows that it is sufficient if the "graph" dimension of the target class is finite. Haussler [13] generalizes this result to a "robust" model of

[*]Received by the editors November 10, 1994; accepted for publication (in revised form) September 2, 1997; published electronically December 7, 1999.
http://www.siam.org/journals/sicomp/29-3/27711.html

[†]Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304 (natarajan@hpl.hp.com).

learning and shows that it is sufficient if the metric dimension of the class is finite. Alon et al. [1] generalize these results further via the "scale-sensitive" dimension. Anthony et al. [4] give sufficient conditions linking approximate interpolation and learning. Learning sets in the presence of noise was studied by Angluin and Laird [2], Kearns and Li [17], and Sloan [27], amongst others. Kearns and Li show that the principle of Occam's Razor can be used to learn in the presence of a limited amount of noise. Bartlett, Long, and Williamson [6] link the learnability of functions in the presence of observation noise to the "fat-shattering" function of the class.

We consider the learning of functions, both with and without random sampling noise. In the latter case, we mean that the function value obtained in the sampling process is corrupted by additive random noise. In this setting, we show that if it is possible to construct sparse but approximate interpolants to collections of examples, then a learning algorithm exists. For instance, it is possible to construct optimally sparse but approximate piecewise linear interpolants to collections of examples of univariate functions. As a consequence, we find that the class of all univariate Baire functions [10] can be learned in terms of the piecewise linear functions, with the number of examples required to learn a particular Baire function depending on the number of pieces required to approximate it as a piecewise linear function. In the absence of noise, our results hold for all L_p metrics over the space of functions. In the presence of noise, the results are for the L_2 and L_∞ metrics. There are two points of significance in regard to these results: (1) the noise need not be of zero mean for the L_∞ metric; (2) for both metrics, the magnitude of the noise can be made arbitrarily large and learning is still possible, although at increased cost.

Our results are closely related to the work in signal processing [24], [25], [16], and [19] on the reconstruction and filtering of discretely sampled functions. However, much of that literature is on linear systems and filters, with the results expressed in terms of the Fourier decomposition, while relatively little is known about nonlinear systems or filters. Our results allow a unified approach to both linear and nonlinear systems and filters, and in that sense, we offer a new and general technique for signal reconstruction and filtering. This is explored in [23], where we analyze a broad class of filters that separate functions with respect to their encoding complexity—since random noise has high complexity in any deterministic encoding and is hard to compress, it can be separated from a function of low complexity.

Of related interest is the literature on sparse polynomial interpolation [8], [7], [12], and [5]. These papers study the identification of an unknown polynomial that can be evaluated at selected points.

The results of this paper appeared in preliminary form in [22].

2. Preliminaries.

We consider functions $f : [0, 1] \to [-K, K]$, where $[0, 1]$ and $[-K, K]$ are intervals on the reals $\mathbf{R}$. A class of functions F is a set of such functions and is said to be of envelope K. Unless we state otherwise, we assume that a class has unit envelope. A complexity measure l is a mapping from F to the natural numbers $\mathbf{N}$. An example for a function f is a pair $(x, f(x))$. Our discussion will involve metrics on several spaces. For the sake of concreteness, we will deal only with the L_p metrics, $p \in \mathbf{N}, p \geq 1$, and will define these exhaustively. For two functions f and g and probability distribution P on $[0, 1]$,

$$L_p(f, g, P) = \left(\int_x |f(x) - g(x)|^p dP \right)^{1/p}. \tag{2.1}$$

For function f and collection of examples $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$,

$$L_p(f, S) = \left(\frac{1}{m}\sum |f(x_i) - y_i|^p\right)^{1/p}. \tag{2.2}$$

Let l be a complexity measure on a class G, f a function not necessarily in G, L a metric, and P a distribution on $[0, 1]$. For $\epsilon > 0$, $l_{\min}(f, \epsilon, P, L)$ is defined by

$$l_{\min}(f, \epsilon, P, L) = \min\ \{l(g) | g \in G, L(f, g, P) \leq \epsilon\}. \tag{2.3}$$

If $\{l(g)|g \in G, L(f, g, P) \leq \epsilon\}$ is empty, then $l_{\min}(f, \epsilon, P, L) = \infty$.
Analogously, for a collection of examples S and metric L,

$$l_{\min}(S, \epsilon, L) = \min\ \{l(g)|g \in G, L(g, S) \leq \epsilon\}. \tag{2.4}$$

If $\{l(g)|g \in G, L(g, S) \leq \epsilon\}$ is empty, then $l_{\min}(S, \epsilon, L) = \infty$.
Using Jensen's inequality [10], it is easy to show that

$$L_1(f, g, P) \leq L_p(f, g, P) \leq (L_1(f, g, P))^{1/p}. \tag{2.5}$$

A learning algorithm A for target class F has at its disposal a subroutine EXAMPLE, that at each call returns an example for an unknown target function $f \in F$. The example is chosen at random according to an arbitrary and unknown probability distribution P on $[0, 1]$. After seeing some number of such examples, the learning algorithm identifies a function g in the hypothesis class G such that g is a good approximation to f. Formally, we have the following. Algorithm A is a learning algorithm for F in terms of G, with respect to metric L, if (a) A takes as input ϵ and δ; (b) A may call EXAMPLE; (c) for all probability distributions P and all functions f in F, A identifies a function $g \in G$, such that with probability at least $(1 - \delta)$, $L(f, g, P) \leq \epsilon$.

The sample complexity of a learning algorithm for F in terms of G, with respect to the complexity measure l, is the number of examples sought by the algorithm as a function of the parameters of interest. In noise-free learning, these parameters are ϵ, δ, and $l_{\min}(f, \epsilon, P, L)$, where f is the target function. In the presence of random sampling noise, the properties of the noise distribution would be additional parameters. Since we permit our learning algorithms to be probabilistic, we will consider the expected sample complexity of a learning algorithm, which is the expectation of the number of examples sought by the algorithm as a function of the parameters of interest.

In light of the relationship between the various L_p metrics established in (2.5), we focus on the L_1 metric and, unless we explicitly state otherwise, we use the term *learning algorithm* to signify a learning algorithm with respect to the L_1 metric. We denote the probability of an event A occurring by $Pr\{A\}$ and the expected value of a random variable x by $\mathbf{E}\{x\}$.

With respect to a class of functions F of envelope K, $\epsilon > 0$ and $m \in \mathbf{N}$, and metric L: The size of the minimum ϵ-cover of a set $X = \{x_1, x_2, \ldots, x_m\}$ is the cardinality of the smallest set of functions G from X to $\mathbf{R}$ such that for each $f \in F$, there exists $g \in G$ satisfying $L(f, g, P_X) \leq \epsilon$, where P_X is the distribution placing equal mass $1/m$ on each of the x_i in X. The covering number $N(F, \epsilon, m, L)$ is the maximum of the size of the minimum ϵ-cover over all $\{x_1, x_2, \ldots, x_m\}$.

The following convergence theorem will be of considerable use to us. The theorem is predicated on mild measurability assumptions as discussed in [26].

THEOREM 2.1 (see [26, pp. 25–27]). *For a class of functions F with envelope K, the probability that the observed mean over $m \geq 8/\epsilon^2$ random samples of any function in F will deviate from its true mean by more than ϵ is at most*

$$8N(F, \epsilon/8, m, L_1)e^{-(m/128)(\epsilon/K)^2}. \tag{2.6}$$

3. Occam approximations. An approximation algorithm Q for hypothesis class G and metric L is an algorithm that takes as input a collection S of examples and a tolerance $\epsilon > 0$ and identifies in its output a function $g \in G$ such that $L(g, S) \leq \epsilon$, if such exists.

For particular values of $m, t \in \mathbf{N}$ and $\epsilon > 0$, let $\hat{G}$ be the class of all functions identified in the output of Q when its input ranges over all collections S of m examples such that $l_{\min}(S, \epsilon, L) \leq t$. Q is said to be an Occam approximation if there exists a polynomial $q(a, b) : \mathbf{N} \times \mathbf{R} \to \mathbf{N}$ and $r : \mathbf{N} \to \mathbf{N}$ such that $r(m)$ is $O(m^\alpha)$, for fixed $0 \leq \alpha < 1$, and for all m, t, ϵ, and $\zeta > 0$, $\log(N(\hat{G}, \zeta, m, L)) \leq q(t, 1/\zeta)r(m)$. We say that (q, r) is the characteristic of Q.

The above notion of an Occam approximation is a generalization to functions of the more established notion of an Occam algorithm for concepts [9] and its extension to probabilistic concepts in [18].

Example 1. We consider univariate polynomials with coefficients of Euclidean norm at most unity, i.e., polynomials of the form $\sum a_i x^i$, where $\sum a_i^2 \leq 1$. Let G be the class of such polynomials over the unit interval. As a complexity measure on G, we choose the degree of the polynomial. Consider the following approximation algorithm Q with respect to the L_2 metric. Given is a collection of samples

$$S = \{(x_1, y_1),\ (x_2, y_2), \ldots,\ (x_m, y_m)\}, \tag{3.1}$$

and tolerance ϵ. We shall fit the data with a polynomial of the form $a_0 + a_1 x + \cdots + a_i x^i + \cdots + a_d x^d$. For $d = 0, 1, 2, \ldots$ construct a linear system $Xa = y$, where X is the matrix of m rows and d columns, with row vectors of the form $(1, x_i, x_i^2 \ldots, x_i^d)$, y is the column vector $(y_1, y_2, \ldots, y_m)$, and a is the column vector $(a_0, a_1, \ldots, a_d)$. Find the smallest value of d for which the minimum Euclidean norm least squares solution a of the linear system has residue at most ϵ in the L_2 norm, and the Euclidean norm of a is at most unity. If such a solution vector a exists, it determines the polynomial g. This computation can be carried out efficiently [11].

Claim 1. Algorithm Q is Occam.

Proof. Fix the number of samples m, the degree d of the polynomial, and the tolerance ϵ. For collection of samples S, $l_{\min}(S, \epsilon, L_2)$ is the degree of the lowest-degree polynomial that fits the samples in S with error at most ϵ in the L_2 metric. By construction, Q outputs only polynomials of degree d on such inputs. Thus, $\hat{G}$ is a subset of the class of polynomials of degree d. By Claim 2,

$$\log(N(\hat{G}, \zeta, m, L_2) \leq (d+1)\log(2m/\zeta), \tag{3.2}$$

which implies that Q is Occam.

Claim 2. If F is the class of polynomials of degree d with coefficients of Euclidean norm at most unity,

$$N(F, \zeta, m, L_2) \leq (2(d+1)/\zeta)^{d+1}. \tag{3.3}$$

Proof. Let $f(x)$ be a polynomial in F. Since f is of Euclidean norm at most unity, each of the $d+1$ coefficients of f lie in $[-1, +1]$. Truncate each of the coefficients to the nearest multiple of $\zeta/(d+1)$ to obtain a polynomial g. The resulting function g is clearly everywhere within ζ of f on $[0,1]$. It follows that for every polynomial $f \in F$ there exists a polynomial $g \in F$ within ζ of f in the L_2 metric, where each coefficient of g is one of $(2(d+1)/\zeta)$ values. Thus, g is one of at most $(2(d+1)/\zeta)^{d+1}$ functions. Hence the claim.

Example 2. Another class with an Occam approximation algorithm with respect to the L_2 metric is the class of trigonometric interpolants with coefficients having bounded Euclidean norm, and complexity measure the highest frequency, i.e., functions of the form

$$f(x) = \sum_{i=0}^{\infty} A_i \cos(2\pi i x) + \sum_{i=0}^{\infty} B_i \sin(2\pi i x), \tag{3.4}$$

where $\sum a_i^2 \leq 1$. For the class of such functions, the least squares technique described in the context of the polynomials in Example 1 can be used to construct an Occam approximation.

4. Noise-free learning. In this section we show that the existence of an Occam approximation is sufficient to guarantee the efficient learnability of a class of functions in the absence of noise. The theorem is stated for Occam approximations with respect to the L_1 metric and subsequently is extended to other metrics.

THEOREM 4.1. *Let F be the target class and G the hypothesis class with complexity measure l, both of envelope* 1. *Let Q be an Occam approximation for G, with respect to metric L_1, with characteristic (q,r). Then, there exists a learning algorithm for F with expected sample complexity polynomial in $1/\epsilon$, $1/\delta$, and $l_{\min}(f, \epsilon/8, P, L_1)$.*

Proof. We claim that Algorithm A_1 below is a learning algorithm for F. Let f be the target function. In words, the algorithm makes increasingly larger guesses for $l_{\min}(f)$, and based on its guesses it constructs approximations for f, halting if and when a constructed approximation appears to be good.

```
ALGORITHM $A_1$.
input $\epsilon, \delta$
begin
    $t = 2$;
    repeat forever
        let $m$ be the least integer satisfying
        $m \geq (9216/\epsilon^2)q(t, 32/\epsilon)r(m)$;
        make $m$ calls of EXAMPLE to get collection $S$;
        let $g = Q(S, \epsilon/4)$;
        let $m_1 = (16t^2)/(\epsilon^2\delta)$;
        make $m_1$ calls of EXAMPLE to get collection $S_1$;
        if $L_1(g, S_1) \leq (3/4)\epsilon$ then output $g$ and halt;
        else $t = t + 1$;
    end
end
```

We first show that the probability of the algorithm halting with a function g as output such that $L_1(f, g, P) > \epsilon$ is at most δ. At each iteration, by Chebyshev's inequality,

$$Pr\left\{|L_1(g, S_1) - L_1(g, f, P)| \geq \epsilon/4\right\} \leq \frac{16}{\epsilon^2 m_1}. \tag{4.1}$$

Since $m_1 = 16t^2/(\epsilon^2\delta)$,

$$Pr\left\{|L_1(g, S_1) - L_1(g, f, P)| \geq \epsilon/4\right\} \leq \frac{\delta}{t^2}. \tag{4.2}$$

If the algorithm halts, then $L_1(g, S_1) \leq (3/4)\epsilon$ and hence

$$Pr\left\{L_1(g, f, P) > \epsilon\right\} \leq \frac{\delta}{t^2}. \tag{4.3}$$

Hence, the combined probability that the algorithm halts at any iteration with $L_1(g, f, P) > \epsilon$ is at most $\sum_{t=2}^{\infty} \frac{\delta}{t^2} < \delta$.

Let $t_0 = l_{\min}(f, \epsilon/8, P, L_1)$. We now show that when $t \geq t_0$, the algorithm halts on iteration t with probability at least 1/4. Let $h \in G$ be such that $L_1(h, f, P) \leq \epsilon/8$ and $l(h) = t_0$. Fix $t \geq t_0$. By Chebyshev's inequality,

$$Pr\left\{|L_1(h, S) - L_1(h, f, P)| \geq \epsilon/8\right\} \leq \frac{64}{\epsilon^2 m}. \tag{4.4}$$

Hence, if $m \geq 256/\epsilon^2$ with probability at least $(1 - 1/4) = 3/4$,

$$|L_1(h, S) - L_1(h, f, P)| \leq \epsilon/8. \tag{4.5}$$

Since $L_1(h, f, P) \leq \epsilon/8$, it follows that with probability 3/4, $L_1(h, S) \leq \epsilon/4$. In other words, with probability 3/4, $l_{\min}(S, \epsilon/4, L_1) \leq l(h) = t_0 \leq t$.

At each iteration of A_1, let $\hat{G}$ be the class of all functions that Q could output if $l_{\min}(S, \epsilon/4, L_1) \leq t$ holds. Since Q is Occam, $\log(N(\hat{G}, \epsilon/32, m)) \leq q(t, 32/\epsilon)r(m)$. Let H be the class of functions $H = \{h : h(x) = |f(x) - g(x)|,\ g \in \hat{G}\}$, and let h_1 and h_2 be any two functions in H. Now,

$$|h_1(x) - h_2(x)| = ||f(x) - g_1(x)| - |f(x) - g_2(x)|| \tag{4.6}$$
$$\leq |g_1(x) - g_2(x)|. \tag{4.7}$$

Hence,

$$N(H, \epsilon/32, m, L_1) \leq N(\hat{G}, \epsilon/32, m, L_1) \leq q(t, 32/\epsilon)r(m). \tag{4.8}$$

If

$$m \geq (9216/\epsilon^2)q(t, 32/\epsilon)r(m), \tag{4.9}$$

m satisfies

$$8e^{q(t,32/\epsilon)r(m)}e^{-(m/128)(\epsilon/4)^2} \leq 1/4, \tag{4.10}$$

and then by Theorem 2.1, with probability at least $(1-1/4) = 3/4$, the observed mean of each function $h \in H$ will be within $\epsilon/4$ of its true mean. That is, with probability at least 3/4, for each $g \in \hat{G}$,

$$|L_1(g, S) - L_1(f, g, P)| \leq \epsilon/4. \tag{4.11}$$

If g is the function that is returned by $Q(S, \epsilon/4)$, then $L_1(g, S) \leq \epsilon/4$. Therefore, if $l_{\min}(S, \epsilon/4, L_1) \leq t$, with probability at least 3/4, $L_1(f, g, P) \leq \epsilon/2$. Since we showed earlier that with probability at least 3/4, $l_{\min}(S, \epsilon/4, L) \leq t$, we get that with probability at least 1/2 the function g returned by Q will be such that $L_1(f, g, P) \leq \epsilon/2$.

We now estimate the probability that the algorithm halts when Q returns a function g such that $L_1(f, g, P) \leq \epsilon/2$. Once again by Chebyshev's inequality,

$$Pr\{|L_1(g, S_1) - L_1(g, f, P)| \geq \epsilon/4\} \leq \frac{16}{\epsilon^2 m_1}. \tag{4.12}$$

Given that $L_1(f, g, P) \leq \epsilon/2$ and that $m_1 = 16t^2/(\epsilon^2\delta)$, we have

$$Pr\{L_1(g, S_1) \geq (3/4)\epsilon\} \leq \frac{\delta}{t^2} \leq 1/2. \tag{4.13}$$

Hence, we have $Pr\{L_1(g, S_1) < (3/4)\epsilon\} \geq 1/2$. That is, if the function g returned by Q is such that $L_1(f, g, P) \leq \epsilon/2$, then A_1 will halt with probability at least 1/2.

We have therefore shown that when $t \geq t_0$, the algorithm halts with probability at least 1/4. Hence, the probability that the algorithm does not halt within some $t > t_0$ iterations is at most $(3/4)^{t-t_0}$. Noting that the function $q()$ is a polynomial in its arguments, it follows that the expected sample complexity of the algorithm is polynomial in $1/\epsilon$, $1/\delta$ and t_0.

We now discuss the extension of Theorem 4.1 to other metrics. The one ingredient in the proof of Theorem 4.1 that does not directly generalize to the other L_p metrics is the reliance on Theorem 2.1, which is in terms of the L_1 metric. This obstacle can be overcome by using the relationship between the various L_p metrics as given by inequality 2.5.

5. Learning in the presence of noise. We assume that the examples for the target function are corrupted by additive random noise in that EXAMPLE returns $(x, f(x) + \nu)$, where ν is a random variable. The noise variable ν is distributed in an arbitrary and unknown fashion and is independent of the target function and the sampling distribution.

We can show that the existence of an Occam approximation suffices to guarantee learnability in the presence of such noise, under some special conditions. Specifically we assume that the strength of the noise in the metric of interest is known a priori to the learning algorithm. Then a learning algorithm would work as follows. Obtain a sufficiently large number of examples, large enough that the observed strength of the noise is close to the true strength. Pass the observed samples through an Occam approximation, with tolerance equal to the noise strength. The function output by the Occam approximation is a candidate for a good approximation to the target function. Test this function against additional samples to check whether it is indeed close. Since these additional samples are also noisy, take enough samples to ensure that the observed noise strength in these samples is close to the true strength. The candidate function is good if it checks to be roughly the noise strength away from these samples.

The essence of the above procedure is that the error allowed of the Occam approximation is equal to the noise strength, and the noise and the error subtract rather than add. In order for us to prove that this indeed is the case, we must restrict ourselves to linear metrics; metrics L such that in the limit as the sample size goes to infinity, the observed distance between a function g in the hypothesis class and the noisy target function is the sum of the strength of the noise and the distance between g and the noise-free target function. Two such metrics are the L_∞ metric and the square of the L_2 metric when the noise is of zero mean.

5.1. Noise of known L_2 measure. We assume that the noise is of bounded magnitude, zero mean, and known L_2 measure. Specifically, (1) we are given $b \geq 0$ such the noise ν is a random variable in $[-b, +b]$; (2) the noise ν has zero mean; (3) we are given the variance c of the noise. It is easy to see that statistical access to the noise variable ν is sufficient to obtain accurate estimates of the variance c.

THEOREM 5.1. *Let F be the target class and G the hypothesis class with complexity measure l, where F and G are of envelope 1 and $b+1$, respectively. Assume a noise source of known L_2 measure per our assumptions. Let Q be an Occam approximation for G with respect to the L_2 metric, with characteristic (q, r). Then, there exists a learning algorithm for F with expected sample complexity polynomial in $1/\epsilon$, $1/\delta$, $l_{\min}(f, \epsilon/8, P, L_2^2)$, and the noise bound b.*

Proof. Let f be the target function. Algorithm A_2 below is a learning algorithm for F in the L_2^2 metric; i.e., on input ϵ and δ, with probability at least $1-\delta$, the algorithm will identify a function g such that $L_2^2(f, g, P) \leq \epsilon$.

ALGORITHM A_2.

```
input $\epsilon, \delta$, noise bound $b$, noise variance $c$;
begin
    $t = 2$;
    repeat forever
        let $m$ be the least integer satisfying
        $m \geq \frac{32768 \log(8)(b+1)^4}{\epsilon^2} q\,(t, (128(b+2)/\epsilon))\, r(m) + \frac{4096(b+1)^4}{\epsilon^2}$;
        make $m$ calls of EXAMPLE to get collection $S$;
        let $g = Q(S, \sqrt{c+\epsilon/4})$;
        let $m_1 = \frac{256(b+1)^4 t^2}{\delta\epsilon^2}$;
        make $m_1$ calls of EXAMPLE to get collection $S_1$;
        if $L_2^2(g, S_1) \leq c + (3/4)\epsilon$ then output $g$ and halt;
        else $t = t + 1$;
    end
end
```

We first show that the probability of the algorithm halting with a function g as output such that $L_2^2(f, g, P) > \epsilon$ is less than δ. Now,

$$L_2^2(g, S_1) = \frac{1}{m_1} \sum_{S_1} (g - (f + \nu))^2 \tag{5.1}$$

$$= \frac{1}{m_1} \sum_{S_1} (g-f)^2 + \frac{2}{m_1} \sum_{S_1} \nu(g-f) + \frac{1}{m_1} \sum_{S_1} \nu^2. \tag{5.2}$$

Since $\mathbf{E}\{\nu(g-f)\} = 0$, $\mathbf{E}\{\nu^2\} = c$, and $\mathbf{E}\{(g-f)^2\} = L_2^2(g, f, P)$, it follows that $\mathbf{E}\{L_2^2(g, S_1)\} = L_2^2(g, f, P) + c$. Noting that $(g-(f+\nu))^2 \leq (2(b+1))^2$, we can invoke Chebyshev's inequality to write

$$Pr\left\{\left|L_2^2(g, S_1) - (L_2^2(g, f, P) - c)\right| \geq \epsilon/4\right\} \leq \frac{256(b+1)^4}{m_1 \epsilon^2}. \tag{5.3}$$

If $L_2^2(g, f, P) > \epsilon$, (5.3) implies that

$$Pr\{L_2^2(g, S_1) \leq c + (3/4)\epsilon\} \leq \frac{256(b+1)^4}{m_1 \epsilon^2}. \tag{5.4}$$

If

$$m_1 = \frac{256(b+1)^4 t^2}{\delta \epsilon^2}, \tag{5.5}$$

then

$$Pr\{L_2^2(g, S_1) \leq c + (3/4)\epsilon\} \leq \frac{\delta}{t^2}. \tag{5.6}$$

Summing over all $t \geq 2$, we get that the probability that the algorithm halts with $L_2^2(g, f, P) > \epsilon$ is at most δ.

Let $h \in G$ be such that $L_2^2(h, f, P) \leq \epsilon/8$ and $l_{\min}(f, \epsilon/8, P, L_2^2) = l(h) = t_0$. As in (5.3) and (5.6), with Chebyshev's inequality we can show that

$$Pr\{L_2^2(h, S) > c + \epsilon/4\} \leq \frac{1024(b+1)^4}{m\epsilon^2}. \tag{5.7}$$

If $m \geq 4096(b+1)^4/\epsilon^2$, then $Pr\{L_2^2(h, S) > c + \epsilon/4\} \leq 1/4$ and as a consequence

$$Pr\{l_{\min}(S, c + \epsilon/4, L_2^2) \leq l(h) \leq t_0\} \geq 3/4. \tag{5.8}$$

Let $\hat{G}$ be the class of functions that Q could output on inputs S and $\sqrt{c + \epsilon/4}$, when S satisfies $l_{\min}(S, c+\epsilon/4, L_2^2) \leq t_0$. Let H be the class of functions $\{(f-g)^2 | g \in \hat{G}\}$. Then, for every pair h_1 and h_2 in H, there exists g_1 and g_2 in $\hat{G}$ such that

$$|h_1 - h_2| = |(f - g_1)^2 - (f - g_2)^2| \tag{5.9}$$
$$= |f^2 + g_1^2 - 2fg_1 - f^2 - g_2^2 + 2fg_2| \tag{5.10}$$
$$= |g_1^2 - g_2^2 - 2f(g_1 - g_2)| \tag{5.11}$$
$$= |(g_1 - g_2)(g_1 + g_2 - 2f)|. \tag{5.12}$$

Since g_1 and g_2 have envelope $b + 1$ and f has envelope 1, we can write

$$|(g_1 - g_2)(g_1 + g_2 - 2f)| \leq 2(b+2)|g_1 - g_2|. \tag{5.13}$$

Hence,

$$|h_1 - h_2| \leq 2(b+2)|g_1 - g_2|, \tag{5.14}$$

and $L_1(h_1, h_2, P) \leq 2(b+2)L_1(g_1, g_2, P)$. Invoking (2.5), we get

$$L_1(h_1, h_2, P) \leq 2(b+2)L_1(g_1, g_2, P) \leq 2(b+2)L_2(g_1, g_2, P). \tag{5.15}$$

Combining the above with the assumption that Q is Occam, it follows that

$$N(H, \epsilon/64, m, L_1) \leq N\left(\hat{G}, \left(\frac{\epsilon}{128(b+2)}\right), m, L_2\right) \tag{5.16}$$
$$\leq e^q\left(t_0, (128(b+2)/\epsilon)\right) r(m). \tag{5.17}$$

Noting that H has envelope $(b+1)^4$ and invoking Theorem 2.1, we have that if $t \geq t_0$, $l_{\min}(S, c + \epsilon/4, L_2) \leq t_0$, and

$$m \geq \frac{32768 \log(8)(b+1)^4}{\epsilon^2} q\left(t, (128(b+2)/\epsilon)\right) r(m), \tag{5.18}$$

then with probability at least 3/4 the observed mean over the m samples of S of each $h \in H$ will be within $\epsilon/8$ of its true mean. That is,

$$Pr\left\{\left|\frac{1}{m}\sum_S (g-f)^2 - L_2^2(g, f, P)\right| \leq \epsilon/8\right\} \geq 3/4. \tag{5.19}$$

Noting that the probability estimate in the above inequality is conditional on (5.8), we can remove the conditionality by combining it with (5.8) to get

$$Pr\left\{\left|\frac{1}{m}\sum_S (g-f)^2 - L_2^2(g,f,P)\right| \le \epsilon/8\right\} \ge 1/2, \tag{5.20}$$

when

$$m \ge \frac{32768\log(8)(b+1)^4}{\epsilon^2} q\left(t,(128(b+2)/\epsilon)\right) r(m) + \frac{4096(b+1)^4}{\epsilon^2}. \tag{5.21}$$

Now,

$$L_2^2(g,S) = (1/m)\sum_S (g-f)^2 + (2/m)\sum_S \nu(g-f) + (1/m)\sum_S \nu^2. \tag{5.22}$$

Once again by Chebyshev's inequality, we can write

$$Pr\left\{\left|(2/m)\sum_S \nu(g-f) + (1/m)\sum_S \nu^2 - c\right| \ge \epsilon/8\right\} \le \frac{256(b+2)^4}{\epsilon^2 m}. \tag{5.23}$$

If

$$m \ge \frac{512(b+2)^4}{\epsilon^2}, \tag{5.24}$$

then

$$Pr\left\{\left|(2/m)\sum_S \nu(g-f) + (1/m)\sum_S \nu^2 - c\right| \ge \epsilon/8\right\} \le 1/2. \tag{5.25}$$

Combining (5.20), (5.22), and (5.25), we have

$$Pr\left\{\left|L_2^2(g,S) - L_2^2(g,f,P) - c\right| \le \epsilon/4\right\} \ge 1/4, \tag{5.26}$$

when

$$m \ge \frac{32768\log(8)(b+1)^4}{\epsilon^2} q\left(t,(128(b+2)/\epsilon)\right) r(m) + \frac{4096(b+1)^4}{\epsilon^2}. \tag{5.27}$$

If g is the function output by $Q(S, \sqrt{c+\epsilon/4})$, then $L_2^2(g,S) \le c+\epsilon/4$ and (5.26) can be written as

$$Pr\{L_2^2(g,f,P) \le \epsilon/2\} \ge 1/4. \tag{5.28}$$

If $L_2^2(g,f,P) \le \epsilon/2$, then by (5.3) we get

$$Pr\{L_2^2(g,S_1) \le c + (3/4)\epsilon\} \ge 1 - \frac{256(b+1)^4}{m_1\epsilon^2}. \tag{5.29}$$

If m_1 is chosen as in the algorithm, then we can rewrite the above as

$$Pr\{L_2^2(g,S_1) \le c + (3/4)\epsilon\} \ge 1 - \delta/t^2 \ge 3/4. \tag{5.30}$$

Combining (5.28) and (5.30), we get that when $t \geq t_0$, with probability at least 3/16, the function g output by $Q(S, \sqrt{c+\epsilon/4})$ will be such that $L_2^2(g, S_1) \leq c+(3/4)\epsilon$ and the algorithm will halt. Hence, the probability that the algorithm does not halt within some $t > t_0$ iterations is at most $(13/16)^{t-t_0}$. Noting that the function $q()$ is a polynomial in its arguments, it follows that the expected sample complexity of the algorithm is polynomial in $1/\epsilon$, $1/\delta$, t_0, and b.

Our assumption that the noise lies in a bounded range $[-b, +b]$ excludes natural distributions over infinite spans such as the normal distribution. However, we can include such distributions by a slight modification to our source of examples. Specifically, assume that we are given a value of b such that the second moment of the noise variable outside the range $[-b, +b]$ is small compared to ϵ. Noting that the target function f has range $[-1, 1]$ by assumption, we can screen the output of EXAMPLE, rejecting the examples with values outside $[-(b+1), +(b+1)]$ and otherwise passing them on to the learning algorithm. Thus, the noise distribution effectively seen by the learning algorithm is of bounded range $[-b, +b]$. We leave it to the reader to calculate the necessary adjustments to the sampling rates of the algorithm.

5.2. Noise of known L_∞ measure. We assume that we know the L_∞ measure of the noise in that we are given the following:

(1) $b \geq 0$ such that the noise ν is a random variable in $[-b, +b]$, not necessarily of zero mean. Also, the symmetry is not essential; it suffices if $b_1 \leq b_2$ are given, with $\nu \in [b_1, b_2]$.

(2) A function $\gamma(\epsilon)$ polynomial in ϵ such that $Pr\{\nu \in [b-\epsilon, b]\} \geq \gamma(\epsilon)$, and $Pr\{\nu \in [-b, -b+\epsilon]\} \geq \gamma(\epsilon)$. Also, b is a meaningful L_∞ estimate in that for all $\epsilon > 0$, $\gamma(\epsilon) > 0$.

It is easy to see that statistical access to the noise variable ν is sufficient to obtain an accurate estimate of the value of γ for a particular value of ϵ. We now revisit our definition of an Occam approximation to define the notion of a strongly Occam approximation.

For a function f, let $band(f, \epsilon)$ denote the set of points within ϵ of f, i.e., $band(f, \epsilon) = \{(x, y) : |y - f(x)| \leq \epsilon\}$. For a class F, $band(F, \epsilon)$ is the class of all band sets for the functions in F, i.e., $band(F, \epsilon) = \{band(f, \epsilon) | f \in F\}$. A class of sets F is said to shatter a set S if the set $\{f \cap S : f \in F\}$ is the power set of S. $D_{VC}(F)$ denotes the Vapnik–Chervonenkis dimension of F and is the cardinality of the largest set shattered by F. The Vapnik–Chervonenkis dimension is a combinatorial measure that is useful in establishing the learnability of sets [9], [21].

Let Q be an approximation algorithm with respect to the L_∞ metric. Fix $m, t \in \mathbf{N}$, and $\epsilon > 0$. Let $\hat{G}$ be the class of all functions identified in the output of Q when its input ranges over all collections S of m examples such that $l_{\min}(S, \epsilon, L) \leq t$. Q is said to be strongly Occam if there exists a function $q(a, b)$ that is polynomial in a and $1/b$ and $r : \mathbf{N} \to \mathbf{N}$ such that $r(m)$ is $O(m^\alpha)$, for fixed $0 \leq \alpha < 1$ and for all m, t, ϵ, and $\zeta > 0$, $D_{VC}(band(\hat{G}, \zeta)) \leq q(t, 1/\zeta)r(m)$.

Example 3. Let F be the Baire functions and G the class of piecewise linear functions with complexity measure the number of pieces.

Consider the following approximation algorithm Q with respect to the L_∞ metric. Given is a collection of samples $S = \{(x_1, y_1),\ (x_2, y_2), \ldots,\ (x_m, y_m)\}$ and tolerance ϵ. Using the linear time algorithm of [15], [28], construct a piecewise linear function g such that $|g(x_i) - y_i| \leq \epsilon$ for each of the samples in S and g consists of the fewest number of pieces over all such functions.

Claim 3. Algorithm Q is strongly Occam.

Proof. Fix the number of samples m, tolerance ϵ, and complexity bound t. For the set of examples S, $l_{\min}(S, \epsilon, L_2)$ is the fewest number of pieces in any piecewise linear function g such that $L_\infty(g, S) \leq \epsilon$. By construction, Q outputs such a function. Thus $\hat{G}$ is a subset of the class of all piecewise linear functions of t pieces. By Claim 4, for all ζ, $D_{VC}(band(\hat{G}, \zeta)) \leq 7t$, and the claim follows.

Claim 4. If F is the class of piecewise linear functions of at most t pieces, for all ζ, $D_{VC}(band(F, \zeta)) \leq 7t$.

Proof. Assume that we are given a set S of more than $7t$ points that is shattered by $band(F, \zeta)$. Let $S = \{(x_1, y_1),\ (x_2, y_2), \ldots,\ (x_m, y_m)\}$, where the x_i are in increasing order. We shall construct a subset S_1 of S that is not induced by any set in $band(F, \zeta)$, thereby contradicting the assumption that $band(F, \zeta)$ shatters S. Start with $S_1 = \{(x_1, y_1),\ (x_7, y_7)\}$. Now some three of (x_i, y_i) for $i = 2, 3, \ldots, 6$ must lie on the same side of the line joining (x_1, y_1) and (x_7, y_7). Call these points a, b, and c, in order of their x coordinate. If b is within the quadrilateral formed by (x_1, y_1), (x_7, y_7), a, and c, add a and c to S_1, else add b to S_1. Repeat this procedure with the rest of the points in S. The resulting set S_1 is such that no function g of fewer than $m/7$ pieces is such that $band(g, \zeta)$ picks out S_1. This is a contradiction and hence the claim.

We leave it to the reader to show that the following classes also possess efficient strongly Occam approximation algorithms:

(1) The polynomials with complexity measure the highest degree (we can construct a strongly Occam approximation via linear programming).

(2) The trigonometric interpolants with complexity measure the highest frequency (we can construct a strongly Occam approximation via linear programming).

(3) The piecewise constant functions with complexity measure the number of pieces (we can construct a greedy approximation algorithm that is strongly Occam).

In Claim 5 we will show that every strongly Occam approximation is an Occam approximation as well, confirming that the strong notion is indeed stronger. In order to prove the claim, we need the following lemma.

LEMMA 5.2 (see [29]). *Let X be a finite set and let F be a set of subsets of X, with $|X| > d = D_{VC}(F)$. Then,*

$$|F| \leq |X|^d. \tag{5.31}$$

Claim 5. If Q is a strongly Occam approximation for a class F, then it is also an Occam approximation with respect to the L_∞ metric.

Proof. Let Q be a strongly Occam approximation for a class G of unit envelope. Fix $m, t \in \mathbf{N}$, $\epsilon > 0$, and $\zeta > 0$. Let $\hat{G}$ be the class of all functions identified in the output of Q when its input ranges over all collections S of m examples such that $l_{\min}(S, \epsilon, L_\infty) \leq t$. Let $X = \{x_1,\ x_2, \ldots, x_m\}$ be a set of m points. We will construct a ζ-cover C of $\hat{G}$ on X for L_∞. The functions in C will be pairwise ζ apart in the L_∞ sense.

```
ALGORITHM.
Pick any $f \in \hat{G}$ and initialize $C = \{f\}$
while there exists $f \in \hat{G}$ such that for all $g \in C$, $L_\infty(f, g, X) > \zeta$
        $C = C \cup \{f\}$.
end
```

When the algorithm halts, for every $f \in \hat{G}$ there exists some $g \in C$ such that $L_\infty(f, g, X) \leq \zeta$. Hence C is indeed a ζ-cover of $\hat{G}$. Furthermore, for every pair of functions f and g in C,

$$L_\infty(f, g, X) > \zeta. \tag{5.32}$$

Since C is a ζ-cover,

$$\log(N(\hat{G}, \zeta, m, L_\infty)) \leq |C|. \tag{5.33}$$

Let Y be the set $\{-n\zeta, -(n-1)\zeta, \ldots, 0, \zeta, 2\zeta, \ldots, n\zeta\}$, where $n = \lfloor 1/\zeta \rfloor$. We construct a class of sets D from C as follows:

$$D = \{band(f, \zeta/2) \cap X \times Y : f \in C\}. \tag{5.34}$$

By virtue of (5.32), the sets in D are all distinct and in one-to-one correspondence with the functions of C. Also, every set of points shattered by D will also be shattered by $band(\hat{G}, \zeta/2)$, and hence $D_{VC}(D) \leq D_{VC}(band(\hat{G}, \zeta/2))$. Let $d = D_{VC}(band(\hat{G}, \zeta/2))$. By Lemma 1,

$$|D| \leq |X \times Y|^d = (2mn)^d \leq (2m/\zeta)^d. \tag{5.35}$$

Combining (5.33) and (5.35), we get

$$\log(N(\hat{G}, \zeta, m, L_\infty)) \leq \log(|C|) = \log(|D|) \leq d \log(2m/\zeta). \tag{5.36}$$

Since Q is strongly Occam, d is bounded as in the definition of strongly Occam. Combining this with (5.36), it follows that Q is an Occam approximation as well.

THEOREM 5.3. *Let F be the target class and G the hypothesis class with complexity measure l, where F and G are of envelope 1 and $b+1$, respectively. Assume a noise source of known L_∞ measure per our assumptions. Let Q be a strongly Occam approximation for G, with characteristic (q, r). Then, there exists a learning algorithm for F with expected sample complexity polynomial in $1/\epsilon$, $1/\delta$, b, and $l_{\min}(f, \epsilon/4, P_u, L_1)$, where P_u is the uniform distribution on $[0, 1]$.*

Proof. Let f be the target function, and let (q, r) be the characteristic of the Occam approximation Q. We claim that Algorithm A_3 is a learning algorithm for F with respect to the L_1 metric.

```
ALGORITHM $A_3$.
input $\epsilon, \delta, b$;
begin
    $t = 2$;
    $\eta = \frac{\gamma(\epsilon/4)\epsilon}{2(b+2)}$;
    repeat forever
        let $m$ be the least integer satisfying
        $m \geq (16/\eta)q(t, 1/(b+\epsilon/4))r(m)\log(m)$;
        make $m$ calls of EXAMPLE to get collection $S$;
        let $g = Q(S, b+\epsilon/4)$;
        let $m_1 = \frac{16t^2}{\eta^2\delta}$;
        make $m_1$ calls of EXAMPLE to get collection $S_1$;
        if no more than a fraction $(3/4)\eta$ of $S_1$
        is outside $band(g, b+\epsilon/4)$then output $g$ and halt;
        else $t = t+1$;
    end
end
```

For a particular function g, let $\mu(g, \beta)$ be the probability that a call of EXAMPLE will result in an example outside $band(g, b+\beta)$. We now estimate $\mu(g, \epsilon/4)$ when g

is such that $L_1(f, g, P) > \epsilon$. For such g, since F is of envelope 1 and G is of envelope $b+1$, $|f - g| \leq b + 2$ and

$$\int_{|f-g|>\epsilon/2} dP \quad > \epsilon/(2(b+2)). \tag{5.37}$$

It follows that

$$\mu(g, \epsilon/4) = Pr\{f(x) + \nu \notin band(g, b + \epsilon/4)\} \tag{5.38}$$
$$> Pr\{f(x) - g(x) > \epsilon/2 \;\&\; \nu \in [b, b - \epsilon/4]\} \tag{5.39}$$
$$+ Pr\{g(x) - f(x) > \epsilon/2 \;\&\; \nu \in [-b, -b + \epsilon/4]\} \tag{5.40}$$
$$\geq \min\left(Pr\{\nu \in [b - \epsilon/4, b]\},\ Pr\{\nu \in [-b, -b + \epsilon/4]\}\right) \tag{5.41}$$
$$\times Pr\{|f - g| > \epsilon/2\} \tag{5.42}$$
$$\geq \gamma(\epsilon/4)\epsilon/(2(b+2)) = \eta. \tag{5.43}$$

Let $\mu_1(g, \beta)$ denote the fraction of S_1 that is outside $band(g, b + \beta)$. By Chebyshev's inequality,

$$Pr\{|\mu_1(g, \epsilon/4) - \mu(g, \epsilon/4)| > \eta/4\} \leq \frac{16}{\eta^2 m_1}. \tag{5.44}$$

Since $m_1 = 16t^2/(\eta^2\delta)$ and the algorithm halts only when $\mu_1(g, \epsilon/4) \leq (3/4)\eta$, when the algorithm halts

$$Pr\{\mu(g, \epsilon/4) > \eta\} \leq \frac{\delta}{t^2}. \tag{5.45}$$

Summing over all $t \geq 2$, we get that the probability that the algorithm halts with $\mu(g, \epsilon/4) > \eta$ is at most δ. It follows that the probability that the algorithm halts with $L_1(g, f, P) > \epsilon$ is at most δ. Let P_u be the uniform distribution on $[0, 1]$. We now show that when $t \geq t_0 = l_{\min}(f, \epsilon/4, P_u, L_\infty)$ the algorithm halts with probability at least 1/4.

Let $t \geq t_0$ at a particular iteration, and let $\hat{G}$ be the class of all functions that Q could output during that iteration. Since $\nu \in [-b, +b]$, it is clear that $l_{\min}(S, b + \epsilon/4, L_\infty) \leq t_0 \leq t$. Since Q is strongly Occam, $D_{VC}(band(\hat{G}, b + \epsilon/4)) \leq q(t, 1/(b + \epsilon/4))r(m)$. By Theorem 4.3 of [21], for $m \geq 8/\eta$, the probability that m random samples will all fall in $band(g, b + \epsilon/4)$ for any $g \in \hat{G}$ such that $\mu(g, \epsilon/4) > \eta/2$ is at most

$$2 \sum_{i=0}^{d} \binom{2m}{i} 2^{-\eta m/4}. \tag{5.46}$$

Hence, if m is chosen so that the above probability is less than 1/2, then with probability at least 1/2, $Q(S, b+\epsilon/4)$ will return a function g such that $\mu(g, \epsilon/4) \leq \eta/2$. Indeed

$$m \geq \frac{16}{\eta} q(t, 1/(b + \epsilon/4))r(m)\log(m) \tag{5.47}$$

suffices. By assumption, $r(m)$ is $O(m^\alpha)$, for some $0 \leq \alpha < 1$. It follows that $r(m)\log(m)$ is $O(m^\beta)$, for some $0 \leq \beta < 1$, and there exists m satisfying (5.47).

We now estimate the probability the algorithm halts when Q returns a function g satisfying $\mu(g, \epsilon/4) \leq \eta/2$. Once again by Chebyshev's inequality,

$$Pr\{|\mu_1(g, \epsilon/4) - \mu(g, \epsilon/4)| > \eta/4\} \leq \frac{16}{\eta^2 m_1}. \tag{5.48}$$

Given that $\mu(g, \epsilon/4) \leq \eta/2$ and that $m_1 = 16t^2/(\eta^2\delta)$, we have

$$Pr\{\mu_1(g, \epsilon/4) > (3/4)\eta\} \leq \frac{\delta}{t^2} \leq 1/2. \tag{5.49}$$

Hence, we have $Pr\{\mu_1(g, \epsilon/4) \leq (3/4)\eta\} \geq 1/2$. That is, if the function g returned by Q is such that $\mu(g, \epsilon/4) \leq \eta/2$, then A_1 will halt with probability at least $1/2$. We have therefore shown that when $t \geq t_0$, the algorithm halts with probability at least $1/4$. Hence, the probability that the algorithm does not halt within some $t > t_0$ iterations is at most $(3/4)^{t-t_0}$, which goes to zero with increasing t.

5.3. Application to filtering. An important problem in signal processing is that of filtering random noise from a discretely sampled signal [24]. The classical approach to this problem involves manipulating the spectrum of the noisy signal to eliminate noise. This works well when the noise-free signal has compact spectral support, but is not effective otherwise. However, the noise-free signal may have compact support in some other representation, where the filtering may be carried out effectively.

When we examine Algorithm A_2, we see that the sampling rate m varies roughly as $q()r(m)/\epsilon^2$, or $\epsilon^2 \approx q()r(m)/m$. In a sense, $q()r(m)$ is the support of the noise-free target function in the hypothesis class G. While spectral filters choose G to be the trigonometric interpolants, we are free to choose any representation, aiming to minimize $q()r(m)$. Furthermore, the Occam approximation Q need not manipulate its input samples in a linear fashion, as is the case with spectral filters. In this sense, our results offer the first general technique for the construction of nonlinear filters.

In the practical situation, our results can be interpreted thus: Pass the samples of the noisy signal through a data compression algorithm, allowing the algorithm an approximation error equal to the noise strength. The decompressed samples compose the filtered signal and are closer to the noise-free signal than the noisy signal. Implementations of this are pursued in [23].

6. Conclusion. We showed that the principle of Occam's Razor is useful in the context of probably approximate learning functions on the reals, even in the presence of arbitrarily large additive random noise. The latter has important consequences in signal processing in that it offers the first general technique for the design and construction of nonlinear filters.

Acknowledgments. Thanks to A. Lempel for discussions. Particular thanks to the referees for their extraordinary patience in carefully reading the manuscript and offering numerous suggestions for improvement.

REFERENCES

[1] N. Alon, S. Ben-David, N. Cesa-Bianchi, and D. Haussler, *Scale-sensitive dimensions, uniform convergence and learnability*, in Proceedings of the 34th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1993, pp. 292–301.

[2] D. Angluin and P. Laird, *Learning from noisy examples*, Mach. Learning, 2 (1988), pp. 319–342.

[3] M. Anthony and N. Biggs, *Computational Learning Theory: An Introduction*, Cambridge University Press, London, UK, 1992.

[4] M. Anthony, P. L Bartlett, Y. Ishai, and J. Shawe-Taylor, *Valid generalization from approximate interpolation*, Combin. Probab. Comput., 5 (1996), pp. 191–214.

[5] S. Ar, R. Lipton, R. Rubenfeld, and M. Sudan, *Reconstructing algebraic functions from mixed data*, in Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 503–511.

[6] P. L. Bartlett, P. M. Long, and R. C. Williamson, *Fat shattering and the learnability of real-valued functions*, in Proceedings of the 7th ACM Symposium on Computer Learning Theory, New Brunswick, NJ, 1994, pp. 299–310.

[7] M. Ben-Or and P. Tiwari, *A deterministic algorithm for sparse multivariate polynomial interpolation*, in Proceedings of the 20th ACM Symposium on Theory of Computing, 1988, pp. 394–398.

[8] E. Berlekamp and L. Welch, *Error Correction of Algebraic Block Codes*, U.S. Patent 4,633,470, 1970.

[9] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, *Learnability and the Vapnik-Chervonenkis dimension*, J. ACM, 36 (1991), pp. 929–965.

[10] W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. II, John Wiley, New York, 1966.

[11] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, MD, 1983.

[12] D. Y. Grigoriev, M. Karpinski, and M. F. Singer, *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19 (1990), pp. 1059–1063.

[13] D. Haussler, *Generalizing the PAC model for neural net and other learning applications*, Inform. and Comput., 100 (1992), pp. 78–150.

[14] D. Haussler and P. Long, *A Generalization of Sauer's Lemma*, Tech. Report UCSC-CRL-90-15, University of California, Santa Cruz, CA, 1990.

[15] H. Imai and M. Iri, *An optimal algorithm for approximating a piecewise linear function*, J. Inform. Processing, 9 (1986), pp. 159–162.

[16] A. H. Jazwinski, *Stochastic Processes and Filtering Theory*, Academic Press, New York, 1970.

[17] M. Kearns and M. Li, *Learning in the presence of malicious errors*, SIAM J. Comput., 22 (1993), pp. 807–837.

[18] M. Kearns and R. E. Schapire, *Efficient distribution-free learning of probabilistic concepts*, in Proceedings of the IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1990, pp. 382–391.

[19] V. Krishnan, *Non-Linear Filtering and Smoothing*, John Wiley, New York, 1984.

[20] B. K. Natarajan, *On learning sets and functions*, Mach. Learning, 4 (1989), pp. 67–97.

[21] B. K. Natarajan, *Machine Learning: A Theoretical Approach*, Morgan Kaufmann, San Mateo, CA, 1991.

[22] B. K. Natarajan, *Occam's razor for functions*, in Proceedings of the Sixth ACM Symposium on Computer Learning Theory, 1993, pp. 370–376.

[23] B. K. Natarajan, *Filtering random noise from deterministic signals via data compression*, IEEE Trans. Signal Processing, (Nov., 1995).

[24] A. V. Oppenheim and R. Schafer, *Digital Signal Processing*, Prentice–Hall, Englewood Cliffs, NJ, 1974.

[25] A. Papoulis, *Probability, Random Variables and Stochastic Processes*, McGraw–Hill, New York, 1965.

[26] D. Pollard, *Convergence of Stochastic Processes*, Springer-Verlag, New York, 1984.

[27] R. Sloan, *Types of noise in data for concept learning*, in Proceedings of the Workshop on Computational Learning Theory, 1988, pp. 91–96.

[28] S. Suri, *On some link distance problems in a simple polygon*, IEEE Trans. Robotics Automation, 6 (1988), pp. 108–113.

[29] V. N. Vapnik and A. Y. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory Probab. Appl., 16 (1970), pp. 264–280.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 728–760

NAVIGATION IN HYPERTEXT IS EASY ONLY SOMETIMES[*]

MARK LEVENE[†] AND GEORGE LOIZOU[‡]

Abstract. One of the main unsolved problems confronting Hypertext is the navigation problem, namely, the problem of having to know where you are in the database graph representing the structure of a Hypertext database, and knowing how to get to some other place you are searching for in the database graph. In order to tackle this problem we introduce a formal model for Hypertext. In this model a Hypertext database consists of an information repository, which stores the contents of the database in the form of pages, and a reachability relation, which is a directed graph describing the structure of the database. The notion of a trail, which is a path in the database graph describing some logical association amongst the pages in the trail, is central to our model.

We define a Hypertext query language for our model based on a subset of propositional linear temporal logic, which we claim to be a natural formalism as a basis for establishing navigation semantics for Hypertext. The output of a trail query in this language is the set (which may be infinite) of all trails that satisfy the query. We show that there is a strong connection between the output of a trail query and finite automata in the sense that, given a Hypertext database and a trail query, we can construct a finite automaton representing the output of the query, which accepts a star-free regular language. We show that the construction of the finite automaton can be done in time exponential in the number of conjunctions, between the subformulas of the trail query, plus one.

Given a Hypertext database and a trail query, the problem of deciding whether there exists a trail in the database that satisfies the trail query is referred to as the model checking problem. We show that, although this problem is NP-complete for different subsets of our query language, it can be solved in polynomial time for some significant special cases. Thus the navigation problem can only be efficiently solved in some special cases, and therefore in practice Hypertext systems could include algorithms which return randomized and/or fuzzy solutions.

Key words. Hypertext, navigation, trail query, temporal logic, finite automata, computational complexity

AMS subject classifications. 68P15, 68P20, 68Q15

PII. S0097539795282730

1. Introduction.

Traditional text, for example in book form, has a single linear sequence defining the order in which the text is to be scanned. In contrast Hypertext [CONK87] (or more generally Hypermedia; see [HALA88]) is text (which may contain multimedia) that can be read nonsequentially. Hypertext presents several different options to readers, and the individual reader chooses a particular sequence at the time of reading.

The inspiration for Hypertext comes from the *memex* machine proposed by Bush [BUSH45] (see also [NYCE89]). The memex is a "sort of mechanized private file and library" which supports "associative indexing" and allows navigation whereby "any item may be caused at will to select immediately and automatically another." Bush emphasizes that "the process of tying two items together is an important thing." In addition, by repeating this process of creating links we can form a *trail* which can be traversed by the user, in Bush's words, "when numerous items have been thus joined together to form a trail they can be reviewed in turn." Hypertext can be viewed as the formalization and realization of Bush's original ideas.

[*]Received by the editors March 8, 1995; accepted for publication (in revised form) October 19, 1998; published electronically December 7, 1999.
http://www.siam.org/journals/sicomp/29-3/28273.html

[†]University College London, Gower Street, London WC1E 6BT, UK (mlevene@cs.ucl.ac.uk).

[‡]Birkbeck College, Malet Street, London WC1E 7HX, UK (george@dcs.bbk.ac.uk).

A *Hypertext database* [STOT89, TOMP89, FRIS92] is formalized as a directed graph [BUCK90] (called the *database graph*) whose nodes represent textual units of information (called *pages*) and whose arcs (also called *links*) allow the reader to navigate from an anchor node to a destination node. The structure of a Hypertext database changes over time and thus differs from traditional databases, such as relational databases [ULLM88], which have a regular structure defined by a relational database schema. It is evident that a Hypertext database can be implemented on top of a relational database provided it has the facilities to store and retrieve textual objects. A comparison of the functional characteristics of several Hypertext systems can be found in [SCHN88].

The process of traversing links and following a *trail* of information in a Hypertext database is called *navigation* (or alternatively *link following*). In graph-theoretic terms a trail in a Hypertext database is a *path* in the database graph (we allow a node to occur more than once in a path; paths are called *walks* in [BUCK90]).

Navigating through a Hypertext database leads to the problem of getting "lost in hyperspace" [CONK87, VAND88], which is the problem of having to know where you are in the database graph and knowing how to get to some other place that you are searching for in the database graph. From now on we will refer to this fundamental problem as the *navigation problem.*

In order to solve the navigation problem we can augment link following with a *query*-based access mechanism [HALA88]. Such a mechanism allows users to specify the characteristics of the information they are searching for and then to obtain the output of their query from the Hypertext system. Two types of querying mechanism are the following:

- *Content-based search*, which typically uses index-based information retrieval technology. When searching by content, pages are searched independently of their association with other pages in the database.
- *Structure-based search*, which extends content-based search by additionally specifying a description of a subgraph [BUCK90] of the database graph. When searching by structure, contents of pages are searched in association with their linkage with other pages in the database according to the said specified subgraph.

We will refer to the formal semantics of the query mechanism used in a Hypertext system as its *navigation semantics.*

In a more general context of a Hypertext system the process of finding and examining the text pages associated with destination nodes is called *browsing* [CONK87]. The *browser* is the component of a Hypertext system that helps users search for the information they are interested in by graphically displaying the relevant parts of the database and providing contextual and spatial cues with the use of *navigational aids* such as *maps* and *guides* [FRIS92]. A set of tools that aid navigation by performing a structural analysis of the database graph is described in [RIVL94].

Herein, we are only interested in the navigation semantics of structure-based search as a means of attempting to solve the navigation problem and thus assume that navigational aids are available within the browser and are independent of the navigation semantics.

Example 1. *The database graph of a Hypertext database, which models a simple* Teletext *system storing pages of information according to the following topics: news, sports, travel, and weather, is shown in Figure* 1.1. *In a more comprehensive example the nodes in this database graph can be expanded into subgraphs; for instance, the*

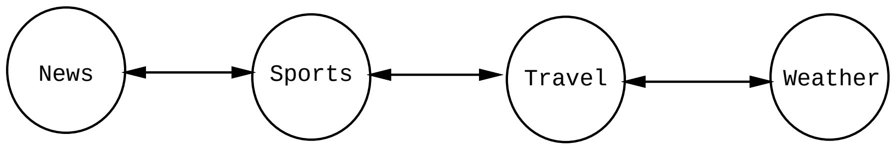


FIG. 1.1. *A simple database graph.*

node labeled news *can be expanded into several pages, one for regional news, one for countrywide news, one for European news, and one for worldwide news.*

In our Hypertext model we separate the contents of nodes (i.e., the pages) from the database graph. We define a Hypertext database to be an ordered pair $\langle I, R\rangle$, where I is an *array* [TREN73] of pages, called the *information repository*, and R is a binary relation in the indices of I, called the *reachability relation*; the indices of I are referred to as the *page numbers* of I. Thus arrays are used to provide an indexing mechanism for pages. This leads us to define a *trail* as an array of page numbers corresponding to a path in the directed graph representing R.

In the navigation semantics that we propose, the notion of a trail is central. The output of a query is the set of trails in the Hypertext database satisfying the query specification; each trail in the output of a query is called an *answer* of the query. Therefore, a query consists of two orthogonal parts:

(i) logical formulas composed of predicates to be satisfied by the contents of pages in the information repository, and

(ii) the order in which the pages, satisfying the logical formulas, are to be found in the reachability relation.

An important requirement of the query language is that users should have as much flexibility as possible when posing a query according to their knowledge of the contents and structure of the Hypertext database.

This suggests that a logic of *positions* [RESC71] would be well suited as a query language for Hypertext due to the spatial interpretation of the database graph. As pointed out by Rescher and Urquhart [RESC71] there is a close connection between *positional* and *temporal* logic. Therefore, we choose to utilize *propositional linear temporal logic* (PLTL) [EMER90] as the underlying semantics for navigation, where in our context *time* actually means *position.* (We mention that temporal logic is widely employed in the very active research area of specification and verification of concurrent programs [EMER90, MANN92].)

In our context we view a Hypertext database as a *temporal structure* [EMER90], where the page numbers of I are associated with the states of the structure and the binary relation R is associated with the transition relation of the structure.

We define a query language, called *Hypertext query language* (HQL), based on a subset of PLTL and call HQL formulas *trail queries* (or HQL queries). In particular, HQL includes the temporal operators "nexttime" (denoted by $\bigcirc$) and "sometimes" (denoted by $\diamond$). A fundamental difference between the semantics of PLTL and HQL is that models of PLTL formulas are *timelines*, which are infinite paths describing computation sequences [EMER90, THOM90], while models of HQL formulas are *trails*, which are finite paths describing sequences of pages that can be traversed by users. Thus, in addition to the temporal operators $\bigcirc$ and $\diamond$, HQL supports a novel temporal operator "finaltime" (denoted by $\triangle$), which refers to the final page number in a trail. (We will show that $\triangle$ adds expressive power to the query language.)

HQL provides a natural vehicle for combining content-based querying with structure-based search via the above temporal operators. The "nexttime" operator allows the user to navigate through the pages one step at a time, the "sometimes" operator allows the user to skip as many pages as necessary in order to arrive at the next page he wishes to browse through, and the "finaltime" operator allows the user to specify the final page in a navigation session.

The output of a trail query is defined to be the set of all trails that satisfy the query, and the problem of whether there exists an answer to a trail query with respect to a Hypertext database is the problem of deciding whether the database is a model of the trail query; this problem is known as the *model checking* problem (see [EMER90]).

In order to construct the output of a query we take advantage of the strong connection between finite automata [HOPC79, PERR90] and PLTL [VARD86a, VARD86b, EMER90, THOM90] as follows.

First, we show that a Hypertext database, say, H, can be represented by a finite automaton $\mathcal{M}_H$ and that the language accepted by $\mathcal{M}_H$ is a *star-free* regular language [MCNA71, PERR90]. Second, we show that a finite automaton $\mathcal{M}_f$ can be constructed for a trail query, say, f, with respect to the Hypertext database H, and that the language accepted by $\mathcal{M}_f$ is also a star-free regular language. Finally, we show that the output of a trail query corresponds to the intersection of $\mathcal{M}_H$ and $\mathcal{M}_f$, which also accepts a star-free regular language, since star-free regular languages are closed under intersection. Thus the navigation semantics of HQL are given in terms of star-free regular languages.

In addition, we show that constructing $\mathcal{M}_H \cap \mathcal{M}_f$ can be done in time exponential in the number of conjunctions, between the subformulas of f, plus one, and in the case when the number of conjunctions between the subformulas of f is one, then the construction can be done in time polynomial in the size of H and the length of f. Thus, we give an exponential time upper bound for query evaluation in HQL, which is tractable as long as the number of conjunctions in the subformulas of trail queries is bounded by some constant.

In order to measure the difficulty of navigation in Hypertext we investigate the computational complexity of the model checking problem for HQL queries. We prove that in the general case the model checking problem is NP-complete, which is to be expected by inspecting the results in [SIST85] (see also [EMER90]). We also show that in two special cases, when trail queries do not contain any occurrences of $\diamond$, the model checking problem can be solved in polynomial time in the length of the information repository I and the length of the trail query f.

The rest of the paper is organized as follows. In section 2 we briefly survey related work. In section 3 we formalize the notion of a Hypertext database. In section 4 we present our query language, HQL. In section 5 we investigate a strong connection between the output of a trail query and finite automata that accept star-free regular languages. In section 6 we investigate the computational complexity of navigation in a Hypertext database. Finally, in section 7 we give our concluding remarks.

2. A brief survey of related work. Several researchers [STOT89, STOT92, MEND95] have recognized that the semantics of navigation in a Hypertext database can be formalized in terms of finite automata [HOPC79, PERR90].

In [MEND95] a query language, called G^+, is described in which the database graph is defined as a finite automaton, a query is defined as a regular expression, and the output of a query is computed from the intersection of the database graph and a finite automaton representing the query. In particular, the output of a query is the set of ordered pairs, (x, y), such that x and y are nodes in the database graph and there is a *simple path* from x to y (i.e., a path in which no node occurs more than once) labeled by a word which is accepted by the query. (See [CONS89] for a description of a later language, called GraphLog, which is an extension of G^+, and for examples of how GraphLog supports structure-based searching.)

It is shown that query evaluation is, in general, exponential time in the size of the input database graph and that the problem of deciding whether an ordered pair of nodes is in an answer to a given query is NP-complete. Some special cases are exhibited when query answering can be done in polynomial time; in particular, query answering is polynomial time when either the database graph is acyclic or the query is a *restricted regular expression* [MEND95]. The contents of nodes are not discussed in [CONS89, MEND95].

In contrast to HQL, the query language G^+ [MEND95] does not utilize temporal logic, which syntactically restricts trail queries to be equivalent to *star-free* regular languages. In addition, the navigation semantics of G^+ [MEND95] process only simple paths in the database graph, while the navigation semantics of HQL process trails which may not be simple paths. Finally, we claim that HQL is a more natural formalism for expressing queries in Hypertext than the language of regular expressions, since it is closer in nature to the navigational requirements of Hypertext.

In [STOT89] *Petri nets* [PETE81] are suggested as an underlying formalism for the specification of navigation semantics. Since Petri nets are inherently a concurrency model, they provide natural semantics for concurrent navigation paths. Furthermore, finite automata are a special case of Petri nets [PETE81] and thus as a special case simplified navigation semantics based on finite automata can be supported.

An important feature of the model presented in [STOT89] is the separation of content from structure, in the sense that the contents of the database are described via a mapping from the nodes of the Petri net (called *places* in Petri net terminology [PETE81]) to the actual pages of information.

Although Petri nets are amenable to formal analysis and constructing the reachability tree for a Petri net is decidable, in general, the complexity of the reachability problem is intractable; in fact, it has been shown that the reachability problem for Petri nets is EXPSPACE-hard [PETE81]; for special cases of Petri nets when the reachability problem is easier see [ESPA94]. Thus there is a trade-off between expressiveness and complexity, which implies that in practice only special cases of the reachability problem can be incorporated into the navigation semantics of the said model.

In [STOT92] a Hypertext database (called a *hyperdocument*) is viewed as a finite automaton, called the *link automaton*. A *branching temporal logic* [EMER90] language, called HTL*, is proposed for the specification of properties that should be exhibited when navigating through a Hypertext database. HTL* is based on *full branching-time logic* (CTL*) and a subset of HTL*, called HTL, is based on a subset of CTL*, called *computational tree logic* (CTL) [CLAR86, EMER90]. HTL* adds direction to CTL* formulas in order to allow both forward and backward navigation paths to be specified. The propositions of HTL* are atomic predicates, which are conditions to be satisfied at specific states of the link automaton. Correspondingly, the formulas of HTL* are viewed as assertions about the navigation paths of the link automaton. Model checking of an HTL* formula is then used in order to verify that the assertions specified by the formula are satisfied in the link automaton.

Although model checking for a CTL formula can be done in linear time in the length of the formula and the temporal structure being verified, model checking for a CTL* formula was shown to be PSPACE-complete [CLAR86]. In practice, it would be useful to investigate a language whose model checking complexity is in between HTL and HTL*, since HTL may turn out to be too restrictive.

There are several differences between the model presented in [STOT92] and our model. First, we are interested in querying a Hypertext database while Stotts, Furata, and Ruiz [STOT92] are interested in verifying that the Hypertext database satisfies a set of specifications. In fact, a query can also be viewed as a specification that is satisfied if and only if the set of trails that satisfy the query is not empty. Second, branching temporal logic is used in [STOT92] while we use linear time temporal logic. This difference is more fundamental, since in querying a Hypertext database we are only interested in the set of trails that satisfies a trail query independently of other trails in the database graph. Third, in [STOT92] results from the area of specification and verification of concurrent programs are directly used, while as mentioned in section 1 there is a fundamental difference between the semantics of PLTL and HQL, in that models of HQL queries are trails which are finite paths, while models of PLTL formulas (and also of path formulas of CTL and CTL* [CLAR86, EMER90]) are infinite paths. Lastly, Stotts, Furata, and Ruiz [STOT92] do not investigate any specific expressiveness or complexity results relating to their model and concentrate mainly on the definition of their model and showing how it can be applied to other models of Hypertext such as Trellis [STOT92] and Hyperties [RIVL94]. On the other hand, the main aim of our work is actually to investigate the expressiveness and complexity of HQL.

Recently Beeri and Kornatzky [BEER94] have proposed a query language for Hypertext databases which is based on branching temporal logic, with the provision for generalized path quantifiers (which capture natural language assertions) over the trails that satisfy a query. Query answering in their language can be computed in polynomial time in the size of the database graph, since only trails of some fixed bounded length are considered. In HQL no quantifiers are allowed, since its semantics are based on linear-time temporal logic. Moreover, we do not bound the lengths of trails under consideration, and thus as we show herein, query answering in our model cannot always be computed in time polynomial in the size of the database graph. Indeed one conclusion of our results may be that for practical purposes the lengths of trails must be bounded, but this comes at the cost of limiting the notion of a trail and thus forcing the user to tune the bounds of the lengths of trails for specific queries.

3. A formal model for Hypertext based on temporal logic. In this section we formalize our model for Hypertext which was motivated in the introduction. We begin by introducing the notation.

We denote the cardinality of a set, S, by $|S|$ and the length of a string, w, by $||w||$. In addition, we denote the maximum of two natural numbers m and n by $\max(m, n)$. We will use the O-notation for measuring the computational complexity of algorithms [GARE79]. Finally, we abbreviate "if and only if" to iff.

In the following we will be using the following disjoint primitive domains of countably infinite sets:

1. The set of all natural numbers, denoted by ω.
2. The set of all finite length strings, Σ^*, over a finite nonempty alphabet Σ; the empty string is denoted by ϵ, concatenation of two strings w and z will be denoted by wz and a string y is a substring of a string w iff there exist strings x and z such that $w = xyz$.

3. A set of attribute names (or simply attributes), denoted by $\mathcal{U}$.
4. A set of variables, denoted by $\mathcal{V}$.

DEFINITION 3.1 (projection). *We define the two* projection *operators* α *and* β, *such that* $\alpha((x, y)) = x$ *and* $\beta((x, y)) = y$, *where* (x, y) *is an ordered pair of values.*

DEFINITION 3.2 (page). *A* page *is an attribute-value pair of the form* $(A,\ w)$, *where* $A \in \mathcal{U}$ *and* $w \in \Sigma^*$.

We now give the basic definitions pertaining to arrays [TREN73].

DEFINITION 3.3 (array). *Let* $N = \{1,\ldots,n\}$ *be a finite index set, where* $n \in \omega$.

An array *is a family* $\{t_i\}$ *of* items $(i \in N)$; *each* $i \in N$ *is called an* index. (*At this stage the definition of the type of items in an array is left unspecified.*)

We will use the usual convention whereby $A[i]$ *denotes the ith item,* $A(i)$, *of an array,* A, *when* $1 \leq i \leq n$; *otherwise* $A[i]$ *is taken to be undefined. If* $A[i] = t$, *then we say that* $t \in A$.

In the special case when $n = 0$, *the array* A *is* empty *and* $\forall i \in \omega$, $A[i]$ *is taken to be undefined; we denote the empty array by* [ ].

The count *of an array* A, *denoted as* $\#A$, *is the number of items in* A, *i.e.,* $\#A = n$.

Two arrays, A_1 *and* A_2, *are* equal, *denoted as* $A_1 = A_2$, *if* $\#A_1 = \#A_2$ *and* $\forall i \in \{1, \ldots, \#A_1\}, A_1[i] = A_2[i]$.

An array A_1 *is a* suffix *of an array* A_2, *if* $\#A_1 \leq \#A_2$ *and* $\forall i \in \{1, \ldots, \#A_1\}$, $A_1[i] = A_2[(\#A_2 - \#A_1) + i]$. *We let* A^i *denote the suffix of an array,* A, *satisfying* $\#A^i = \#A - i$, *where* $i \in \{0, \ldots, \#A\}$.

An array A_1 *is a* prefix *of an array* A_2, *if* $\#A_1 \leq \#A_2$ *and* $\forall i \in \{1, \ldots, \#A_1\}$, $A_1[i] = A_2[i]$. *We let* A_i *denote the prefix of an array,* A, *satisfying* $\#A_i = i$, *where* $i \in \{0, \ldots, \#A\}$.

The concatenation *of two arrays,* A_1 *and* A_2, *denoted by* A_1A_2, *is an array* A_3 *such that* $\#A_3 = \#A_1 + \#A_2$, $A_{3_{\#A_1}} = A_1$, *and* ${A_3}^{\#A_1} = A_2$.

An array, A, *is said to be* simple *whenever* $\forall i \in \{1, \ldots, \#A\}$, *if* $i \neq j$, *then* $A[i] \neq A[j]$.

An *information repository* is a simple array of pages; i.e., it satisfies the constraint that pages are unique within the repository. The formal definition follows.

DEFINITION 3.4 (information repository). *An* information repository (*or simply a repository*) *is a simple array,* I, *whose items are pages. The indices of* I *are called the* page numbers *of* I. *We let* $\|I\|$ *denote the length of the string* $I[1]I[2]\ldots I[\#I]$.

The constraint that a repository be a simple array is similar to entity integrity [CODD79] and thus avoids duplication of information. Furthermore, the assumption that a repository is an array, which is *not* nested, is similar to the *first normal form assumption* of tuples in relational databases [LEVE99a].

A reachability relation R over a repository, I, is a set of ordered pairs of page numbers of I. The formal definition follows.

DEFINITION 3.5 (reachability relation). *A* reachability relation, R, *over a repository* I (*or simply a reachability relation if* I *is understood from context*) *is a binary relation in* $\{1, \ldots, \#I\}$.

When R *corresponds to an acyclic directed graph* [BUCK90], *we say that* R *is* acyclic; *otherwise* R *is* cyclic (*unless explicitly stated otherwise, we assume that* R *is cyclic*).

It follows that a reachability relation corresponds to a directed graph. A *trail* in a reachability relation, R, is an array of page numbers corresponding to a path in R. The full definition follows.

Definition 3.6 (trail). *A* trail*, T, in a reachability relation, R, over a repository, I (or simply a trail in R if I is understood from context), is an array of page numbers of I such that $\forall i \in \{1, \ldots, \#T-1\}, (T[i], T[i+1]) \in R$. The indices of T are called the* markers *of T.*

A trail *which is a simple array is called a* loop-free *trail. A* subtrail *of a trail, T in R, is a trail in R that is a suffix of a prefix of T (or, equivalently, a prefix of a suffix of T).*

The string induced by a trail T in R, denoted by $\rho(T)$, is defined by $\rho(T) = T[1]T[2]\ldots T[\#T]$.

In general, a trail in a reachability relation, R, corresponds to a path in the directed graph representing R. There are two special cases worth mentioning: a trail of count zero, which corresponds to the empty array and a trail of count one which corresponds to a single page number in I.

The following proposition states that the set of trails in R is *suffix closed* and *prefix closed* (cf. [EMER90]).

Proposition 3.7. *The following statements are true:*

1. *A trail T is in a reachability relation, R, iff $\forall i \in \{0, \ldots, \#T\}$, T^i is in R.*
2. *A trail T is in a reachability relation, R, iff $\forall i \in \{0, \ldots, \#T\}$, T_i is in R.*

We next define Hypertext databases.

Definition 3.8 (Hypertext database). *A* Hypertext database *(or simply a database)* H *is an ordered pair $\langle I, R\rangle$, where I is a repository and R is a reachability relation over I.*

We note that there are no constraints on the reachability relation, R . A Hypertext database $\langle I, R\rangle$ can be viewed as a *temporal structure* [EMER90], where the page numbers of I are associated with the states of the structure and the binary relation R is associated with the transition relation of the structure (we do not assume that R is total as is the case in [EMER90]).

From now on we will assume that $H = \langle I, R\rangle$ is a Hypertext database.

4. A query language for navigating in Hypertext.

In this section we define the syntax and semantics of HQL.

We will assume that strings in Σ^* are distinguished by strings beginning with lowercase letters; attribute names in $\mathcal{U}$ are distinguished by strings beginning with uppercase letters excluding X, Y, and Z; and variables are distinguished by strings beginning with the uppercase letters X, Y, or Z.

Definition 4.1 (normal and unique variables). *We assume that $\mathcal{V}$ is partitioned into two countably infinite sets of variables called* normal variables *and* unique variables.

Normal variables are distinguished by strings beginning with the uppercase letter X, and unique variables are distinguished by strings beginning with the uppercase letter Y. Variables which may be either normal or unique are distinguished by strings beginning with the uppercase letter Z.

In an interpretation, defined below, both normal and unique variables map to page numbers. However, unique variables restrict the mapping such that no two unique variables can map to the same page number.

Informally, a condition is a propositional logic formula such that its atomic formulas are binary predicates.

Definition 4.2 (condition). *Assume a countably infinite set of binary predicate letters. A* numeric term *is either a natural number or a variable and a* symbolic term *is either an attribute or a string. An* atomic formula *is an expression of the* *form $P(t_1, t_2)$, where P is a binary predicate letter, t_1 is a numeric term, and t_2 is a symbolic term. An atomic formula, $P(t_1, t_2)$, is said to be* ground *if t_1 is a natural number; otherwise if t_1 is a variable, then $P(t_1, t_2)$ is said to be* nonground.

We recursively define the class of conditions *using the following rules:*

C1. *A nonground atomic formula $P(t_1, t_2)$ is a condition.*

C2. *If C is a condition, then $\neg(C)$ is a condition.*

C3. *If C_1 and C_2 are conditions, then $(C_1 \wedge C_2)$ is a condition.*

C4. *If C is a condition, then $|\mathbf{V}| = 1$, where $\mathbf{V}$ is the set of variables appearing in C; the single variable appearing in C is called* the variable *of C.*

We note that ground atomic formulas are excluded from conditions, since pages are identified by their content and not by their page number. As usual $C_1 \vee C_2$ stands for $\neg(\neg(C_1) \wedge \neg(C_2))$. Also, when no ambiguity arises we omit parentheses in conditions. In addition, the *length* of a condition, C, is the number of symbols in C assuming that no parentheses are omitted.

An interpretation gives meaning to the predicate letters and variables in conditions such that predicate letters are mapped to polynomial-time algorithms and variables are mapped to page numbers.

Definition 4.3 (an interpretation). *An* interpretation *σ over a Hypertext database $H = \langle I, R\rangle$ (or simply an interpretation σ if H is understood from context) assigns an appropriate meaning to binary predicate letters and terms as follows:*

- *If P is a binary predicate letter, then $\sigma(P)$ is a mapping from $\omega \times (\mathcal{U} \cup \Sigma^*)$ to* {true, false} *such that $\sigma(P)$ is a polynomial-time algorithm* [GARE79] *in $||I||$.*
- *If t is a natural number, an attribute, or a string, then $\sigma(t) = t$.*
- *If Z is a variable, then $\sigma(Z) \in \{1, \ldots, \#I\}$, with the constraint that the restriction* [HALM74] *of σ to unique variables is a one-to-one mapping.*

Restricting σ to be a one-to-one mapping, when its domain is the set of unique variables, implies that two distinct unique variables are mapped by σ to distinct page numbers and this allows us to assert the inequality of page numbers.

Definition 4.4 (satisfaction in an interpretation). *Let σ be an interpretation and let C be a condition. Then we say that σ* satisfies *C, written $\sigma \models C$, provided C is true under the interpretation σ in the usual sense. Specifically,*

C1. $\sigma \models P(t_1, t_2)$ *iff* $\sigma(P)(\sigma(t_1), \sigma(t_2)) =$ true.

C2. $\sigma \models \neg(C)$ *iff* $\sigma \not\models C$.

C3. $\sigma \models (C_1 \wedge C_2)$ *iff* $\sigma \models C_1$ *and* $\sigma \models C_2$.

The following proposition follows by a straightforward induction on the length of a condition.

Proposition 4.5. *Let σ be an interpretation over a Hypertext database $H = \langle I, R\rangle$ and let C be a condition. Then $\sigma \models C$ can be evaluated in polynomial time in $||I||$ and the length of C.*

Hereafter the binary predicate letters *substr* and *att* will be utilized. For all interpretations σ over $\langle I, R\rangle$ their meaning is fixed as follows:

1. $\sigma(\mathit{substr})$ is the substring pattern matching algorithm; that is, $\sigma \models \mathit{substr}(t_1, t_2)$ iff $\sigma(t_2)$ is a substring of $\beta(\mathrm{I}[\sigma(t_1)])$.
2. $\sigma(\mathit{att})$ is the attribute equality algorithm; that is, $\sigma \models \mathit{att}(t_1, t_2)$ iff $\alpha(I[\sigma(t_1)]) = \sigma(t_2)$.

Both $\sigma(\mathit{substr})$ and $\sigma(\mathit{att})$ can be evaluated in polynomial time in $||\mathrm{I}||$ (see [SEDG90] for efficient substring pattern matching).

We next define the notion of a trail formula and its satisfaction utilizing a *tem-*

poral logic framework [EMER90]. In particular, we will employ propositional linear temporal logic (PLTL) with discrete time.

The temporal operators that will be utilized in the context of a Hypertext database are as follows: $\bigcirc$ means "nexttime" (one step at a time navigation) and $\diamond$ means "sometimes" (several steps at a time navigation). We also define an additional temporal operator, denoted by $\triangle$, which means "finaltime" (reaching the last step of navigation). For simplicity, in this paper, we do not consider the "until" temporal operator, which would add expressive power to our query language [GABB80, EMER90]. From now on, we will refer to these temporal operators as *trail operators.*

Strictly speaking, when we refer to *time* we are actually referring to a *position* in a trail. As pointed out by [RESC71] there is a close connection between *positional* and temporal logic which motivates our use of a subset of PLTL as a query language for Hypertext.

We next define trail formulas, which are similar to PLTL formulas [EMER90]. For simplicity, at this stage, we do not consider negation or disjunction in formulas.

Definition 4.6 (trail formulas). *We recursively define the class of* trail formulas (*or simply formulas*) *using the following rules:*

T1. *A condition C is a trail formula.*

T2. *If f_1 and f_2 are trail formulas, then $(f_1 \wedge f_2)$ is also a trail formula.*

T3. *If f is a trail formula, then $\diamond(f)$ is also a trail formula.*

T4. *If f is a trail formula, then $\bigcirc(f)$ is also a trail formula.*

T5. *If f is a trail formula, then $\triangle(f)$ is also a trail formula.*

When no ambiguity arises we omit parentheses in formulas.

Example 2. *We now demonstrate the usefulness of* HQL *with several example formulas and their intuitive semantics, where we assume that the substring pattern matching algorithm is case insensitive.*

1. *The formula*

$$\diamond(substr(X_1, local\ news)) \wedge \diamond(substr(X_2, world\ news)) \wedge \\ \diamond(substr(X_3, sports)) \wedge \diamond(substr(X_4, weather))$$

specifies the trails that have a page of local news, a page of world news, a page of sports news and a page with the weather information.

2. *The formula*

$$substr(X_1, UK\ news) \wedge \bigcirc(substr(X_2, South\ East\ news)) \wedge \\ \bigcirc\bigcirc(substr(X_3, London\ news)) \wedge \triangle(substr(X_4, North\ London\ news))$$

specifies the trails whose first page contains UK news, followed by a page containing South East news, followed by a page containing London news, and a final page containing North London news.

3. *The formula*

$$\diamond(substr(X_1, five\ star\ hotels) \wedge substr(X_1, Austin\ Texas)) \wedge \\ \bigcirc(substr(X_2, five\ star\ hotels) \wedge substr(X_1, Dallas\ Texas))$$

specifies the trails that have a page of five star hotels in Austin, Texas, followed by a page of five star hotels in Dallas, Texas.

4. *The formula*

$$substr(X_1, underground) \wedge \neg(substr(X_1, delayed)) \wedge \\ \triangle((substr(X_2, buses)) \wedge \neg(substr(X_2, delayed)))$$

specifies the trails whose first page gives us the information about the underground lines that are running normally and whose last page gives us the information about the bus lines that are running normally.

DEFINITION 4.7 (subformulas). *The set of subformulas of a formula, f, is defined recursively as follows:*

1. *f is a subformula of f.*
2. *If f is of the form $f_1 \wedge f_2$, then f_1 and f_2 are subformulas of f.*
3. *If f is any of the forms $\Diamond(f')$, $\bigcirc(f')$, or $\triangle(f')$, then f' is a subformula of f.*

We denote the set of all subformulas of a trail formula, f, by $Sub(f)$. The length *of a formula, f, is the number of symbols in f assuming that all conditions have the same length of one and that no parentheses are omitted. We denote the length of f by $|f|$. It therefore follows that $|Sub(f)| \leq |f|$.*

The reason we have assumed that all conditions *have the same length of one* is that, from a database point of view, the purpose of conditions is to retrieve the set of pages satisfying the condition. Herein we are not interested in the internal structure of conditions, since by Proposition 4.5 we can test whether a page satisfies a condition in polynomial time.

DEFINITION 4.8 (satisfaction of a trail formula). *Let $H = \langle I, R \rangle$ be a Hypertext database, σ be an interpretation over H, and T be a trail in R . Then we say that T* satisfies *a formula, f, with respect to H and σ (or simply T satisfies f if H and σ are understood from context), written $T \models f$, provided f is true under T . Specifically,*

T1. *$T \models C$ iff $\#T > 0, \sigma \models C$, and $\sigma(Z) = T[1]$, where Z is the variable of C.*
T2. *$T \models f_1 \wedge f_2$ iff $T \models f_1$ and $T \models f_2$.*
T3. *$T \models \Diamond(f)$ iff $\exists i \in \{0, \ldots, \#T - 1\}$ such that $T^i \models f$.*
T4. *$T \models \bigcirc(f)$ iff $\#T > 1$ and $T^1 \models f$.*
T5. *$T \models \triangle(f)$ iff $\#T > 0$ and $T^i \models f$, where $i = \#T - 1$.*

From Definition 4.8 it follows that for all trail formulas, $f, [\,] \not\models f$.

DEFINITION 4.9 (a model of a trail formula). *A Hypertext database $H = \langle I, R \rangle$ is a* model *of a trail formula, f, if $\exists T$ in R such that $T \models f$ with respect to H and some interpretation σ. Whenever H is understood from context we also say that T is a model of f.*

DEFINITION 4.10 (trail equivalence). *A formula f_1* trail implies *(or simply implies) a formula f_2, written $f_1 \Rightarrow f_2$, if*

for all Hypertext databases $\langle I, R \rangle$, for all trails T in R, if $T \models f_1$ then $T \models f_2$.

A formula f_1 is trail equivalent *(or simply equivalent) to a formula f_2, written $f_1 \equiv f_2$, if $f_1 \Rightarrow f_2$ and $f_2 \Rightarrow f_1$.*

The following proposition gives the significant equivalences and implications between HQL formulas. We note that the implications given in the proposition cannot be strengthened to equivalences.

PROPOSITION 4.11. *The following equivalences and implications are satisfied:*

1. $\Diamond(f) \equiv \Diamond\Diamond(f)$.
2. $\triangle(f) \equiv \triangle\triangle(f)$.
3. $\Diamond \bigcirc (f) \equiv \bigcirc\Diamond(f)$.
4. $\triangle(f) \equiv \triangle\Diamond(f) \equiv \Diamond\triangle(f)$.
5. $\bigcirc(f_1 \wedge f_2) \equiv \bigcirc(f_1) \wedge \bigcirc(f_2)$.
6. $\triangle(f_1 \wedge f_2) \equiv \triangle(f_1) \wedge \triangle(f_2)$.
7. $\Diamond(f_1 \wedge f_2) \Rightarrow \Diamond(f_1) \wedge \Diamond(f_2)$.

8. $f \Rightarrow \Diamond(f)$.
9. $\bigcirc(f) \Rightarrow \Diamond(f)$.
10. $\triangle(f) \Rightarrow \Diamond(f)$.
11. $\bigcirc\triangle(f) \Rightarrow \triangle(f)$.

The following definition will be useful when we consider subclasses of HQL.

DEFINITION 4.12 (op-free trail formulas). *A trail formula is* op-free *if it does not contain any occurrences of* op, *where* op $\in \{\Diamond, \bigcirc, \triangle\}$.

The $\Diamond$ operator is declarative since it does not specify the precise navigation sequence, while the $\bigcirc$ operator is procedural since it progresses navigation one step at a time. Thus $\bigcirc$-free trail queries can be viewed as declarative queries and $\Diamond$-free trail queries can be viewed as procedural queries. On the other hand, $\triangle$-free trail queries are queries which do not specify any condition on the final item of a trail which satisfies the query.

The next proposition shows that the general class of trail formulas is more expressive than the class of $\triangle$-free trail formulas. That is, if we remove $\triangle$ from HQL we lose expressive power.

PROPOSITION 4.13. *It is not the case that for all $\triangle$-free trail formulas, g, there exists a $\triangle$-free trail formula, f, such that $f \Rightarrow \triangle(g)$.*

Proof. The result follows by a straightforward induction on the length of f.

(*Basis*): If $|f| = 1$, then f is a condition, say, C. Without loss of generality, let $H = \langle I, R\rangle$ be a Hypertext database having a trail T in R with $\#T = 2$ and such that $T \models C$ but $T^1 \not\models g$. The result that $T \not\models \triangle(g)$ follows by Definition 4.8 part (T5).

(*Induction*): Assume that the result holds when $|f| = k$, where $k \geq 1$; we then need to prove that the result holds when $|f| > k$.

We now consider in turn the different cases pertaining to the structure of f:

1. If f is the trail formula $f_1 \wedge f_2$, then by the inductive hypothesis there exists a Hypertext database $H = \langle I, R\rangle$ and a trail T in R such that $T \models f_1$ and $T \models f_2$ but $T \not\models \triangle(g)$. The result follows by Definition 4.8 part (T2), since $T \models f$.
2. If f is the trail formula $\Diamond(f')$, then by the inductive hypothesis there exists a Hypertext database $H = \langle I, R\rangle$ and a trail T in R such that $T^i \models f'$ but $T^i \not\models \triangle(g)$, where $i \in \{0, \ldots, \#T - 1\}$. The result that $T \models f$ but $T \not\models \triangle(g)$ follows by Definition 4.8 part (T3) and part (T5), respectively.
3. If f is the trail formula $\bigcirc(f')$, then by the inductive hypothesis there exists a Hypertext database $H = \langle I, R\rangle$ and a trail T in R such that $\#\, T > 1$ and $T^1 \models f'$ but $T^1 \not\models \triangle(g)$. The result that $T \models f$ but $T \not\models \triangle(g)$ follows by Definition 4.8 part (T4) and part (T5), respectively. $\square$

A *trail query* is a trail formula viewed as a mapping from Hypertext databases to sets of trails, where a trail is in the output of the trail query iff it satisfies the trail formula with respect to the input Hypertext database and some interpretation. The formal definition follows.

DEFINITION 4.14 (trail query). *A* trail query (*or simply an HQL query or a query*) *is a trail formula, f, viewed as a mapping from Hypertext databases to sets of trails such that, given an input Hypertext database $H = \langle I, R\rangle$, the output $f(H)$ is defined by*

$$\{T \mid T \textit{ is a trail in } R \textit{ and } T \models f\}.$$

A trail T in R satisfying $T \models f$ is called an answer *of $f(H)$.*

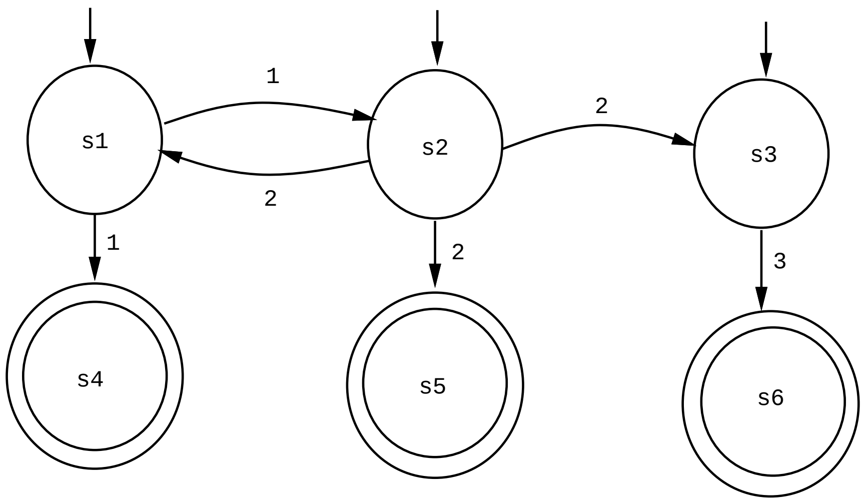


FIG. 5.1. *A finite automaton for a Hypertext database.*

5. Finite automata and trail queries. In this section we utilize the theory of finite automata [HOPC79, PERR90] in order to construct the output of a query, which in the general case may consist of a countably infinite set of trails. In particular, we exploit the strong connection between finite automata accepting star-free regular languages and PLTL [VARD86a, VARD86b, EMER90, THOM90].

DEFINITION 5.1 (Hypertext automaton). *The* Hypertext automaton *representing a Hypertext database, $H = \langle I, R\rangle$ (or simply the Hypertext automaton whenever H is understood from context), is a finite automaton defined by a quintuple of the form $\mathcal{M}_H = (\mathcal{A}, Q, \Delta, S, F)$ (or simply $\mathcal{M}$ whenever H is understood from context), where*

- $\mathcal{A} = \{1, \dots, m\}$ *is a finite alphabet with $m = \#I$.*
- $Q = \{s_1, \dots, s_m, s_{m+1}, \dots, s_{2m}\}$ *is a set of $2m$ states.*
- $\Delta \subseteq Q \times \mathcal{A} \times Q$ *is a transition relation, where $(s_j, \delta(s_j), s_k) \in \Delta$ iff $j \in \{1, \dots, m\}$ and either $k \in \{1, \dots, m\}$, with $(\delta(s_j), \delta(s_k)) \in R$, or $k = m + j$, where δ is a one-to-one and onto mapping from $\{s_1, \dots, s_m\}$ to $\mathcal{A}$ such that $\delta(s_i) = i$, $1 \le i \le m$.*
- $S = \{s_1, \dots, s_m\}$ *is the set of initial states.*
- $F = \{s_{m+1}, \dots, s_{2m}\}$ *is the set of terminal states.*

If $S = \emptyset$ and thus $F = \emptyset$, then $\mathcal{M}$ is called the empty Hypertext automaton.

We observe that the construction of $\mathcal{M}_H$ is independent of the actual contents of the pages in the repository I of H. It only depends on the cardinality, $\#I$ of I, and the reachability relation R of H. In other words, a finite automaton representing a Hypertext database is independent of the actual data stored in the repository and thus induces an equivalence class of Hypertext databases having repositories of the same cardinality and having the same reachability relation.

EXAMPLE 3. *The finite automaton $\mathcal{M}_H$ is shown in Figure 5.1, where $H = \langle I, R\rangle$, $\#I = 3$ and $R = \{(1, 2), (2, 1), (2, 3)\}$.*

DEFINITION 5.2 (the language accepted by a finite automaton). *A* path *in the finite automaton, $\mathcal{M} = (\mathcal{A}, Q, \Delta, S, F)$ (or simply a path if $\mathcal{M}$ is understood from context), is a nonempty sequence $\{(s_i, \delta(s_i), s_{i+1})\}$ $(i \in N)$ of transitions in Δ, with $N = \{1, \dots, n\}$ and $n \in \omega$. (We also say that there exists a path from*

$(s_1, \delta(s_1), s_2)$ *to* $(s_n, \delta(s_n), s_{n+1})$.) *If* $s_1 = s_{n+1}$, *then the path is a* cycle. *The string* $w = \delta(s_1)\delta(s_2)\ldots\delta(s_n)$ *is called the* label *of the path,* s_1 *is called its* origin, *and* s_{n+1} *is called its* end. *The* length *of the path is* n.

A path is successful *if its origin is in* S *and its end is in* F. *A string* $w \in \mathcal{A}^*$ *is said to be* accepted *by* $\mathcal{M}$ *if it is the label of some successful path. The language accepted by* $\mathcal{M}$, *denoted by* $\mathcal{L}(\mathcal{M})$, *is the set of all strings accepted by* $\mathcal{M}$.

When $\mathcal{M}$ is the empty Hypertext automaton, $\mathcal{L}(\mathcal{M}) = \emptyset$, that is, the empty language is accepted. Furthermore, there does not exist a Hypertext database H such that $\mathcal{L}(\mathcal{M}_H) = \{\epsilon\}$; i.e., no finite automaton representing a Hypertext database accepts the empty string. Thus only subsets of $\mathcal{A}^+ = \mathcal{A}^* - \{\epsilon\}$ are accepted by finite automata representing Hypertext databases.

We proceed to show that $\mathcal{L}(\mathcal{M}_H)$ is a *star-free* regular language [MCNA71, PERR90] (or simply a star-free language).

DEFINITION 5.3 (star-free languages). *A regular language is* star-free *if it can be generated from a finite set of strings by repeated applications of the Boolean operations, union, intersection and complementation* (*with respect to* $\mathcal{A}^*$), *together with concatenation.*

The following definition is needed in order to state an alternative characterization of star-free languages.

DEFINITION 5.4 (aperiodic regular languages). *A regular language* $\mathcal{L} \subseteq \mathcal{A}^*$ *is* aperiodic *if* $\exists n \in \omega (n > 0)$ *such that* $\forall x, y, z \in \mathcal{A}^*$,

$$xy^n z \in \mathcal{L} \textit{ iff } xy^{n+1} z \in \mathcal{L}, \tag{5.1}$$

where y^n *is the concatenation of* y *with itself* n *times.*

The following theorem states the equivalence of star-free and aperiodic regular languages [PERR90, Theorem 6.1].

THEOREM 5.5. *A regular language is* star-free *iff it is aperiodic.*

THEOREM 5.6. $\mathcal{L}(\mathcal{M}_H)$ *is a star-free language.*

Proof. Let $\mathcal{L} = \mathcal{L}(\mathcal{M}_H)$. In order to prove the result we use Theorem 5.5 to show that (5.1) is satisfied, with $n = 2$. If $y = \epsilon$, then the result holds trivially, so we assume that this is not the case. Next assume that $xy^2z \in \mathcal{L}$ and let $y = a_1 \ldots a_k = \delta(s_1)\ldots\delta(s_k)$ be the label of the path $\{(s_i, \delta(s_i), s_{i+1})\}(i \in \{1, \ldots, k\})$ in $\mathcal{M}_H$.

It follows that $s_i (i \in \{1, \ldots, k\})$ is one of the initial $\#I$ states in Q, i.e., $1 \leq i \leq \#I$, since by Definition 5.1 terminal states in $\mathcal{M}_H$ cannot be the first component of any transition. Furthermore, $s_{k+1} = s_1$, since $a_1 \ldots a_k a_1$ is a substring of y^2 and by Definition 5.1 δ is a one-to-one and onto mapping from the initial $\#I$ states in Q to $\mathcal{A}$.

Thus, the substring $a_1 \ldots a_k$ corresponds to a *cycle* [BUCK90] in the directed graph corresponding to R (where we allow a node to appear more than once in a cycle). The result follows, since it is implied that $xy^2z \in \mathcal{L}$ iff $xy^nz \in \mathcal{L}$, with $n \geq 2$. □

It is easy to demonstrate that the converse of Theorem 5.6 does not hold. For example, let $\mathcal{A} = \{a\}$ and $\mathcal{L} = \{aa\}$. $\mathcal{L}$ is obviously star-free, since it is a finite regular language, but there does not exist a Hypertext database H such that $\mathcal{L} = \mathcal{L}(\mathcal{M}_H)$. Thus, the class of languages accepted by finite automata representing Hypertext databases is a proper subclass of the class of star-free languages.

DEFINITION 5.7 (the language induced by a trail query). *The language induced by a set* $\mathbf{T}$ (*possibly countably infinite*) *of trails in* R, *denoted by* $\mathcal{L}(\mathbf{T})$, *is defined by*

$\mathcal{L}(\mathbf{T}) = \{\rho(T) | T \in \mathbf{T}\}$. *The language induced by the output* $f(H)$ *of a trail query* f *is defined to be* $\mathcal{L}(f(H))$.

It follows that, if a string $w = \delta(s_1)\delta(s_2)\ldots\delta(s_n)$ is accepted by $\mathcal{M}$, then there exists a trail, T, in R such that $\rho(T) = w$.

The following lemma states that the language induced by the output of a trail query over a Hypertext database H is a subset of the language accepted by the finite automaton representing H.

LEMMA 5.8. $\mathcal{L}(f(H)) \subseteq \mathcal{L}(\mathcal{M}_H)$, *where* f *is a trail query and* H *is a Hypertext database.*

Proof. The result follows immediately from Definitions 5.1 and 5.7 on using Definition 4.14. $\square$

The following proposition shows that the finite automaton representing a Hypertext database can be viewed as the output of a certain trail query.

PROPOSITION 5.9. *For all Hypertext databases,* $H = \langle I, R\rangle$, *there exists a trail query* f *such that* $\mathcal{L}(f(H)) = \mathcal{L}(\mathcal{M}_H)$.

Proof. Let $S = \{s_1, \ldots, s_m\}$ be the set of initial states of $\mathcal{M}_H$, and let $F = \{s_{m+1}, \ldots, s_{2m}\}$ be the set of terminal states of $\mathcal{M}_H$. Furthermore, let pages(S) = pages$(F) = \{I[i] | s_i \in S\}$ be the set of pages in I corresponding to S and F, respectively. Hereafter we assume that pages(S) = pages$(F) = \{(A_1, w_1), \ldots, (A_m, w_m)\}$.

Let f^S be the trail formula

$$(att(X_1, A_1) \wedge substr(X_1, w_1)) \vee \ldots \vee (att(X_1, A_m) \wedge substr(X_1, w_m))$$

and let f^F be the trail formula

$$\triangle((att(X_2, A_1) \wedge substr(X_2, w_1)) \vee \ldots \vee (att(X_2, A_m) \wedge substr(X_2, w_m))),$$

where X_1 and X_2 are distinct normal variables.

Finally, let T be a trail in R and let f be the trail formula $f^S \wedge f^F$. By Definition 4.8 it follows that $T \models f$ iff $T \models f^S$ and $T \models f^F$. Furthermore, $T \models f^S$ iff $T[1] \in$ pages(S) and $T \models f^F$ iff $T[\#T] \in$ pages(F). Thus T is an answer of $f(H)$ iff $T[1] \in$ pages(S) and $T[\#T] \in$ pages(F). Therefore, we have that $\mathcal{L}(f(H)) = \mathcal{L}(\mathcal{M}_H)$ holds as required. $\square$

We proceed to investigate the correspondence between finite automata and trail queries.

Recall that star-free regular languages are closed under union, intersection, and concatenation [HOPC79, PERR90]. We will assume that when taking the union, intersection, or concatenation of two finite automata *their sets of states are disjoint.* Furthermore, for convenience we will also assume that the concatenation of any finite automaton with the empty Hypertext automaton yields the empty Hypertext automaton.

We next define a useful finite automaton with respect to a repository, I, which accepts the star-free regular language $\mathcal{A}^+$; we call this finite automaton the *complete automaton.*

DEFINITION 5.10 (the complete automaton of a repository). *The complete automaton with respect to a repository,* I, *is a quintuple of the form* $\mathcal{M}_{(\gamma,I)} = (\mathcal{A}, Q_\gamma, \Delta_\gamma, S_\gamma, F_\gamma)$ (*or simply* $\mathcal{M}_\gamma$ *whenever* I *is understood from context*), *where*

- $\mathcal{A} = \{1, \ldots, m\}$ *is a finite alphabet with* $m = \#I$.
- $Q_\gamma = \{s\}$ *is the singleton set of states.*
- $\Delta_\gamma = Q_\gamma \times \mathcal{A} \times Q_\gamma$ *is the transition relation.*
- $S_\gamma = \{s\}$ *is the initial state.*

- $F_\gamma = \{s\}$ *is the terminal state.*

The finite automaton representing a trail query f, with respect to a repository I (or simply the finite automaton representing f if I is understood from context), is a quintuple of the form $\mathcal{M}_{(f,I)} = (\mathcal{A}, Q_f, \Delta_f, S_f, F_f)$ (or simply $\mathcal{M}_f$ whenever I is understood from context), whereas before $\mathcal{A} = \{1, \ldots, m\}$ with $m = \#I$.

Prior to constructing $\mathcal{M}_f$ we present an algorithm, which recursively constructs an intermediate finite automaton, $\mathcal{M}_{f'}$, in accordance with the subformulas of f.

We will assume that with each transition, (s_i, p, s_j), in the transition relation of a finite automaton, say, $\mathcal{M}$, we maintain an auxiliary set, denoted by cond($\mathcal{M}$, (s_i, p, s_j)) (or simply cond(s_i, p, s_j) if $\mathcal{M}$ is understood from context), which is initialized to the empty set. The set cond(s_i, p, s_j) will store the conditions associated with the said transition.

Definition 5.11 (the intermediate finite automaton representing a trail query). *As an intermediate step we present an algorithm, which recursively constructs a finite automaton $\mathcal{M}_{f'} = (\mathcal{A}, Q_{f'}, \Delta_{f'}, S_{f'}, F_{f'})$ with f and I given as its inputs; the algorithm is designated by τ^I when I is its input repository (or simply τ when I is understood from context). For the purpose of the algorithm we will assume that τ has an additional implicit Boolean parameter, designated by $\triangle$flag, which is initialized to* false.

The algorithm considers all of the subformulas $g \in Sub(f)$ in decreasing order of $|g|$ in accordance with the structure of g (recall that we have assumed that all conditions have the same length of one):

1. *If g is just the condition C, with variable Z, then let $\{i_1, \ldots, i_k\}$ be the largest subset of $\{1, \ldots, \#I\}$ such that there exists an interpretation σ satisfying $\sigma(Z) = i_j$, $1 \leq j \leq k$, and $\sigma \models C$.*
   If $k \geq 1$, then $\mathcal{M}_C = (\mathcal{A}, Q_C, \Delta_C, S_C, F_C)$, where
   - $Q_C = \{s_{C_1}, s_{C_2}\}$.
   - $\Delta_C = \{(s_{C_1}, i_1, s_{C_2}), \ldots, (s_{C_1}, i_k, s_{C_2})\}$ *and* $\forall j \in \{1, \ldots, k\}$, *set* cond$(s_{C_1}, i_j, s_{C_2})$ *to* C.
   - $S_C = \{s_{C_1}\}$ *is the initial state.*
   - $F_C = \{s_{C_2}\}$ *is the terminal state.*

   If $\triangle$flag is true, *then $\tau(g)$ returns $\mathcal{M}_C$; otherwise (when $\triangle$flag is* false) *$\tau(g)$ returns the concatenation of $\mathcal{M}_C$ and $\mathcal{M}_\gamma$. Finally, if the aforesaid subset is empty, i.e., $k = 0$, then $\tau(g)$ returns the empty Hypertext automaton.*
2. *If g is the formula $g_1 \wedge g_2$, such that $\tau(g_1)$ returns $\mathcal{M}_{g_1}$ and $\tau(g_2)$ returns $\mathcal{M}_{g_2}$, then $\tau(g)$ returns the intersection of $\mathcal{M}_{g_1}$ and $\mathcal{M}_{g_2}$. In addition, if* cond$((s_{i_1}, s_{i_2}), p, (s_{j_1}, s_{j_2}))$ *is in the transition relation of the intersection of $\mathcal{M}_{g_1}$ and $\mathcal{M}_{g_2}$, then* cond$((s_{i_1}, s_{i_2}), p, (s_{j_1}, s_{j_2}))$ *is set to* cond$(\mathcal{M}_{g_1}, (s_{i_1}, p, s_{j_1}))$ $\cup$ cond$(\mathcal{M}_{g_2}, (s_{i_2}, p, s_{j_2}))$.
3. *If g is the formula $\diamond(g')$ and $\tau(g')$ returns $\mathcal{M}_{g'}$, then $\tau(g)$ returns $\mathcal{M}_{g'}$ if $\triangle$flag is* true*; otherwise (when $\triangle$flag is* false) *$\tau(g)$ returns the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_{g'}$.*
4. *If g is the formula $\bigcirc(g')$ and $\tau(g')$ returns $\mathcal{M}_{g'}$, then $\tau(g)$ returns the empty Hypertext automaton if $\triangle$flag is* true, *otherwise (when $\triangle$flag is* false) *$\tau(g)$ returns the concatenation of $\mathcal{M}_\bigcirc$ and $\mathcal{M}_{g'}$, where $\mathcal{M}_\bigcirc = (\mathcal{A}, Q_\bigcirc, \Delta_\bigcirc, S_\bigcirc, F_\bigcirc)$ is defined as follows:*
   - $Q_\bigcirc = \{s_{\bigcirc_1}, s_{\bigcirc_2}\}$.
   - $\Delta_\bigcirc = \{(s_{\bigcirc_1}, 1, s_{\bigcirc_2}), \ldots, (s_{\bigcirc_1}, m, s_{\bigcirc_2})\}$, *where* $m = \#I$.
   - $S_\bigcirc = \{s_{\bigcirc_1}\}$ *is the initial state.*

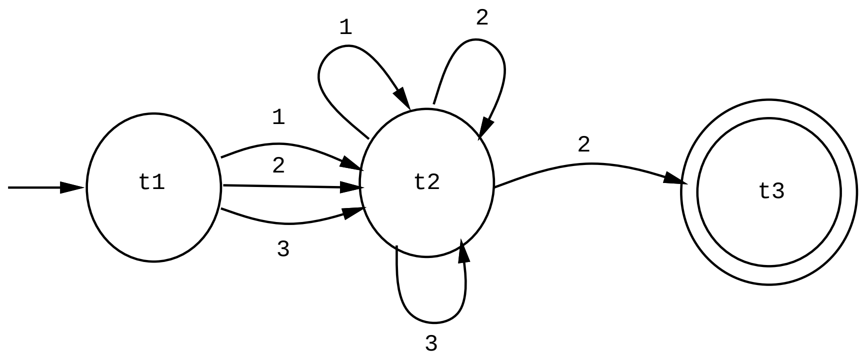


FIG. 5.2. *A finite automaton for a trail query.*

- $F_{\bigcirc} = \{s_{\bigcirc_2}\}$ *is the terminal state.*

5. *If g is the formula $\triangle(g')$, $\triangle$flag is set to* true*, and if $\tau(g')$ returns $\mathcal{M}_{g'}$, then $\tau(g)$ returns $\mathcal{M}_{g'}$ provided $\triangle$flag was* true *prior to invoking $\tau(g')$; otherwise (when $\triangle$flag was* false *prior to invoking $\tau(g')$) $\tau(g)$ returns the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_{g'}$.*

EXAMPLE 4. *Let f be the trail query $\bigcirc\triangle(C)$, with C being a condition such that the only interpretation σ over H, with $\sigma \models C$, satisfies $\sigma(Z) = 2$, where Z is the variable of C and 2 is a page number of I. In addition, let I be the information repository of the Hypertext database H of Example* 3*. The finite automaton for $\mathcal{M}_{f'}$ is shown in Figure* 5.2*.*

It can be verified that the regular language accepted by the finite automaton shown in Figure 5.2 *would remain the same were f to be the trail formula $\bigcirc\diamond\triangle(C)$. This can also be deduced from Proposition* 4.11 *part* (4)*.*

We next prove that the language accepted by $\tau(f)$ is a star-free regular language.

LEMMA 5.12. *$\mathcal{L}(\tau(f))$ is a star-free regular language.*

Proof. The result follows by a straightforward induction on the length of f, observing that $\mathcal{L}(\mathcal{M}_\gamma) = \mathcal{A}^+$ is star-free and by the fact that, due to the construction of $\mathcal{M}_g$ via τ, $\mathcal{L}(\mathcal{M}_g)$ is also star-free for all subformulas g of f. (Recall that a finite regular language is star-free and that a finite automaton having no cycles accepts a finite regular language.) □

The following lemma gives an upper bound on the cardinality of the set of states of the finite automaton $\tau(f)$ constructed by the algorithm given in Definition 5.11. Note that this upper bound is independent of the repository, I, and is exponential only in the number of conjunctions, between subformulas of f, plus one.

LEMMA 5.13. *$|Q_{f'}| \leq O(|f|)^{\wedge(f)}$, where f is a trail formula, $\tau(f) = \mathcal{M}_{f'} = (\mathcal{A}, Q_{f'}, \Delta_{f'}, S_{f'}, F_{f'})$ and $\wedge(f)$ is the number of conjunctions, between subformulas of f, plus one.*

Proof. We prove the result by induction on $\wedge(f)$. We observe that given two finite automata, whose state sets are Q_1 and Q_2, when we concatenate, or correspondingly, intersect the two finite automata, then the cardinality of the state set of the resulting finite automaton is $|Q_1| + |Q_2|$, or correspondingly, $|Q_1| \cdot |Q_2|$ [HOPC79, PERR90].

Recall that we have assumed that all conditions in f have the same length of one.

(*Basis*): If $\wedge(f) = 1$, then f has no conjunctions. The result that $|Q_{f'}| \leq O(|f|)$ follows by a straightforward induction on the length of f by inspecting Definition 5.11

in accordance with the structure of f.

(*Induction*): Assume that the result holds when $\wedge(f) = k$, where $k \geq 1$; we then need to prove that the result holds when $\wedge(f) = k + 1$.

It follows that f must be of the form $op_1(\dots op_q(g)\dots)$, where $op = op_1 \dots op_q$ is a sequence of trail operators, with $q \geq 0$, and g is a subformula of f of the form $g_1 \wedge g_2$, satisfying $\wedge(g_1) \leq k$ and $\wedge(g_2) \leq k$.

By part (2) of the description of τ in Definition 5.11 $|Q_g| \leq |Q_{g_1}| \cdot |Q_{g_2}|$, where Q_{g_1} and Q_{g_1} are the state sets of the finite automata $\mathcal{M}_{g_1}$ and $\mathcal{M}_{g_2}$, respectively. Furthermore, by the inductive hypothesis $|Q_{g_1}| \leq O(|g_1|)^{\wedge(g_1)}$ and $|Q_{g_2}| \leq O(|g_2|)^{\wedge(g_2)}$. It therefore follows that $|Q_g| \leq O(|f|)^{\wedge(f)}$, since $|g_1| + |g_2| < |f|$ and $\wedge(g_1) + \wedge(g_2) = \wedge(f)$.

The result now follows by a straightforward induction on the number, q, of trail operators in op by inspecting Definition 5.11 according to the three different cases where the first operator, op_1, is $\Diamond, \bigcirc$, or $\triangle$. $\square$

In order to conclude the construction of $\mathcal{M}_f$ we need to take into account the semantics of normal and unique variables. To accomplish this we use the auxiliary sets $\text{cond}(s_i, p, s_j)$ and modify the output of $\tau(f)$ so that it satisfies the constraints enumerated below.

First, we define some convenient terminology. A *time unit* of a trail formula f (or simply a time unit whenever f is understood from context) is defined to be a transition, (s_i, p, s_j), in the transition relation of the finite automaton $\tau(f)$, where p is a page number of I and s_i, s_j are states. The variable of a condition, $C \in \text{cond}(s_i, p, s_j)$, is said to be *appearing* at the time unit (s_i, p, s_j).

The following two constraints must be enforced on variables appearing at time units:

1. No two unique variables, which are distinct, can appear at two (not necessarily distinct) time units having the same page number (recall Definition 4.1 and the comment that follows).
2. If the same variable appears at two distinct time units, then its corresponding interpretations must be the same.

In order to formalize the above constraints, let $\varphi(\text{cond}(s_i, p, s_j))$ denote the set of variables of the conditions contained in $\text{cond}(s_i, p, s_j)$, where (s_i, p, s_j) is a time unit of a trail formula f. Thus the following conditions, corresponding to the above two constraints, must be enforced in the finite automaton $\tau(f)$.

V1. For all (not necessarily distinct) time units (s_{i_1}, p, s_{j_1}) and (s_{i_2}, p, s_{j_2}), if there exists a path from (s_{i_1}, p, s_{j_1}) to (s_{i_2}, p, s_{j_2}), then there do not exist two unique variables, which are distinct, included in $\varphi(\text{cond}(s_{i_1}, p, s_{j_1})) \cup \varphi(\text{cond}(s_{i_2}, p, s_{j_2}))$.

V2. For all distinct time units, (s_{i_1}, p_1, s_{j_1}) and (s_{i_2}, p_2, s_{j_2}), if there exists a path from (s_{i_1}, p_1, s_{j_1}) to (s_{i_2}, p_2, s_{j_2}) and $\varphi(\text{cond}(s_{i_1}, p_1, s_{j_1})) \cap \varphi(\text{cond}(s_{i_2}, p_2, s_{j_2})) \neq \emptyset$, then $p_1 = p_2$.

We now present a further algorithm, designated by π, that takes as its input the finite automaton $\tau(f)$ and transforms it into the desired finite automaton $\mathcal{M}_f$.

ALGORITHM 1 ($\pi(f)$).

```
1. begin
2.   $\mathcal{M}_f = (\mathcal{A}, Q_f, \Delta_f, S_f, F_f) := \tau(f)$;
3.   while $\exists (s_{i_1}, p_1, s_{j_1}), (s_{i_2}, p_2, s_{j_2}) \in \Delta_f$ that violate either (V1) or (V2) do
4.     $\Delta_1 := \Delta_f - \{(s_{i_1}, p_1, s_{j_1})\}$;
```

```
5.        $\mathcal{M}_1 := (\mathcal{A}, Q_f, \Delta_1, S_f, F_f)$;
6.        $\Delta_2 := \Delta_f - \{(s_{i_2}, p_2, s_{j_2})\}$;
7.        $\mathcal{M}_2 := (\mathcal{A}, Q_f, \Delta_2, S_f, F_f)$;
8.        $\mathcal{M}_f :=$ the union of $\mathcal{M}_1$ and $\mathcal{M}_2$;
9.    end while
10.   return $\mathcal{M}_f$;
11. end.
```

EXAMPLE 5. *It can be verified that for the trail query f of Example* 4 $\mathcal{M}_{f'} = \mathcal{M}_f$, *since both the constraints* (V1) *and* (V2) *are satisfied.*

In order to prove that the regular language, accepted by the finite automaton, $\mathcal{M}_f = \pi(f)$, output from Algorithm 1, is a star-free regular language we first prove the following lemma.

LEMMA 5.14. *If $\mathcal{L}(\mathcal{M})$ is a star-free regular language, then $\mathcal{L}(\mathcal{M}')$ is also a star-free regular language, where $\mathcal{M} = (\mathcal{A}, Q, \Delta, S, F)$ and $\mathcal{M}' = (\mathcal{A}, Q, \Delta - \{(s_i, p, s_j)\}, S, F)$ are finite automata.*

Proof. Let $\mathcal{L} = \mathcal{L}(\mathcal{M})$ and $\mathcal{L}' = \mathcal{L}(\mathcal{M}')$. Also, assume that $(s_i, p, s_j) \in \Delta$; otherwise the result follows trivially. We note that by the construction of $\mathcal{M}'$ we have $\mathcal{L}' \subseteq \mathcal{L}$.

We say that an occurrence of an alphabet symbol p in a substring y of $w \in \mathcal{A}^*$ is *necessary* with respect to (s_i, p, s_j) (or simply $p \in y$ is necessary if w and (s_i, p, s_j) are understood from context) whenever the following statement is satisfied: If w is accepted by $\mathcal{M}$ and p is the ith symbol in w, then (s_i, p, s_j) is the ith transition of *all* successful paths whose label is w.

Since $\mathcal{L}$ is a star-free regular language, then from Theorem 5.5 $\exists n \in \omega(n > 0)$ such that $\forall x, y, z \in \mathcal{A}^*$, (5.1) is satisfied; let us fix n to be such a natural number.

In order to conclude the proof we show that $\exists m(m \geq n)$ such that $\forall x, y, z \in \mathcal{A}^*$,

$$xy^m z \in \mathcal{L}' \text{ iff } xy^{m+1} z \in \mathcal{L}'.$$

Let us fix $m = n + 1$ and $x, y, z \in \mathcal{A}^*$ to be arbitrary strings.

By Theorem 5.5 $xy^m z \in \mathcal{L}$ iff $xy^{m+1} z \in \mathcal{L}$. Furthermore, if $xy^m z \in \mathcal{L}$, then it is also the case that $xy^{m-1} z \in \mathcal{L}$, since $n = m - 1$, and thus the mth occurrence of y in y^m must be the label of a cycle that causes the string $xy^m z$ to be in $\mathcal{L}$.

We have that the following two statements are equivalent:

1. $xy^m z \in \mathcal{L}'$ iff $xy^{m+1} z \in \mathcal{L}'$.
2. The number of necessary occurrences of p in the mth occurrence of the substring y of $xy^m z \in \mathcal{L}$ is zero iff the number of necessary occurrences of p in the $(m+1)$th occurrence of the substring y of $xy^{m+1} z \in \mathcal{L}$ is also zero.

It remains to show that statement (2) above is true.

The *if part* of statement (2) implies that there exists a cycle, say, θ, whose label is y, that causes $xy^{m+1} z$ to be in $\mathcal{L}$ and does not include (s_i, p, s_j). Therefore, θ also causes $xy^m z$ to be in $\mathcal{L}$, since $n = m - 1$, implying that there exists a cycle, whose label is y, that causes $xy^m z$ to be in $\mathcal{L}$. It follows that the number of necessary occurrences of p in the mth occurrence of the substring y of $xy^m z \in \mathcal{L}$ is zero, as required.

Similarly, the *only if part* of statement (2) implies that there exists a cycle, whose label is y, that causes $xy^m z$ to be in $\mathcal{L}$ and does not include (s_i, p, s_j). Therefore, the number of necessary occurrences of p in the $(m+1)$th occurrence of the substring y of $xy^{m+1} z \in \mathcal{L}$ is also zero, as required. $\square$

We proceed to show that $\mathcal{L}(\mathcal{M}_f)$ is a star-free regular language. In particular, the class of languages accepted by finite automata representing trail queries is a proper subclass of the class of star-free regular languages. (Recall that trail formulas do not include the until temporal operator and that negation and disjunction are included only in conditions; see [EMER90, Theorem 6.4].)

THEOREM 5.15. *$\mathcal{L}(\mathcal{M}_f)$ is a star-free regular language.*

Proof. By Lemma 5.12 the regular language $\mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'} = \tau(f)$, is star-free. In addition, by Lemma 5.14 the regular languages accepted by the finite automata, $\mathcal{M}_1$ and $\mathcal{M}_2$, of Algorithm 1 are both star-free. Furthermore, the regular language accepted by $\mathcal{M}_f$, the finite automaton assigned a value at line 8 of the said algorithm, is also star-free, since star-free regular languages are closed under union. Thus, the transformation carried out via the while loop of Algorithm 1, beginning at line 3 and ending at line 9, preserves star-freeness of the regular language accepted by $\mathcal{M}_f$. The result now follows. □

The next theorem follows by a straightforward inspection of Algorithm 1 noting that given the auxiliary set, cond(s_i, p, s_j), $\sum_q |C_q| \leq |f|$, where $C_q \in$ cond(s_i, p, s_j). (See Algorithm 2 in section 6, whose polynomial-time complexity is $O(|Q_f| \cdot |\Delta_f|)$, which can be used to decide whether there exists a path from (s_{i_1}, p_1, s_{j_1}) to (s_{i_2}, p_2, s_{j_2}), where $\pi(f) = \mathcal{M}_f = (\mathcal{A}, Q_f, \Delta_f, S_f, F_f)$.)

THEOREM 5.16. *The finite automaton, $\mathcal{M}_f = \pi(f)$, returned by Algorithm 1, can be constructed in polynomial time in $\#I, |f|$, and $|Q_{f'}|$, where $\tau(f) = (\mathcal{A}, Q_{f'}, \Delta_{f'}, S_{f'}, F_{f'})$.*

The following theorem shows that the language induced by the output of a trail query f is equal to the regular language accepted by the intersection of the finite automaton representing f and the finite automaton representing the Hypertext database H, which constitutes the input to f.

THEOREM 5.17. *$\mathcal{L}(f(H)) = \mathcal{L}(\mathcal{M}_H \cap \mathcal{M}_f)$, where H is a Hypertext database and f is a trail query.*

Proof. ($\mathcal{L}(f(H)) \subseteq \mathcal{L}(\mathcal{M}_H \cap \mathcal{M}_f)$): By Lemma 5.8 $\mathcal{L}(f(H)) \subseteq \mathcal{L}(\mathcal{M}_H)$. Thus, we need to show that it is also the case that $\mathcal{L}(f(H)) \subseteq \mathcal{L}(\mathcal{M}_f)$.

Let $\rho(T) \in \mathcal{L}(f(H))$ implying by Definition 4.14 that $T \models f$. We note that the constraint corresponding to (V1), i.e., that no two unique variables, which are distinct, can be mapped by an interpretation to the same $T[i]$, and the constraint corresponding to (V2), i.e., that an interpretation maps the same variable to the same $i \in \{1, \ldots, \#I\}$ are both satisfied by Definition 4.3 of an interpretation σ.

We show that $\rho(T) \in \mathcal{L}(\mathcal{M}_f)$ by induction on the length of f. Recall that we have assumed that all conditions in f have the same length of one.

(*Basis*): If $|f| = 1$, then f is a condition, say, C. The result follows by Definition 4.8 part (T1) and the description of τ in Definition 5.11 part (1), since there is a transition in $\mathcal{M}_f$ for each page number i_j with $\sigma(Z) = i_j$.

(*Induction*): Assume that the result holds when $|f| = k$, where $k \geq 1$; we then need to prove that the result holds when $|f| > k$.

We consider the four cases according to the structure of f as follows:

1. If f is the query $g_1 \wedge g_2$, then by Definition 4.8 part (T2) $T \models f$ iff $T \models g_1$ and $T \models g_2$. Furthermore, by the inductive hypothesis, $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_1})$ and $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_2})$. Therefore, by the description of τ in Definition 5.11 part (2), $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the intersection of the intermediate automata $\mathcal{M}_{g'_1}$ and $\mathcal{M}_{g'_2}$. The result follows from Algorithm 1, which implies that $\mathcal{M}_f$ is equal to the intersection of $\mathcal{M}_{g_1}$ and $\mathcal{M}_{g_2}$.

2. If f is the query $\Diamond(g)$, then by Definition 4.8 part (T3) $T \models f$ iff $\exists i \in \{0, \ldots, \#T - 1\}$ such that $T^i \models g$. Furthermore, by the inductive hypothesis, $\rho(T^i) \in \mathcal{L}(\mathcal{M}_g)$. Therefore, by the description of τ in Definition 5.11 part (3), $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\gamma$ and the intermediate automaton $\mathcal{M}_{g'}$. The result follows from Algorithm 1, which implies that $\mathcal{M}_f$ is equal to the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_g$.
3. If f is the query $\bigcirc(g)$, then by Definition 4.8 part (T4) $T \models f$ iff $\# T > 1$ and $T^1 \models g$. Furthermore, by the inductive hypothesis, $\rho(T^1) \in \mathcal{L}(\mathcal{M}_g)$. Therefore, by the description of τ in Definition 5.11 part (4), $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\bigcirc$ and the intermediate automaton $\mathcal{M}_{g'}$. The result follows from Algorithm 1, which implies that $\mathcal{M}_f$ is equal to the concatenation of $\mathcal{M}_\bigcirc$ and $\mathcal{M}_g$.
4. If f is the query $\triangle(g)$, then by Definition 4.8 part (T5) $T \models f$ iff $\# T > 0$ and $T^i \models g$, where $i = \#T - 1$. Furthermore, by the inductive hypothesis, $\rho(T^i) \in \mathcal{L}(\mathcal{M}_g)$. Therefore, by the description of τ in Definition 5.11 part (5), $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\gamma$ and the intermediate automaton $\mathcal{M}_{g'}$. Moreover, when contructing $\mathcal{M}_{g'}$ $\triangle$flag will have been set to *true* implying that $\rho(T^i) \in \mathcal{L}(\mathcal{M}_{g'})$, where $i = \#T - 1$. The result follows from Algorithm 1, which implies that $\mathcal{M}_f$ is equal to the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_g$.

($\mathcal{L}(\mathcal{M}_H \cap \mathcal{M}_f) \subseteq \mathcal{L}(f(H))$): Let $w \in \mathcal{L}(\mathcal{M}_H \cap \mathcal{M}_f)$, where $w = \delta(s_1)\delta(s_2)\ldots\delta(s_n)$ (see Definition 5.2). Thus $w = \rho(T)$ for some trail T in R, with $\#T = n$, since $w \in \mathcal{L}(\mathcal{M}_H)$. Furthermore, there exists an interpretation σ over H such that $\forall i \in \{1, \ldots, n\}$ $\forall Z \in \varphi(\text{cond}(s_i, \delta(s_i), s_{i+1})), \sigma(Z) = \delta(s_i)$, since by Algorithm 1 both conditions, (V1) and (V2), are satisfied by the path whose label is w.

We now show that $T \models f$ with respect to H and σ, implying that $w \in \mathcal{L}(f(H))$ as required, by induction on the length of f. As before, recall that we have assumed that all conditions have the same length of one.

(*Basis*): If $|f| = 1$, then f is a condition, say, C. The result follows as in the previous basis step.

(*Induction*): Assume that the result holds when $|f| = k$, where $k \geq 1$; we then need to prove that the result holds when $|f| > k$.

We consider the four cases according to the structure of f as follows:

1. If f is the query $g_1 \wedge g_2$, then by the description of τ in Definition 5.11 part (2) $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_1'})$ and $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_2'})$. Moreover, by Algorithm 1, $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_1})$ and $\rho(T) \in \mathcal{L}(\mathcal{M}_{g_2})$. Furthermore, by the inductive hypothesis, $T \models g_1$ and $T \models g_2$. The result then follows by Definition 4.8 part (T2).
2. If f is the query $\Diamond(g)$, then by the description of τ in Definition 5.11 part (3) $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_{g'}$. Moreover, by Algorithm 1, $\exists i \in \{0, \ldots, \#T - 1\}$ such that $\rho(T^i) \in \mathcal{L}(\mathcal{M}_g)$. Furthermore, by the inductive hypothesis, $T^i \models f$. The result then follows by Definition 4.8 part (T3).
3. If f is the query $\bigcirc(g)$, then by the description of τ in Definition 5.11 part (4) $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\bigcirc$ and $\mathcal{M}_{g'}$. Moreover, by Algorithm 1, $\#T > 1$ and $\rho(T^1) \in \mathcal{L}(\mathcal{M}_g)$. Furthermore, by the inductive hypothesis, $T^1 \models f$. The result then follows by Definition 4.8 part (T4).
4. If f is the query $\triangle(g)$, then by the description of τ in Definition 5.11 part (5) $\rho(T) \in \mathcal{L}(\mathcal{M}_{f'})$, where $\mathcal{M}_{f'}$ is the concatenation of $\mathcal{M}_\gamma$ and $\mathcal{M}_{g'}$. Moreover, when contructing $\mathcal{M}_{g'}$ $\triangle$flag will have been set to *true* implying that $\rho(T^i) \in$

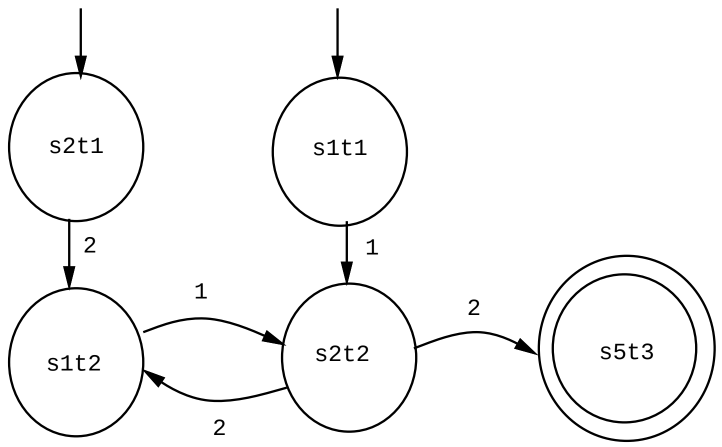


FIG. 5.3. *A finite automaton for the output of a trail query.*

$\mathcal{L}(\mathcal{M}_{g'})$, where $i = \#T - 1$. By Algorithm 1 $\# \ T > 0$ and $\rho(T^i) \in \mathcal{L}(\mathcal{M}_g)$, where $i = \#T - 1$. Furthermore, by the inductive hypothesis, $T^i \models f$. The result then follows by Definition 4.8 part (T5). □

EXAMPLE 6. *The finite automaton $\mathcal{M}_H \cap \mathcal{M}_f$, where $\mathcal{M}_H$ is shown in Figure* 5.1 *and $\mathcal{M}_f$ is shown in Figure* 5.2*, which accepts the same language that is induced by $f(H)$, is shown in Figure* 5.3*.*

In order to test whether *f(H)* is nonempty, by Theorem 5.17 we can check whether $\mathcal{L}(\mathcal{M}_H \cap \mathcal{M}_f)$ is nonempty, which can be done in nondeterministic logspace [JONE75] and thus in polynomial time in the size of $\mathcal{M}_H \cap \mathcal{M}_f$.

The following corollary is an immediate consequence of Theorems 5.6, 5.15, and 5.17, since the class of regular star-free languages is closed under intersection.

COROLLARY 5.18. *$\mathcal{L}(f(H))$ is a star-free regular language.*

We note that by Theorem 6.4 in [EMER90] PLTL is exactly as expressive as the class of star-free languages which do not include ϵ. Corollary 5.18, in view of Theorem 5.17, implies that $\mathcal{L}(f(H))$ is actually a member of a proper subclass of the class of star-free regular languages corresponding to the intersection of the star-free regular languages accepted by the finite automata representing trail queries and Hypertext databases, respectively.

The following corollary concerning the complexity of constructing $\mathcal{M}_H$, $\mathcal{M}_f$ and the intersection thereof is an immediate consequence of Definition 5.1, Lemma 5.13, and Theorem 5.16.

COROLLARY 5.19. *The following statements are true:*

1. *$\mathcal{M}_H$ can be constructed in linear time in $\#I$ and $|R|$.*
2. *$\mathcal{M}_f$ can be constructed in time exponential in the number of conjunctions, between subformulas of f, plus one; if there are no conjunctions in f, then $\mathcal{M}_f$ can be constructed in polynomial time in $\#I$ and $|f|$.*
3. *$\mathcal{M}_H \cap \mathcal{M}_f$ can be constructed in polynomial time in $|Q_H|$ and $|Q_f|$, where Q_H and Q_f are the state sets of $\mathcal{M}_H$ and $\mathcal{M}_f$, respectively.*

It is interesting to compare Corollary 5.19 with a result of [LICH84] showing that although model checking of PLTL formulas (see Definition 6.2) is exponential time in the length of the formulas it is linear time in the size of the temporal structure being checked.

A special case worth considering is when R is acyclic. In this case the number of trails in R is finite and thus the regular language defined by the output of a query is a finite regular language and therefore trivially star-free. As we will see in the next section, the fact that R is acyclic does not necessarily reduce the complexity of computing the star-free regular language induced by $f(H)$, namely, $\mathcal{L}(f(H))$.

Finally, it would be useful to extend HQL to enable the specification of second-order notions such as specifying that only the shortest trails (i.e., the trails with the least count) would be included in $f(H)$. Obviously for such an extension further analysis of $\mathcal{M}_H \cap \mathcal{M}_f$ would have to be carried out.

6. The complexity of navigation in Hypertext.

In this section we investigate the complexity of deciding whether a Hypertext database is a model of a trail formula. Our results show that, in general, this problem cannot be solved efficiently (assuming P $\neq$ NP). On the positive side we exhibit two significant subclasses of $\Diamond$-free trail formulas for which this decision problem can be solved in polynomial time.

Our results imply that the navigation problem is not easily solved in the general case. In practice Hypertext systems could include algorithms which return randomized and/or fuzzy solutions.

The following lemma shows that if a Hypertext database is a model of a given trail formula f, then we can find a trail, which satisfies f, and whose count is no greater than $\#I$ multiplied by $|Sub(f)|$.

LEMMA 6.1. *Let a Hypertext database, $H = \langle I, R\rangle$, be a model of the trail formula, f. Then $\exists T$ in R such that $T \models f$ and $\#T \leq \#I \cdot |Sub(f)|$.*

Proof. Let L in R be a trail such that $L \models f$ holds; such a trail exists in R, since H is a model of f. We prove the result by showing that there exists a subtrail (see Definition 3.6), T of L, such that $T \models f$ and $\#T \leq \#I \cdot |Sub(f)|$; the proof is by induction on $|Sub(f)|$.

(*Basis*): If $|Sub(f)| = 1$, then the result follows from Definition 4.8 part (T1), since $\#L > 0$ and the prefix L_1 satisfies $L_1 \models f$ and $\#L_1 = 1$.

(*Induction*): Assume that the result holds when $|Sub(f)| = k$, where $k \geq 1$; we then need to prove that the result holds when $|Sub(f)| = k + 1$.

We conclude the result by considering the form of f according to the four cases, T2 to T5, of Definition 4.8 as follows (recalling the definition of the concatenation of two arrays given in Definition 3.3):

1. If f is of the form $f_1 \wedge f_2$, then $L \models f_1$ and $L \models f_2$. Furthermore, by the inductive hypothesis $T \models f_1$ and $T \models f_2$, where T is a subtrail of L, and $\# T \leq \#I \cdot \max(|Sub(f_1)|, |Sub(f_2)|)$. The result follows, since $\max(|Sub(f_1)|, |Sub(f_2)|) \leq |Sub(f)|$.
2. If f is of the form $\Diamond(f')$, then $\exists i \in \{0, \ldots, \#L - 1\}$ such that $L^i \models f'$. Furthermore, by the inductive hypothesis there exists a subtrail T of L^i, with $\#T \leq \#I \cdot |Sub(f')|$, such that $T \models f'$. The result follows by Proposition 4.11 part (8) which implies that $T \models \Diamond(f')$, since $|Sub(f')| \leq |Sub(f)|$.
3. If f is of the form $\bigcirc(f')$, then $\#L > 1$ and $L^1 \models f'$. Furthermore, by the inductive hypothesis there exists a subtrail T of L^1 such that $T \models f'$, with $\#T \leq \#I \cdot |Sub(f')|$. Let L_j be the prefix of L such that $L_j T$ is also a prefix of L. It follows that there exists a subtrail T' of L in R such that $\#T' = 1$ and $T'[1] = L[j]$, since $\#L_j \geq 1$. The result follows, since by Definition 4.8 part (T4) $T'T \models f$ and $\#(T'T) = \#T + 1$.

4. If f is of the form $\triangle(f')$, then $\#L > 0$ and $L^i \models f'$, where $i = \#L - 1$. The result follows, since by Definition 4.8 part (T5) $L^i \models \triangle(f')$ and $\#L^i = 1$. $\square$

DEFINITION 6.2 (model checking). *The* model checking *problem is the problem of deciding whether a Hypertext database is a model of a trail formula.*

We proceed to investigate the complexity of the model checking problem.

LEMMA 6.3. *The model checking problem is in NP.*

Proof. In order to show that the model checking problem is in NP we present a nondeterministic polynomial-time algorithm that decides the problem (cf. [SIST85, Theorem 3.5]).

Let $H = \langle I, R\rangle$ be a Hypertext database and f be a trail formula. We first guess a trail T in R and an interpretation σ over H. It can easily be checked in polynomial time in $|f|$ that the restriction of σ to the unique variables in the subformulas of f is a one-to-one mapping. Furthermore, by Lemma 6.1 we can assume without loss of generality that $\#\, T \leq \#I \cdot |Sub(f)|$.

We now present an algorithm, which verifies whether $T \models f$ or not. The algorithm maintains a set, called *label*(i), for each marker i of T, which is initialized to the empty set. It then considers all of the subformulas $g \in Sub(f)$ in increasing order of $|g|$, assuming as before that all conditions in g have the same length of one, as follows:

1. If g is the condition C, with variable Z, then add g to label(i) iff $\sigma \models C$ and $T[i] = \sigma(Z)$.
2. If g is the formula $g_1 \wedge g_2$, then add g to label(i) iff $g_1, g_2 \in$ label(i).
3. If g is the formula $\diamond(g')$, then add g to label(i) iff $\exists j \in \{i, \ldots, \#T\}$ such that $g' \in$ label(j).
4. If g is the formula $\bigcirc(g')$, then add g to label(i) iff $g' \in$ label$(i+1)$.
5. If g is the formula $\triangle(g')$, then add g to label (i) iff $g' \in$ *label*$(\#T)$.

At the end of the above algorithm, whose time complexity is a polynomial in $\#I$ and $|f|$, it can be verified that $T \models f$ iff $f \in$ label(1). $\square$

THEOREM 6.4. *The model checking problem is NP-complete.*

Proof. By Lemma 6.3 model checking is in NP. It remains to show that the problem is NP-hard.

In order to show NP-hardness we reduce, in polynomial time, 3SAT [GARE79] to the model checking problem using a reduction similar to that used in [SIST85, Theorem 3.5].

Let $J = J_1 \wedge \ldots \wedge J_m$ be a conjunction of m clauses on a finite set of variables $\{x_1, \ldots, x_n\}$ such that $\forall i \in \{1, \ldots, m\}$, $J_i = L_{i1} \vee L_{i2} \vee L_{i3}$ is a clause of three literals, where each literal L_{ij} is either x_q or $\neg x_q$, with $q \in \{1, \ldots, n\}$.

We define an operator, denoted by ϕ, which returns the clauses that contain a literal x_i or $\neg x_i$ as follows:

- $\phi(x_i) = \{J_j | x_i \text{ is a literal in } J_j\}$ and
- $\phi(\neg x_i) = \{J_j | \neg x_i \text{ is a literal in } J_j\}$.

Let $H(J) = \langle I(J), R(J)\rangle$ be the Hypertext database defined by the following:

1. The set of strings $\{a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_m, y_0, y_1, \ldots, y_n\} \subseteq \Sigma^*$, where the a_i's, b_i's, c_i's, and y_i's are distinct, and $A \in \mathcal{U}$.
2. $\#I(J) = 3n + 1$.
3. $\forall i \in \{1, \ldots, n\}, I(J)[i] = (A, w_i)$, with $w_i = a_i c_1 \ldots c_k, k \leq m$, and where $\phi(x_i) = \{J_1, \ldots, J_k\}$.
4. $\forall i \in \{n+1, \ldots, 2n\}, I(J)[i] = (A, w_i)$, with $w_i = b_{i-n} c_1 \ldots c_k, k \leq m$, and where $\phi(\neg x_{i-n}) = \{J_1, \ldots, J_k\}$.
5. $\forall i \in \{2n+1, \ldots, 3n+1\}, I(J)[i] = (A, y_{i-(2n+1)})$.

6. $R(J) = \{(\psi(y_{i-1}), \psi(w_i)), (\psi(w_i), \psi(y_i)), (\psi(y_{i-1}), \psi(w_{i+n})), (\psi(w_{i+n}), \psi(y_i)) \mid i \in \{1, \ldots, n\}\}$, where $\forall j \in \{1, \ldots, \#I(J)\}, \psi(I(J)[j])$ denotes the page number j of $I(J), \psi(w_i)$ stands for $\psi((A, w_i))$, and $\psi(y_i)$ stands for $\psi((A, y_i))$.

The structure of $R(J)$ is shown in Figure 6.1, where for notational convenience the page numbers $\psi((A, a_i c_1 \ldots c_k))$, $\psi((A, b_{i-n} c_1 \ldots c_k))$, and $\psi((A, y_{i-(2n+1)}))$ are abbreviated to a_i, b_i, and y_i, respectively.

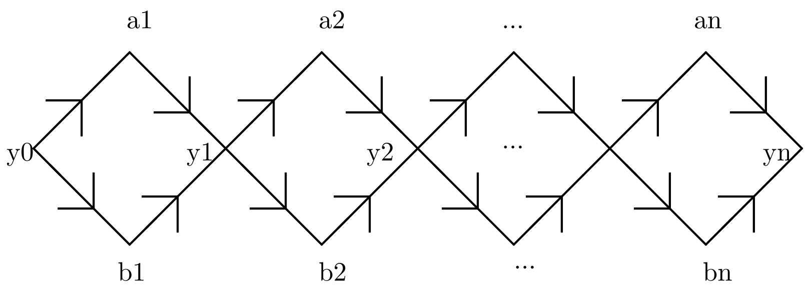


FIG. 6.1. *The reachability relation R(J).*

It can now be verified that J is satisfiable iff $H(J)$ is a model of the trail formula,

$$substr(X_0, y_0) \wedge \Diamond(substr(X_1, c_1)) \wedge \ldots \wedge \Diamond(substr(X_m, c_m)) \wedge \triangle(substr(X_{m+1}, y_n)),$$

where $X_0, X_1, \ldots, X_m, X_{m+1}$ are $m + 2$ distinct normal variables. □

The following result is an immediate consequence of the reduction in the proof of Theorem 6.4.

COROLLARY 6.5. *The model checking problem is NP-complete for the class of Hypertext databases having acyclic reachability relations and the class of* $\bigcirc$*-free trail formulas whose conditions do not contain any unique variables.*

The next result shows that the model checking problem does not become tractable by allowing unique variables in conditions and disallowing normal variables.

THEOREM 6.6. *The model checking problem is NP-complete for the class of Hypertext databases having acyclic reachability relations and the class of* $\bigcirc$*-free trail formulas whose conditions do not contain any normal variables.*

Proof. As in Theorem 6.4, by Lemma 6.3 the model checking problem is in NP. It remains to show that the problem is NP-hard. In order to conclude the result we modify the reduction of 3SAT given in the proof of Theorem 6.4. First, we modify parts 3 and 4 of the construction of $H(J)$ as follows:

3. $\forall i \in \{1, \ldots, n\}, I(J)[i] = (A, w_i)$, with $w_i = a_i$; these pages represent the literals x_i.

4. $\forall i \in \{n + 1, \ldots, 2n\}, I(J)[i] = (A, w_i)$, with $w_i = b_{i-n}$; these pages represent the literals $\neg x_i$.

Second, for each $x_q, q \in \{1, \ldots, n\}$, such that $\phi(x_q) = \{J_1, \ldots, J_k\}$ we add k additional pages to $I(J)$. Each such additional page $I(J)[i]$ is of the form $(A, a_i c_j), j \in \{1, \ldots, k\}$, where c_j corresponds to the clause J_j. Similarly, for each $\neg x_q, q \in \{1, \ldots, n\}$, such that $\phi(\neg x_q) = \{J_1, \ldots, J_k\}$ we add another k additional pages to $I(J)$. As in the preceding case each such additional page $I(J)[i]$ is of the form $(A, b_{i-n} c_j), j \in \{1, \ldots, k\}$, where c_j corresponds to the clause J_j.

As in the previous theorem the page numbers $\psi((A, a_i)), \psi((A, b_{i-n}))$, and $\psi((A, y_{i-(2n+1)}))$ are abbreviated for notational convenience to a_i, b_i, and y_i, respectively, with $i \in \{1, \ldots, n\}$. In addition, both the page numbers $\psi((A, a_i c_j))$ and

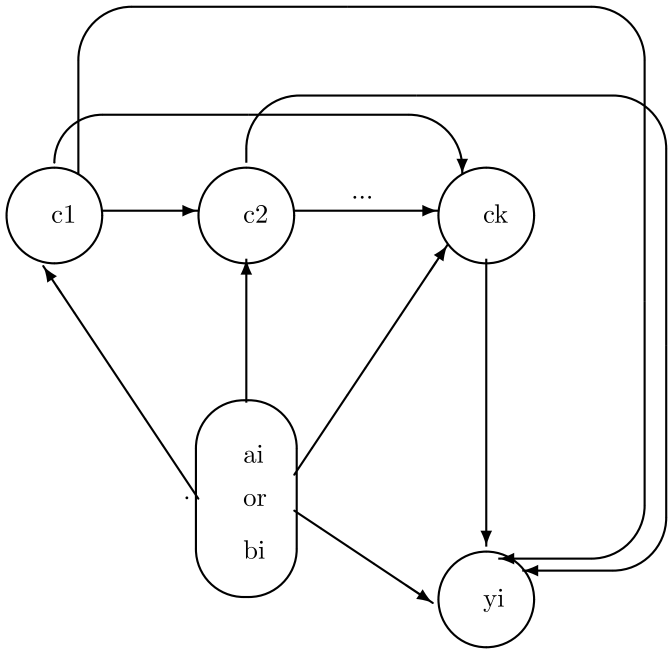


FIG. 6.2. *The subgraph added to R(J).*

$\psi((A, b_{i-n}c_j))$ are abbreviated to c_j, with $j \in \{1, \ldots, k\}$, where the distinction between a_ic_j and $b_{i-n}c_j$ is understood from context.

Moreover, for each page number i of $I(J)$ representing x_i, we add a subgraph to the reachability relation $R(J)$ of Theorem 6.4 (see Figure 6.1) as follows. First, we add an arc from the page number i of $I(J)$ to each of the page numbers c_j. Second, we add an arc from each of the page numbers c_j, $j \in \{1, \ldots, k\}$, to each of the page numbers c_p, with $j < p$ and $p \in \{1, \ldots, k\}$. Finally, we add an arc from each of the page numbers c_j to the page number y_i. We repeat this process for each page number i of $I(J)$ representing $\neg x_i$.

The subgraph added to the reachability relation $R(J)$ of Theorem 6.4 is shown in Figure 6.2.

It can now be verified that J is satisfiable iff $H(J)$ is a model of the trail formula,

$$\begin{aligned} &substr(Y_0, y_0) \wedge \Diamond(substr(Y_1, c_1)) \wedge \ldots \\ &\ldots \wedge \Diamond(substr(Y_m, c_m)) \wedge \triangle(substr(Y_{m+1}, y_n)), \end{aligned} \tag{6.1}$$

where $Y_0, Y_1, \ldots, Y_m, Y_{m+1}$ are $m+2$ distinct unique variables.

Specifically, if $H(J)$ is a model of (6.1), then for some trail $T \in R(J), T$ satisfies (6.1) with respect to some interpretation σ over H. Now, for each $\sigma(Y_i), i \in \{1, \ldots, m\}$, let a_i or b_i be the page number such that either $(a_i, \sigma(Y_i))$ or $(b_i, \sigma(Y_i))$ is in $R(J)$. It follows therefore that J is satisfiable under an assignment where each literal corresponding to a_i or b_i is made true.

Correspondingly, if J is satisfiable then we need to exhibit a trail T that satisfies (6.1) with respect to some interpretation σ over H. Now, since J is satisfiable there is an assignment that makes each J_j in J, $j \in \{1, \ldots, m\}$, true. Furthermore, J_j is true due to a literal, say, L_i, being true in J_j. Let either a_i or b_i be the page number corresponding to L_i and let c_j be the page number corresponding to J_j such that either (a_i, c_j) or (b_i, c_j) is in $R(J)$. Let σ be such that $\sigma(Y_0) = y_0$, $\sigma_{m+1}(Y_{m+1}) = y_n$ and for $j \in \{1, \ldots, m\}$ $\sigma(Y_j) = c_j$. The result now follows, since there is a trail passing through all the c_j's satisfying the desired trail formula. $\square$

We next show that the model checking problem is also NP-complete for the subclass of $\diamond$-free trail formulas.

THEOREM 6.7. *The model checking problem is NP-complete for the class of $\diamond$-free trail formulas.*

Proof. By Lemma 6.3 the model checking problem is in NP. It remains to show that the problem is NP-hard. In order to show NP-hardness we reduce, in polynomial time, the longest path problem, which is known to be NP-complete [GARE79], to the model checking problem for the class of $\diamond$-free trail formulas.

Let $G = (V, E)$ be a directed graph with $V = \{v_1, \ldots, v_m\}$ and $s, t \in V$. We need to solve the problem: does there exist a *simple path* (that is, a path in which no node occurs more than once) from s to t in G of length k or more?

We first let ϕ be a one-to-one mapping from V to $\{A\} \times \Sigma^*$, where $A \in \mathcal{U}$.

Next let $H(G) = \langle I(G), R(G) \rangle$ be a Hypertext database defined by

1. $\#I(G) = |V| = m$.
2. $\forall i \in \{1, \ldots, m\}, \phi(v_i) = I(G)[i]$.
3. $\psi(G) = R(G)$, where ψ is an isomorphism from G to $R(G)$ such that $\psi((v_i, v_j)) = (i, j)$, where i is the page number satisfying $\phi(v_i) = I(G)[i]$ and j is the page number satisfying $\phi(v_j) = I(G)[j]$.

It can now be verified that there exists a simple path from s to t in G of length k or more iff $H(G)$ is a model of the trail formula

$$\begin{aligned} &substr(Y_0, \beta(\phi(s))) \wedge \bigcirc(att(Y_1, A)) \wedge \bigcirc^2(att(Y_2, A)) \wedge \\ &\bigcirc^{k-1}(att(Y_{k-1}, A)) \wedge \triangle(substr(Y_k, \beta(\phi(t)))), \end{aligned} \tag{6.2}$$

where $Y_0, Y_1, \ldots, Y_k$ are distinct unique variables and $\bigcirc^p$ denotes the composition, $\bigcirc \ldots \bigcirc, p$ times, with $p \in \omega$. $\square$

The following two results follow from the reduction in the proof of Theorem 6.7.

COROLLARY 6.8. *The model checking problem is NP-complete for the class of $\diamond$-free trail formulas whose conditions do not contain any normal variables.*

The next corollary is easily shown by replacing all occurrences of $\bigcirc$ in (6.2) by $\diamond$; it also follows from Theorem 6.6.

COROLLARY 6.9. *The model checking problem is NP-complete for the class of $\bigcirc$-free trail formulas whose conditions do not contain any normal variables.*

We now show that model checking can be done in polynomial time for $\diamond$-free trail formulas when the reachability relation of the Hypertext database is acyclic. In order to prove the result we first solve the following graph-theoretic problem: Given a directed acyclic graph (V, E) and a family $\{V_i\}$ $(i \in \{0, \ldots, k\})$ of subsets of V, does there exist a path, $\langle v_0, \ldots, v_k \rangle$, of length k in G such that $v_i \in V_i$?

Let us call this the *path of length k* problem. We note that when we consider directed acyclic graphs every path is actually a simple path.

We next show that a solution to the path of length k problem can be obtained in $O(k \cdot |E|)$ time; when (V, E) is acyclic then $k \leq |V|$.

LEMMA 6.10. *Given a directed graph (V, E) and a family $\{V_i\}$ $(i \in \{0, \ldots, k\})$ of subsets of V, then the path of length k problem can be solved in $O(k \cdot |E|)$ time.*

Proof. In order to solve the problem we give the pseudocode of an algorithm designated PATH$((V, E), \{V_i\})$, which takes as input a directed graph and a family of $k + 1$ subsets of V and returns YES if there is a solution to the path of length k problem; otherwise it returns NO.

For each subset $V_i \subseteq V$ we maintain an auxiliary set, called P_i, which stores all the nodes that can be reached from a node in V_0 via a path of length i. We also assume that adjacent(v) denotes the set of nodes: $\{u|(v, u) \in E\}$.

Algorithm 2 (PATH((V, E), $\{V_i\}$)).

```
1.  begin
2.    $P_0 := V_0$;
3.    for $i = 1$ to $k$ do
4.      $P_i := \emptyset$;
5.      for all $v \in P_{i-1}$ do
6.        for all $u \in adjacent(v)$ do
7.          if $u \in V_i$ then
8.            $P_i := P_i \cup \{u\}$;
9.          end if
10.       end for
11.     end for
12.   end for
13.   if $P_k \neq \emptyset$ then
14.     return YES;
15.   else
16.     return NO;
17.   end if
18. end.
```

It can be verified that the algorithm is correct and solves the path of length k problem in $O(k \cdot |E|)$ time. $\square$

We next present a special case when model checking can be done in polynomial time.

Theorem 6.11. *The model checking problem can be solved in polynomial time in the length of the repository and the length of the trail formula for the class of Hypertext databases having acyclic reachability relations and for the class of $\diamond$-free trail formulas.*

Proof. Let $H = \langle I, R\rangle$ be a Hypertext database with R being acyclic and let f be a $\diamond$-free trail formula. Since f is $\diamond$-free it follows that we can use Proposition 4.11 in order to obtain a trail formula which is equivalent to f and is in the following *normal form*:

$$C_{1_1} \wedge \ldots \wedge C_{n_1} \wedge \triangle(C_{1_2}) \wedge \ldots \wedge \triangle(C_{n_2}) \wedge \bigcirc \ldots \bigcirc (C_{1_3}) \wedge \ldots \wedge \\ \bigcirc \ldots \bigcirc (C_{n_3}) \wedge \bigcirc \ldots \bigcirc \triangle(C_{1_4}) \wedge \ldots \wedge \bigcirc \ldots \bigcirc \triangle(C_{n_4}),$$

where each C_i is a condition which is also a subformula of f, noting that for any trail T in R it cannot be the case that $T \models \triangle \bigcirc (f)$.

On using Proposition 4.11 it can be verified that if a trail formula, f, is $\diamond$-free, then f can be converted into a normal form $\diamond$-free trail formula in time polynomial in $|f|$. For the rest of the proof we assume that f is in normal form.

Let $\bigcirc^k[f]$ denote the set of conditions in the conjuncts of f having no instance of $\triangle$ and exactly k instances of $\bigcirc$, where $k \geq 0$, and let max$\bigcirc(f)$ denote the maximum number of instances of $\bigcirc$ in any conjunct of f. In addition, let $\triangle[f]$ denote the set of conditions, $\{C_{1_2}, \ldots, C_{n_2}, C_{1_4}, \ldots, C_{n_4}\}$, in conjuncts of f having an instance of $\triangle$.

Finally, let F be an array of sets of page numbers such that $\#F = \max\bigcirc(f) + 2$ and such that initially $\forall i \in \{1, \ldots, \#F\}, F[i] = \emptyset$.

The first step in our model checking algorithm is syntactic. If one of the following constraints is violated, then H is not a model of f, since a contradiction arises:

1. There do not exist two unique variables, which are distinct, contained in the set of conditions $\bigcirc^k[f]$ for any $k \geq 0$; i.e., no two unique variables, which are distinct, can appear at the *same time*.
2. There do not exist two unique variables, which are distinct, contained in the set of conditions $\triangle[f]$; i.e., no two unique variables, which are distinct, can appear at the *final time*.
3. There do not exist natural numbers, k_1, k_2, with $k_1 \neq k_2$ and $0 \leq k_1, k_2 \leq \max\bigcirc(f)$, such that the intersection of the set of variables contained in $\bigcirc^{k_1}[f]$ with the set of variables contained in $\bigcirc^{k_2}[f]$ is nonempty; i.e., no variable can appear in two distinct *times*.
4. $\forall k, 0 \leq k < \max\bigcirc(f)$, the intersection of the set of variables contained in $\bigcirc^k[f]$ with the set of variables contained in $\triangle[f]$ is empty; i.e., no variable can appear before and at the *final time*.

It can be verified that all of the above constraints can be checked in polynomial time in $|f|$. Assuming that the above constraints are satisfied, then the following statements are true:

1. $\forall k, 0 \leq k \leq \max\bigcirc(f)$, if $\bigcirc^k[f]$ is nonempty, then we can assume that there exists only one variable in its conjuncts such that this variable does not appear in any other set $\bigcirc^q[f]$, where $0 \leq k \neq q \leq \max\bigcirc(f)$; i.e., there is only one variable per *time unit*.
2. If $\triangle[f]$ is nonempty, then we can assume that there is only one variable in its conjuncts, such that this variable does not appear in any other set $\bigcirc^k[f]$, where $0 \leq k < \max\bigcirc(f)$; i.e., there is only one variable for the *final time unit*.
3. If the intersection of the set of variables contained in $\triangle[f]$ and the set of variables contained in $\bigcirc^k[f]$ is nonempty, where $k = \max\bigcirc(f)$, then the single variable assumed to be in their conjuncts must be the same variable; i.e., in this case the *final time* coincides with the time implied by $\max\bigcirc(f)$. Otherwise, we assume that their single respective variables are distinct.

We note that statements (1) and (2) above are valid, since if H is a model of f and T is a trail in R such that $T \models f$ with respect to some interpretation σ over H, then for the variables X_i in the conjunctions of $\bigcirc^k[f]$ or $\triangle[f]$, σ must map X_i to the same page number.

From now on we assume that the above statements are enforced in normal form trail formulas by a straightforward renaming of variables. We are not concerned whether the single variable per time unit is normal or unique due to the fact that R is acyclic and thus all the paths in the directed acyclic graph corresponding to R are simple.

For each page number, *pn* of I, we maintain an auxiliary set, called cond(*pn*), which is initialized to the empty set. We then consider all the conditions C, of all the conjuncts of f, in turn and add C to cond(*pn*) according to the following constraint:

- Add C to cond(*pn*) iff $\sigma \models C$, assuming that σ is an interpretation over H and $\sigma(Z) = pn$, where Z is the variable of C.

Constructing the sets cond(*pn*) can be done in polynomial time in $\#I$ and $|f|$.

Next, for each page number *pn* of I add *pn* to F according to the following two rules:

1. If $(\bigcirc^k[f] \neq \emptyset) \subseteq \text{cond}(pn)$, where $0 \leq k \leq \max\bigcirc(f)$, then add pn to $F[k+1]$.
2. If $(\triangle[f] \neq \emptyset) \subseteq \text{cond}(pn)$, then add pn to $F[\#F]$.

In the next step of our model checking algorithm we check if one of the ensuing constraints is violated in which case H is not a model of *f*.

1. For any $k, 0 \leq k \leq \max\bigcirc(f)$, if $\bigcirc^k[f] \neq \emptyset$, then $F[k+1] \neq \emptyset$.
2. If $\triangle[f] \neq \emptyset$, then $F[\#F] \neq \emptyset$.

The above constraints can be tested in polynomial time in $\#I$ and $|f|$. We therefore assume that the above two constraints are satisfied.

In order to utilize Lemma 6.10, we modify F by setting $F[i]$ to $\{1, \ldots, \#I\} \forall i \in \{1, \ldots, \#F\}$ whenever $F[i] = \emptyset$. The result now follows, on viewing R as a directed acyclic graph, and viewing the prefix F_{k+1} of F, where $k+1 = \#F - 1$, as a family of $k+1$ subsets of I. The model checking problem thus reduces to the path of length k problem, with the following final test: $P_k^* \cap F[\#F] \neq \emptyset$, where ${P_k}^*$ denotes the set of nodes, $P_k \cup \{u | v \in P_k$ and (v, u) is an arc in the transitive closure of $R\}$. □

The following corollary, which is an immediate consequence of Theorem 6.11, demonstrates a special case when model checking can be solved in polynomial time even when the reachability relation is cyclic.

Corollary 6.12. *The model checking problem can be solved in polynomial time in the length of the repository and the length of the trail formula for the class of $\diamond$-free trail formulas whose conditions do not contain any unique variables and such that the normal variables appearing in conditions at distinct times are distinct.*

Proof. The result follows, since in this special case constraints (1) to (4) in the proof of Theorem 6.11 do not have to be satisfied. Furthermore, in this special case, Algorithm 2 will return YES iff there exists a trail in R, which is not necessarily loop-free; i.e., a node may appear more than once in the path corresponding to the trail. □

The following corollary, which is given for completeness, follows immediately from Theorem 6.11 on using Algorithm 2, since we need only inspect $O(\#I^2)$ pages.

Corollary 6.13. *The model checking problem can be solved in polynomial time in the length of the repository and the length of the trail formula for the classes of $\diamond$-free and $\bigcirc$-free trail formulas.*

The next theorem shows that if we relax the condition stated in Corollary 6.12, namely, that normal variables appearing in conditions at distinct times be distinct, then the model checking problem is again NP-complete.

Theorem 6.14. *The model checking problem is NP-complete for the class of $\diamond$-free trail formulas whose conditions do not contain any unique variables.*

Proof. By Lemma 6.3 the model checking problem is in NP. It remains to show that the problem is NP-hard. In order to show NP-hardness we reduce, in polynomial time, the clique [BUCK90] of size k problem, which is known to be NP-complete [GARE79], to the model checking problem for the class of $\diamond$-free trail formulas, whose conditions do not contain any unique variables.

Let $G = (V, E)$ be a graph with $V = \{v_1, \ldots, v_m\}$. We need to solve the problem: does G contain a clique of size k or more, where $k \leq |V|$?

We first let ϕ be a one-to-one mapping from V to $\{A\} \times \Sigma^*$, where $A \in \mathcal{U}$. Next, let $H(G) = \langle I(G), R(G) \rangle$ be a Hypertext database defined by

1. $\#I(G) = |V| = m$.
2. $\forall i \in \{1, \ldots, m\}, \phi(v_i) = I(G)[i]$.
3. $\psi(G) = R(G)$, where ψ is an isomorphism from G to R (G) such that

$\psi(\{v_i, v_j\}) = \{(i,j),(j,i)\}$, where i and j are the page numbers satisfying $\phi(v_i) = I(G)[i]$ and $\phi(v_j) = I(G)[j]$, respectively.

Let $K = \{1,\dots,k\}$ and $\text{lex}(K) = \langle (1,2),(1,3),\dots,(1,k),(2,3),\dots,(2,k), \dots,(k-1,k)\rangle$ be the lexicographically ordered sequence [HALM74] of $(k^2-k)/2$ pairs in K^2 such that (i,j) is in $\text{lex}(K)$ iff $i < j$. Furthermore, let $\text{path}(K) = \langle 1,2,1,3, \dots,1,k,2,3,\dots,2,k,\dots,k-1,k,1\rangle$ be the sequence of the $(k^2-k)+1$ numbers in K resulting from transforming each pair (i,j) in $\text{lex}(K)$ into the subsequence i, j and then adding 1 at the end of the resulting sequence.

It can be verified that G contains a clique of size k or more iff we can find k nodes in the directed graph corresponding to $R(G)$ numbered from 1 to k such that $\text{path}(K)$ is a path in this directed graph.

We next transform $\text{path}(K)$ into the following $\Diamond$-free trail formula, denoted by $f(\text{path}(K))$, whose conditions do not contain any unique variables, namely,

$$\begin{aligned}
&att(X_1,A) \wedge \bigcirc(att(X_2,A)) \wedge \bigcirc^2(att(X_1,A)) \wedge \bigcirc^3(att(X_3,A)) \wedge \dots \wedge\\
&\bigcirc^{2(k-1)-2}(att(X_1,A)) \wedge \bigcirc^{2(k-1)-1}(att(X_k,A)) \wedge \bigcirc^{2(k-1)}(att(X_2,A)) \wedge\\
&\bigcirc^{2(k-1)+1}(att(X_3,A)) \wedge \dots \wedge \bigcirc^{4(k-2)}(att(X_2,A)) \wedge \bigcirc^{4(k-2)+1}(att(X_k,A)) \wedge \dots \wedge\\
&\bigcirc^{(k^2-k)-1}(att(X_{k-1},A)) \wedge \bigcirc^{k^2-k}(att(X_k,A)) \wedge \bigcirc^{(k^2-k)+1}(att(X_1,A)),
\end{aligned}$$

where $X_1,\dots,X_k$ are distinct normal variables and $\bigcirc^k$ denotes the composition, $\bigcirc\dots\bigcirc$, k times, with $k \in \omega$.

It can be verified that G contains a clique of size k or more iff $H(G)$ is a model of the trail formula $f(\text{path}(K))$. $\square$

Table 6.1, shown below, summarizes the complexity results obtained in this section; the reader can verify that all possible cases have been considered. "Yes" in the acyclic column indicates that the reachability relation of the Hypertext database is acyclic, "Yes" in the no-normal column indicates that trail formulas do not contain normal variables, "Yes" in the no-unique column indicates that trail formulas do not contain unique variables, "Yes" in the $\Diamond$-free column indicates that trail formulas are $\Diamond$-free, "Yes" in the $\bigcirc$-free column indicates that trail formulas are $\bigcirc$-free, "NPC" stands for NP-complete, "P" stands for polynomial time; finally, "No†" in the no-normal column indicates that normal variables appearing in conditions at distinct times are distinct.

TABLE 6.1
Summary of the complexity results.

Acyclic	No-normal	No-unique	$\Diamond$-free	$\bigcirc$-free	Complexity
Yes	No	Yes	No	Yes	NPC
Yes	Yes	No	No	Yes	NPC
No	Yes	No	Yes	No	NPC
No	No	Yes	Yes	No	NPC
Yes	No	No	Yes	No	P
No	No†	Yes	Yes	No	P
No	No	No	Yes	Yes	P

7. Concluding remarks.

An attempt has been made to tackle the navigation problem for Hypertext, namely, the problem of getting "lost in hyperspace." In order to solve this problem we utilized temporal logic and defined the navigation semantics of Hypertext in terms of the query language HQL, which is based on a subset of PLTL. In the navigation semantics of HQL the notion of a trail is central. The output to a trail query with respect to a Hypertext database is the set of all trails in the Hypertext database which satisfy the trail query.

We investigated the evaluation of HQL queries, showing that by using the automata-theoretic approach we can construct a finite automaton representing the output $f(H)$ of a trail query f with respect to a Hypertext database H. We have shown in Corollary 5.18 that the language $\mathcal{L}(f(H))$ induced by $f(H)$ is a star-free regular language. Furthermore, in Corollary 5.19 we obtained an upper bound on the time complexity of the construction of $\mathcal{M}_H \cap \mathcal{M}_f$, which is exponential in the number of conjunctions, between subformulas of f, plus one.

We also investigated the complexity of the model checking problem for HQL queries and summarized our results in Table 6.1. We have shown that, in general, this problem is NP-complete but that there are important special cases when the problem can be solved in polynomial time.

Our conclusion is that the navigation problem in Hypertext cannot be solved efficiently unless users have some a priori knowledge about the order in which the pages of information are structured in the database graph. Having this knowledge users can utilize the $\diamond$-free polynomial-time subset of HQL to pursue a one-step-at-a-time navigation session through the database graph. Experimental research has to be carried out in order to ascertain whether expensive queries will often be posed. In practice, it may also be useful to seek randomized and/or fuzzy solutions to this problem [LEVE99b].

Acknowledgments. The authors would like to thank Trevor Fenner for helping us solve some of the complexity problems posed in section 6. We would also like to thank the referee for his many incisive comments which helped us to improve an earlier version of the paper.

REFERENCES

[BEER94] C. Beeri and Y. Kornatzky, *A logical query language for hypermedia systems,* Inform. Sci., 77 (1994), pp. 1–37.

[BUCK90] F. Buckley and F. Harary, *Distance in Graphs,* Addison–Wesley, Redwood City, CA, 1990.

[BUSH45] V. Bush, *As we may think,* Atlantic Monthly, 76 (1945), pp. 101–108.

[CLAR86] E.M. Clarke, E.A. Emerson, and A.P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications,* ACM Trans. Programming Languages, 8 (1986), pp. 244–263.

[CONK87] J. Conklin, *Hypertext: An introduction and survey,* IEEE Comput., 20 (1987), pp. 17–41.

[CONS89] M.P. Consens and A.O. Mendelzon, *Expressing structural Hypertext queries in GraphLog,* in Proceedings of ACM Conference on Hypertext (Hypertext'89), ACM, New York, 1989, pp. 269–292.

[CODD79] E.F. Codd, *Extending the database relational model to capture more meaning,* ACM Trans. Database Systems, 4 (1979), pp. 379–434.

[EMER90] E.A. Emerson, *Temporal and modal logic,* in Handbook of Theoretical Computer Science, Volume B, J. Van Leeuwen, ed., Elsevier Science Publishers, Amsterdam, 1990, pp. 997–1072.

[ESPA94] J. Esparza and M. Nielson, *Decidability issues for Petri nets,* Bull. EATCS, 52 (1994), pp. 245–262.

[FRIS92] M.F. Frisse and S.B. Cousins, *Models for Hypertext,* J. Amer. Soc. Inform. Sci., 43 (1992), pp. 183–191.

[GABB80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the temporal analysis of fairness,* in Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, New York, 1980, pp. 163–173.

[GARE79] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[HALA88] F.G. Halasz, *Reflections on notecards: Seven issues for the next generation of hypermedia systems,* Comm. ACM, 31 (1988), pp. 836–852.

[HALM74] P.R. Halmos, *Naive Set Theory*, Springer-Verlag, New York, 1974.

[HOPC79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, Reading, MA, 1979.

[JONE75] N.D. Jones, *Space-bounded reducibility among combinatorial problems,* J. Comput. System Sci., 11 (1975), pp. 68–85.

[LEVE99a] M. Levene and G. Loizou, *A Guided Tour of Relational Databases and Beyond,* Springer-Verlag, London, UK, 1999.

[LEVE99b] M. Levene and G. Loizou, *A probabalistic approach to navigation in Hypertext,* Inform. Sci., 114 (1999), pp. 165–186.

[LICH84] O. Lichtenstein and A. Pnueli, *Checking that finite state concurrent programs satisfy their linear specification,* in Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, New York, 1994, pp. 97–107.

[MCNA71] R. McNaughton and S. Pappert, *Counter-Free Automata*, MIT Press, Cambridge, MA, 1971.

[MANN92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, Berlin, 1992.

[MEND95] A.O. Mendelzon and P.T. Wood, *Finding regular simple paths in graph databases,* SIAM J. Comput., 24 (1995), pp. 1235–1258.

[NYCE89] J.M. Nyce and P. Kahn, *Innovation, pragmaticism, and technological continuity: Vannevar Bush's memex,* J. Amer. Soc. Inform. Sci., 40 (1989), pp. 214–220.

[PETE81] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981.

[PERR90] D. Perrin, *Finite automata,* in Handbook of Theoretical Computer Science, Volume B, J. Van Leeuwen, ed., Elsevier Science Publishers, Amsterdam, 1990, pp. 1–57.

[RESC71] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, Wien, 1971.

[RIVL94] E. Rivlin, R. Botafogo, and B. Shneiderman, *Navigating in hyperspace: Designing a structure-based toolbox,* Comm. ACM, 37 (1994), pp. 87–96.

[SCHN88] J.L. Schnase, J. Leggett, C. Kacmar, and C. Boyle, *A Comparison of Hypertext Systems,* Research report 88-017, Hypertext Research Lab., Texas A&M University, College Station, TX, 1988.

[SEDG90] R. Sedgewick, *Algorithms in C*, Addison–Wesley, Reading, MA, 1990.

[SIST85] A.P. Sistla and E.M. Clarke, *The complexity of propositional linear temporal logics,* J. ACM, 32 (1985), pp. 733–749.

[STOT89] P.D. Stotts and R. Furata, *Petri-net-based Hypertext: Document structure with browsing semantics,* ACM Trans. Inform. Systems, 7 (1989), pp. 3–29.

[STOT92] P.D. Stotts, R. Furata, and J.C. Ruiz, *Hyperdocuments as automata: Trace-based browsing property verification,* in Proceedings of the ACM Conference on Hypertext (ECHT '92), ACM, New York, 1992, pp. 272–281.

[THOM90] W. Thomas, *Automata on infinite objects,* in Handbook of Theoretical Computer Science, Volume B, J. Van Leeuwen, ed., Elsevier Science Publishers, Amsterdam, 1990, pp. 133–191.

[TOMP89] F.W.M. Tompa, *A data model for flexible Hypertext database systems,* ACM Trans. Inform. Systems, 7 (1989), pp. 85–100.

[TREN73] M. Trenchard Jr., *Axioms and theorems for a theory of arrays,* IBM J. Res. Develop., 17 (1973), pp. 135–175.

[ULLM88] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. I, Computer Science Press, Rockville, MD, 1988.

[VAND88] A. Van Dam, *Hypertext '87 keynote address,* Comm. ACM, 31 (1988), pp. 887–895.

[VARD86a] M.Y. Vardi and P. Wolper, *Automata-theoretic techniques for modal logics of programs,* J. Comput. System Sci., 32 (1986), pp. 183–221.

[VARD86b] M.Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification,* in Proceedings of the IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 332–344.

SIAM J. Comput.
Vol. 29, No. 3, pp. 761–778

A POLYNOMIAL-TIME APPROXIMATION SCHEME FOR MINIMUM ROUTING COST SPANNING TREES[*]

BANG YE WU[†], GIUSEPPE LANCIA[‡], VINEET BAFNA[§], KUN-MAO CHAO[¶], R. RAVI[‖], AND CHUAN YI TANG[**]

Abstract. Given an undirected graph with nonnegative costs on the edges, the routing cost of any of its spanning trees is the sum over all pairs of vertices of the cost of the path between the pair in the tree. Finding a spanning tree of minimum routing cost is NP-hard, even when the costs obey the triangle inequality. We show that the general case is in fact reducible to the metric case and present a polynomial-time approximation scheme valid for both versions of the problem. In particular, we show how to build a spanning tree of an n-vertex weighted graph with routing cost at most $(1+\epsilon)$ of the minimum in time $O(n^{O(\frac{1}{\epsilon})})$. Besides the obvious connection to network design, trees with small routing cost also find application in the construction of good multiple sequence alignments in computational biology.

The communication cost spanning tree problem is a generalization of the minimum routing cost tree problem where the routing costs of different pairs are weighted by different requirement amounts. We observe that a randomized $O(\log n \log\log n)$-approximation for this problem follows directly from a recent result of Bartal, where n is the number of nodes in a metric graph. This also yields the same approximation for the generalized sum-of-pairs alignment problem in computational biology.

Key words. approximation algorithms, network design, spanning trees, computational biology

AMS subject classifications. 68W25, 68M10, 05C05, 92B02

PII. S009753979732253X

1. Introduction.

Consider the following problem in network design: given an undirected graph with nonnegative delays on the edges, the goal is to find a spanning tree such that the average delay of communicating between any pair using the tree is minimized. The delay between a pair of vertices is the sum of the delays of the edges in the path between them in the tree. Minimizing the average delay is equivalent to minimizing the total delay between all pairs of vertices in the tree.

In general, when the cost on an edge represents a price for routing messages between its endpoints (such as the delay), we define the *routing cost* for a pair of vertices in a given spanning tree as the sum of the costs of the edges in the unique tree path between them. The routing cost of the tree itself is the sum over all pairs of vertices of the routing cost for the pair in this tree.

[*]Received by the editors June 9, 1997; accepted for publication (in revised form) November 25, 1998; published electronically December 7, 1999.
http://www.siam.org/journals/sicomp/29-3/32253.html

[†]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C. (bangye@ms16.hinet.net).

[‡]Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy (lancia@dei.unipd.it).

[§]Celera Genomics, 45 West Gude Drive, Rockville, MD 20850 (Vineet.Bafna@celera.com).

[¶]Department of Computer Science and Information Management, Providence University, Shalu, Taiwan, R.O.C. (kmchao@csim.pu.edu.tw). The research of this author was supported in part by NSC grant NSC86-2213-E-126-002.

[‖]GSIA, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213 (ravi@cmu.edu). The research of this author was supported by NSF Career grant CCR-9625297.

[**]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C. (cytang@cs.nthu.edu.tw). The research of this author was supported by NSC grant NSC86-2213-E-007-008.

Finding a spanning tree of minimum routing cost in a general weighted undirected graph is known to be NP-hard [11]. In this paper we show that finding a minimum routing cost tree in a general weighted graph G is equivalent to solving the same problem on a complete graph in which the edge weights are the shortest path lengths in G. This result implies that the minimum routing tree problem with metric inputs is also NP-hard.

Wong [22] studied the minimum routing cost tree problem and presented a 2-approximation algorithm even without the metric requirement. We give a better result for the metric case, which, by the above remark, applies to the general case as well.

THEOREM 1.1. *There is a polynomial-time approximation scheme (PTAS) for finding the minimum routing cost tree of a weighted undirected graph. In particular, on an n-vertex graph, we can find a $(1+\epsilon)$-approximate solution in time $O(n^{2\lceil \frac{2}{\epsilon} \rceil - 2})$.*

Our result is derived by approximating a minimum routing cost tree by a restricted class of trees that we call *k-stars.* For any fixed size k, a k-star is a tree in which at most k vertices have degree greater than one. For a given accuracy parameter ϵ, we consider all $\lceil \frac{2}{\epsilon} - 1 \rceil$-stars and output the one with the minimum routing cost. To argue the performance guarantee, we show how a minimum routing cost tree can be converted into a k-star without much degradation in its routing cost (no more than a factor of $1 + \frac{2}{k+1}$). We also prove that for any fixed k, the minimum k-star can be determined in polynomial time. Hence, by finding the $\lceil \frac{2}{\epsilon} - 1 \rceil$-star with the minimum routing cost, we get a $(1+\epsilon)$-approximate solution.

There is an important difference between our PTAS for the routing cost tree problem and Wong's 2-approximation: While we show an approximation bound to the best tree's routing cost, Wong's proof shows that his trees have routing cost at most twice the value of the sum of pairwise distances between nodes in the input graph. This stronger connection is exploited by Gusfield [9] in an application to multiple alignments in computational biology (described later).

1.1. Optimum communication spanning trees. Hu [10] formulated a general version of the routing cost spanning tree problem that he called optimum communication spanning trees. In this problem, in addition to the costs on edges, a requirement value r_{ij} is specified for every pair of vertices i, j. The communication cost between a pair in a given spanning tree is the cost of the path between them in the tree multiplied by their requirement r_{ij}. The communication cost of the tree is the sum of all the pairwise communication costs. Thus the routing cost is a special case of the communication cost when all the requirement values are one.[1] In [10], Hu derives weak conditions under which the optimum routing cost tree is a star. In this paper, we demonstrate that simple generalizations of stars are indeed sufficient to guarantee any desirable accuracy in approximating optimal routing trees.

By using a recent result of Bartal [3] on approximating metrics probabilistically by tree metrics, we notice the following result.

THEOREM 1.2. *There is an $O(\log^2 n)$-approximation algorithm for the communication spanning tree problem on an n-node metric.*

Recent improvements to Bartal's original result in [4, 6] also lead to an improvement of the performance guarantee in Theorem 1.2 to $O(\log n \log\log n)$.

The result in Theorem 1.2 is actually stronger in the same sense as Wong [22].

[1]Hu uses the term "optimum distance spanning trees" to denote trees with minimum routing cost.

Given (symmetric) requirement values r_{ij} and metric distances d_{ij} between node pairs i, j, our approximate solution has communication cost at most $O(\log^2 n)$ times $\sum_{i,j} r_{ij} d_{ij}$. As in [9], we exploit this connection in the application to computational biology.

An overview of the remainder of the paper is as follows. In section 2 we describe the application of minimum routing cost trees to alignment problems in computational biology. In section 3 we give some basic definitions. In section 4, we show how the general case of the problem can be reduced to the metric one. Section 5 describes how k-stars provide good approximations to the optimum routing cost trees in metrics. In section 6, we discuss a polynomial algorithm for finding minimum cost k-stars in a graph. Finally, in section 7 we describe an algorithm for approximating optimum communication spanning trees.

2. An application to computational biology.

2.1. Multiple sequence alignments. Multiple sequence alignments are important tools for highlighting patterns common to a set of genetic sequences in computational biology. A multiple alignment of a set of n strings involves inserting gaps in the strings and arranging their characters into columns with n rows, one from each string. The order of characters along a row corresponding to string s_i is the same as that in s_i, with possibly some blanks inserted. The following is an example of an alignment of three strings `ATTCGAC`, `TTCCGTC`, and `ATCGTC`.

```
A T T - C G A - C
- T T C C G - T C
A - T - C G - T C
```

The intent of identifying common patterns is represented by attempting as much as possible to place the same character in every column.

The multiple sequence alignment problem has typically been formalized as an optimization problem in which some explicit objective function is minimized or maximized. One of the most popular objective functions for multiple alignment generalizes ideas from optimally aligning two sequences. The pairwise-alignment problem [21] can be phrased as that of finding a minimum mutation path between two sequences. Formally, given costs for inserting or deleting a character and for substituting one character of the alphabet for another, the problem is to find a minimum-cost mutation path from one sequence to the other. The cost of this path is the *edit distance* between them. An optimal alignment of two sequences of length l can be computed effectively by dynamic programming [14, 21] in $O(l^2)$ time. The generalization to multiple sequences leads to the sum-of-pairs objective.

The *sum-of-pairs* (SP) objective for multiple alignment is to minimize the sum, over all pairs of sequences, of the pairwise distance between them *in the alignment* (where the distance of two sequences in an alignment with l columns is obtained by adding up the costs of the pairs of characters appearing at positions $1, \ldots, l$).

Pioneering work of Sankoff and Kruskal [17] and Sankoff, Morel, and Cedergren [18] led to an exponential-time dynamic programming solution to the SP-alignment problem. A straightforward implementation requires time proportional to $2^n l^n$ for a problem with n sequences each of length at most l. Considering that in typical real-life instances l can be a few hundred, the basic dynamic programming approach turns out to be infeasible for all but very small problems. Carrillo and Lipman [5] have introduced some bounding criteria which reduce the time and space requirements of dynamic programming and make solvable problems for $n \leq 6$ and $l \leq 200$. However, constructing *optimal* alignments is bound to be computationally expensive, since the problem has been shown to be NP-complete (Wang and Jiang, [20]). Despite these very expensive solution methods, the SP-objective is implemented in several popularly available multiple alignment packages such as MACAW [19] and MSA [13].

2.2. Approximation algorithms via routing cost trees. The first approximation algorithm for the SP-alignment problem was by Gusfield [9]. It had a performance ratio of $2 - \frac{2}{n}$ where n is the number of sequences aligned. This was slightly improved to $2 - \frac{3}{n}$ by Pevzner [15]. The best-known approximation algorithm for this problem is due to Bafna, Lawler, and Pevzner [2], which achieves a ratio of $2 - \frac{r}{n}$ for any fixed value of r. The running time is exponential in r. Notice that this is not a PTAS for the problem, and no polynomial-time approximation scheme is known yet for the SP-alignment problem.

Gusfield's approximation algorithm for the SP-alignment problem is based on the 2-approximation for minimum routing cost trees due to Wong [22]. Gusfield's algorithm uses a folklore approach to multiple alignment guided by a tree, due to Feng and Doolittle [8]: Given a spanning tree on the complete graph on the sequences to be aligned, the multiple alignment guided by the tree is built recursively as follows. First, remove a leaf sequence l in the tree attached to sequence v by a tree edge (l, v), and align the remaining sequences recursively. Then, reinsert the leaf sequence into the alignment guided by an optimal pairwise alignment between the pair l and v. If this optimal pairwise alignment introduces a gap in v, insert the same gap in the recursively computed alignment for the tree without the leaf. Since the cost of aligning a blank to a blank is assumed to be zero, the resulting alignment has the property that for every pair related by a tree edge, the cost of the induced pairwise alignment equals their edit distance. By the triangle inequality on edit-distances, the SP-cost of the alignment derived from this spanning tree can be upper-bounded by the routing cost of the tree.

Wong's 2-approximation algorithm considers the shortest path tree rooted at every vertex in turn, and picks the one with minimum routing cost. For graphs with metric distances obeying the triangle inequality, every shortest path tree is isomorphic to a star. Furthermore, in this case, Wong's analysis shows that the best star has routing cost at most twice the total cost of the graph itself. The cost of the graph in this case is the sum of pairwise edit distances between sequences, which is a lower bound on the SP-cost. Thus, Gusfield observed that a multiple alignment derived from the best center-star gives a 2-approximation for the SP-alignment problem.

2.3. Tree-driven SP-alignment. Despite the popularity of the SP-objective, most of the currently available methods for finding alignments use a *progressive* approach of incrementally building the alignment adding sequences one at a time with no performance guarantee on the SP-cost. The Feng–Doolittle procedure can be viewed as one such procedure. The advantage of such approaches is their low running time, but the shortcoming is that the order in which the sequences are merged into the alignment determines its cost.

In trying to define a middle ground between the SP-objective and the more practical progressive methods, we introduce the tree-driven SP-alignment method: apply the Feng–Doolittle procedure to the *best possible* spanning tree in the complete graph on the sequences. By our reasoning above, the tree that gives the best upper bound on the SP-cost of the alignment is the one with the minimum routing cost. Thus, our PTAS for routing cost trees may be useful in finding good trees for applying any progressive alignment method such as the Feng–Doolittle procedure.

2.4. Generalized SP-alignments. A simple generalization of the SP objective for multiple alignments is to weight the different sequence pairs in the alignment differently in the objective function. Given a priority value r_{ij} for the pair i, j of sequences, the *generalized sum-of-pairs* objective for multiple alignment is to minimize the sum, over all pairs of sequences, of the pairwise distance between them in the alignment multiplied by the priority value of the pair. This allows one to increase the priority of aligning some pairs while down-weighting others, using other information (such as evolutionary) to decide on the priorities. An extreme case of assigning priorities is the *threshold* objective.

In an evolutionary context, a multiple alignment is used to reconstruct the blocks or motifs in a single ancestral sequence from which the given sequences have evolved. However, if the evolutionary events of the ancestral sequence occur randomly at a certain rate over the course of time, and independently at each location (character) of the string, after a sufficiently long time, the mutated sequence appears essentially like a random sequence compared to the initial ancestral sequence. If we postulate a threshold time beyond which this happens, this translates roughly to a threshold edit distance between the pair of sequences. The threshold objective sets r_{ij} to be one for all pairs of input sequences whose edit-distance is less than this threshold and zero for other pairs which are more distant. In this way we try to capture the most information about closely related pairs in the objective function by setting an appropriate threshold.

In the same vein as Gusfield [9], Theorem 1.2 can be used to approximate the generalized SP objective within an $O(\log^2 n)$ factor on inputs with n sequences. Let d_{ij} denote the edit distance between sequences i and j. The theorem guarantees a tree whose communication cost using the r_{ij} values given by the priority function is at most $O(\log^2 n)$ times $\sum_{i,j} r_{ij} d_{ij}$, which is a lower bound on the generalized SP value of any alignment. The Feng–Doolittle procedure guarantees that the generalized SP value of the resulting alignment is at most the communication cost of the tree which in turn is at most $O(\log^2 n)$ times the generalized SP value of any alignment.

3. Definitions.

Throughout the paper we will be referring to a given weighted, connected, undirected graph $G = (V, E, w)$, where we assume $V = \{1, \dots, n\}$ and w is a nonnegative edge weight function, not necessarily metric. For a subset $S \subseteq V$, by $\mathcal{P}(S)$ we denote the set of all unordered pairs of elements of S.

DEFINITION 3.1. *Let $G = (V, E, w)$ and $i, j \in V$. Let $S = (V_S, E_S, w)$ be a subgraph of G. By $SP(S, i, j)$ we denote a shortest path from i to j on S. When S is a tree, $SP(S, i, j)$ denotes the unique path between i and j.*

DEFINITION 3.2. *Let S be a subgraph of G and $i, j \in V$. The weight of S is denoted by $w(S) = \sum_{e \in E_S} w(e)$. The distance of i and j in S is denoted by $d_S(i, j) := w(SP(S, i, j))$. We define $d_G(i, S) = \min_{j \in V_S} d_G(i, j)$. If T is a tree and $S \subset T$, we denote the value $w(SP(T, i, j) \cap S)$ by $w_S(T, i, j)$.*

DEFINITION 3.3. *Let S be a subgraph of G. The* routing cost *of S is defined as $C(S) = \sum_{(i,j) \in \mathcal{P}(V_S)} d_S(i, j)$.*

DEFINITION 3.4. *Given a graph $G = (V, E, w)$, the* minimum routing cost spanning tree problem (MRCT) *is to find a spanning tree $\widehat{T}_G$ of G such that $C(\widehat{T}_G)$ is minimum.*

DEFINITION 3.5. *A* metric graph $G = (V, E, w)$ *is a complete graph in which* $w(i, j) \geq 0$ *and* $w(i, j) + w(j, k) \geq w(i, k)$ *for all* $i \neq j \neq k \in V$.

DEFINITION 3.6. *The* metric closure *of* G *is the complete weighted graph* $\bar{G} = (V, \mathcal{P}(V), \delta)$, *where* $\delta(i, j) := d_G(i, j)$ *for all* $(i, j) \in \mathcal{P}(V)$. *Note that* $\bar{G}$ *is a metric graph.*

DEFINITION 3.7. *Given a metric graph* G, *the* metric minimum routing cost spanning tree problem *(*ΔMRCT*) is to find a spanning tree* T *of* G *such that* $C(T)$ *is minimum.*

4. A reduction from the general to the metric case.

Let $G = (V, E, w)$ and $\bar{G} = (V, \mathcal{P}(V), \delta)$ be its metric closure. In this section, we present an algorithm which can transfer a spanning tree of $\bar{G}$ into a spanning tree of G without increasing cost. This implies that we can solve the MRCT problem on G by solving the same problem on $\bar{G}$. An edge (a, b) in $\bar{G}$ is termed a *bad edge* if $(a, b) \notin E$ or $w(a, b) > \delta(a, b)$. For any bad edge $e = (a, b)$, there must exist a path $P \neq e$ such that $w(P) = \delta(a, b)$. Given any spanning tree T of $\bar{G}$, the algorithm iteratively replaces bad edges (if any) in T with edges from the path defining the weight of the edge until there are no more bad edges in the tree. Since the resulting tree Y has no bad edge, it can be thought of as a spanning tree of G with the same cost. It will be shown later that the iteration will be executed at most $O(n^2)$ times and the cost is never increased while replacing the bad edges. The algorithm listed below details how to obtain Y from T.

Algorithm Remove_bad
Input: a spanning tree T of $\bar{G}$
Output : a spanning tree Y of G (i.e., without any bad edge) such that $C(Y) \leq C(T)$.

```
Compute all-pairs shortest paths of $G$.
while there exists a bad edge in $T$                                  (1)
    Pick a bad edge $(a, b)$. Root $T$ at $a$.
    /* assume $SP(G, a, b) = (a, x, ..., b)$ and $y$ is the father of $x$ in $T$ */
    if $b$ is not an ancestor of $x$ then
        $Y_1 = T \cup (x, b) - (a, b)$
        $Y_2 = Y_1 \cup (a, x) - (x, y)$
    else
        $Y_1 = T \cup (a, x) - (a, b)$
        $Y_2 = Y_1 \cup (b, x) - (x, y)$
    endif
    if $C(Y_1) < C(Y_2)$ then
        $Y = Y_1$
    else
        $Y = Y_2$
    endif
    $T = Y$                                                           (2)
endwhile
```

We assume that the shortest paths obtained in the beginning of the algorithm have the following property: If the obtained shortest path between a and b is $(a, x) \cup P$, then P is the obtained shortest path between x and b. Note that since x is on the shortest a-b path, $\delta(a, b) = \delta(a, x) + \delta(x, b)$.

PROPOSITION 4.1. *The loop* (1) *is executed at most* $O(n^2)$ *times.*

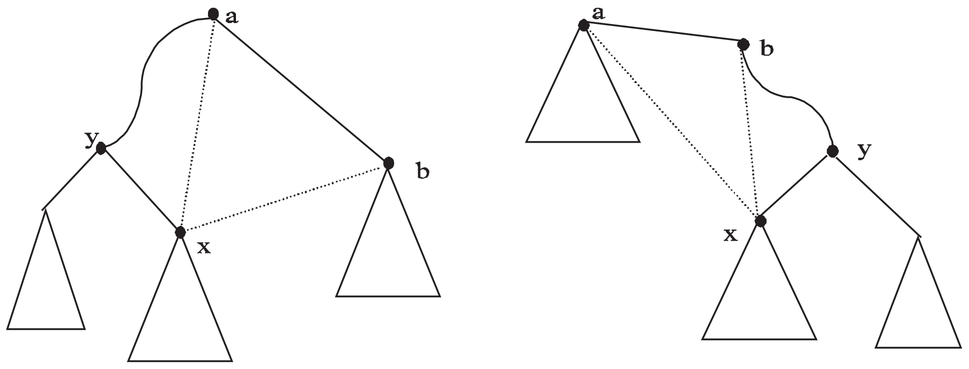


FIG. 4.1. *Remove bad edge (a, b). Case 1 (left) and Case 2 (right).*

Proof. For each bad edge $e = (a, b)$, let $l(e)$ be the number of edges in $SP(G, a, b)$ and $f(T) = \sum_{\text{bad } e} l(e)$. Since $l(e) \leq n - 1$, $f(T) < n^2$. Since $l(x, b) < l(a, b)$ and (a, x) is not a bad edge, it is easy to check that $f(T)$ decreases by at least 1 at each loop iteration. □

PROPOSITION 4.2. *Before instruction* (2) *is executed, $C(Y) \leq C(T)$.*

Proof. For any node v, define $S_v = \{u | v \text{ is an ancestor of } u \text{ on } T\} \cup \{v\}$. Also, let $C(T, S_1, S_2) = \sum_{i \in S_1, j \in S_2} d_T(i, j)$.

Case 1. (see Figure 4.1.) $x \in S_a - S_b$. If $C(Y_1) \leq C(T)$, the result follows. Otherwise, let $S_1 = S_a - S_b$ and $S_2 = S_a - S_b - S_x$. Since the distance between any two vertices both in S_1 (or both in S_b) does not change, we have

$$\begin{aligned}
&C(T) < C(Y_1) \\
\Rightarrow\ &C(T, S_1, S_b) < C(Y_1, S_1, S_b) \\
\Rightarrow\ &|S_b| C(T, a, S_1) + |S_1||S_b| \delta(a, b) < |S_b| C(T, x, S_1) + |S_1||S_b| \delta(x, b) \\
\Rightarrow\ &C(T, a, S_1) + |S_1| \delta(a, b) < C(T, x, S_1) + |S_1| \delta(x, b) \\
\Rightarrow\ &C(T, a, S_1) < C(T, x, S_1) - |S_1| \delta(a, x).
\end{aligned}$$

The last inequality follows from the property of the shortest path lengths alluded to earlier.

Also,

$$C(Y_2) - C(T) = (C(Y_2, S_2, S_x) - C(T, S_2, S_x)) + (C(Y_2, S_b, S_1) - C(T, S_b, S_1)).$$

Since $d_{Y_2}(i, j) \leq d_T(i, j)$ for $i \in S_b$ and $j \in S_1$, the second term is not positive, and

$$\begin{aligned}
&C(Y_2) - C(T) \\
&\leq C(Y_2, S_2, S_x) - C(T, S_2, S_x) \\
&= |S_x| C(T, a, S_2) + |S_2||S_x| \delta(a, x) - |S_x| C(T, x, S_2) \\
&= |S_x| ((C(T, a, S_1) - C(T, a, S_x)) + \ |S_2| \delta(a, x) - (C(T, x, S_1) - C(T, x, S_x))) \\
&= |S_x| ((C(T, a, S_1) - C(T, x, S_1)) + \ |S_2| \delta(a, x) + (C(T, x, S_x) - C(T, a, S_x))) \\
&< |S_x| \left(-|S_1| \delta(a, x) + |S_2| \delta(a, x)\right) \\
&\leq 0.
\end{aligned}$$

Case 2. $x \in S_b$. The case is identical to Case 1 if we reroot the tree at b and follow the analysis in Case 1 exchanging the roles of a and b. □

As a direct consequence of Propositions 4.1 and 4.2 we obtain the following lemma.

LEMMA 4.3. *Given a spanning tree T of $\bar{G}$, the algorithm Remove_bad constructs a spanning tree Y of G with $C(Y) \le C(T)$ in $O(n^3)$ time.*

The above lemma implies that $C(\widehat{T}_G) \le C(\widehat{T}_{\bar{G}})$. Since, for any edge, the weight on the original graph is no less than the weight on the metric closure, it is easy to see that $C(\widehat{T}_G) \ge C(\widehat{T}_{\bar{G}})$. Therefore, we have the following corollary.

COROLLARY 4.4. $C(\widehat{T}_G) = C(\widehat{T}_{\bar{G}})$.

COROLLARY 4.5. *If there is a $(1+\varepsilon)$-approximation algorithm for* ΔMRCT *with time complexity $O(f(n))$, then there is a $(1+\varepsilon)$-approximation algorithm for* MRCT *with time complexity $O(f(n)+n^3)$.*

Proof. Let G be the input graph for a MRCT problem. We can construct $\bar{G}$ in time $O(n^3)$ (see, e.g., [7]). If there is a $(1+\varepsilon)$-approximation algorithm for the ΔMRCT problem, we can compute in time $O(f(n))$ a spanning tree T_1 of $\bar{G}$ such that $C(T_1) \le (1+\varepsilon)C(\widehat{T}_{\bar{G}})$. Using Algorithm Remove_bad, we can then construct a spanning tree T_2 of G such that $C(T_2) \le C(T_1) \le (1+\varepsilon)C(\widehat{T}_{\bar{G}}) = (1+\varepsilon)C(\widehat{T}_G)$. The overall time complexity is then $O(f(n)+n^3)$. ◻

5. A PTAS for the ΔMRCT problem.

5.1. Overview. As described in the previous section, the fact that the costs w may not obey the triangle inequality is irrelevant, since we can simply replace these costs by their metric closure. Therefore, in this and the following sections we may assume that $G = (V, E, w)$ is a metric graph. We remind the reader that $n = |V|$. Also, for a subgraph G' of G, we use $V(G')$ to denote the vertex set of G'.

To establish the performance guarantee, we use k-stars, i.e., trees with no more than k internal nodes. In section 6 we show that for any constant k, the minimum routing cost k-star can be determined in polynomial (in n) time. In order to show that a k-star achieves a $(1+\epsilon)$ approximation, we show that, for any tree T and constant $\delta \le 1/2$:

1. It is possible to determine a δ-separator (a particular subtree of T to be defined later), and the separator can be cut into several δ–paths such that the total number of cut nodes and leaves of the separator is at most $\lceil \frac{2}{\delta} \rceil - 3$ (Lemma 5.9).
2. Using the separator, T can be converted into a $(\lceil \frac{2}{\delta} \rceil - 3)$-star $X(T)$, whose internal nodes are just those cut nodes and leaves. The routing cost of $X(T)$ satisfies $C(X(T)) \le (1 + \frac{\delta}{1-\delta})C(T)$ (Lemma 5.13).

By using $T = \widehat{T}_G$, $\delta = \frac{\epsilon}{1+\epsilon}$ and finding the best $(\lceil \frac{2}{\delta} \rceil - 3)$-star K, we obtain $C(K) \le C(X(\widehat{T}_G)) \le (1+\frac{\delta}{1-\delta})C(\widehat{T}_G) = (1+\epsilon)C(\widehat{T}_G)$, i.e., the desired approximation.

5.2. The δ-spine of a tree.

DEFINITION 5.1. *Let T be a spanning tree of G and S be a connected subgraph of T. A* branch *of S is a connected component of $T \setminus S$. Let $\delta \le 1/2$ be a positive number. If $|V(B)| \le \delta n$ for every branch B of S, then S is a δ-*separator *of T. A δ-separator S is* minimal *if any proper subgraph of S is not a δ-separator of T.*

Intuitively, a δ-separator is like a "center" of the tree. Starting from any node, there are sufficiently many nodes which cannot be reached without touching the separator. To illustrate the concept of separator, we examine the simplest case for $\delta = 1/2$. For any tree T, there always exists a $1/2$-separator which contains only one vertex. That is, we can always cut a tree at a node c such that each branch contains at most

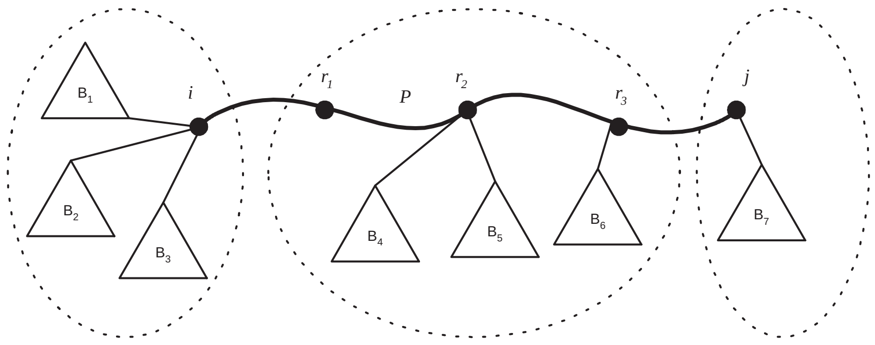


FIG. 5.1. *$B_1, \ldots, B_7$ are branches of P. $VB(T, P, i) = \{i\} \cup V(B_1) \cup V(B_2) \cup V(B_3)$. P^c is the number of vertices in $\{r_1, r_2, r_3\} \cup V(B_4) \cup V(B_5) \cup V(B_6)$.*

half of the nodes. Such a node is usually called the *centroid* of the tree in the literature. Note that this also shows the existence of a minimal δ-separator for any $\delta \leq 0.5$.

If we construct a star X centered at the centroid c, the routing cost will be at most twice that of T. This can be easily shown as follows. First, if i and j are two nodes not in the same branch, $d_T(i, j) = d_T(i, c) + d_T(j, c)$. Consider the total distance of all *ordered* pairs of nodes on T. This value is exactly $2C(T)$ by the definition. For any node i, since each branch contains no more than half of the nodes, the term $d_T(i, c)$ will be counted in the total distance at least n times, $n/2$ times for i to others, and $n/2$ times for others to i. Hence, we have $2C(T) \geq n \sum_i d_T(i, c)$. Since $C(X) = (n-1) \sum_i d_G(i, c)$, it follows that $C(X) \leq 2C(T)$. The idea in this paper can be thought as a generalization of the above method. However, the proof is much more involved.

DEFINITION 5.2. *Let T be a spanning tree of G and S be a connected subgraph of T. For any vertex i in S, $VB(T, S, i)$ denotes the set of vertex i and the vertices in the branches connected to i.*

DEFINITION 5.3. *Let $P = SP(T, i, j)$ in which $|VB(T, P, i)| \geq |VB(T, P, j)|$. We define $P^a = |VB(T, P, i)|$, $P^b = |VB(T, P, j)|$, and $P^c = n - |VB(T, P, i)| - |VB(T, P, j)|$. Assume $P = (i, r_1, r_2, \ldots, r_h, j)$. Define $Q(P) = \sum_{1 \leq x \leq h} |VB(T, P, r_x)| \times d_T(r_x, i)$.*

The above notations are defined to simplify the expressions. P^a and P^b are the numbers of vertices that are hanging off the two endpoints of the path. Note that we always assume $P^a \geq P^b$. In the case that P contains only one edge, $P^c = 0$. The notations are illustrated in Figure 5.1.

LEMMA 5.4. *Let S be a minimal δ-separator of T. If i is a leaf of S, then $|VB(T, S, i)| > \delta n$.*

Proof. If S contains only one vertex, the result is trivial since $|VB(T, S, i)| = n$. Otherwise, if $|VB(T, S, i)| \leq \delta n$, deleting i from S we still get a δ-separator. This is a contradiction to S being minimal. $\square$

DEFINITION 5.5. *Let $1 \leq k \leq n$. A* k-star *is a spanning tree of G which has no more than k internal nodes. The set of all k-stars is denoted by $k^*(G)$. T is a* minimum k-star *if $T \in k^*(G)$ and $C(T) \leq C(Y)$ for all $Y \in k^*(G)$.*

We now turn to the notions of δ-path and δ-spine. Informally, a δ-path is a path such that not too many nodes (at most $\delta n/2$) are hanging off its internal nodes. A δ-spine is a set of edge-disjoint δ-paths, whose union is a minimal δ-separator. That is, a δ-spine is obtained by cutting the minimal δ-separator into δ-paths. In the case

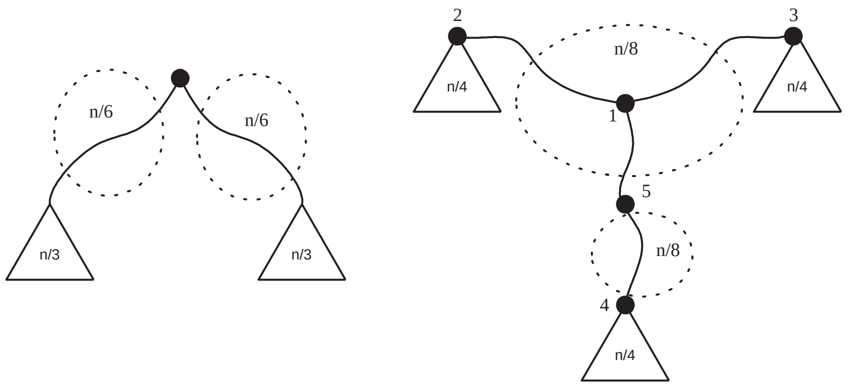


FIG. 5.2. *Trees with maximum value for the size of the minimum cut and leaf set.*

that the minimal δ-separator contains just one node, the only δ-spine is the empty set.

DEFINITION 5.6. *Given a spanning tree T of G, and $0 < \delta \leq 0.5$, a δ-path of T is a path P such that $P^c \leq \delta n/2$.*

DEFINITION 5.7. *Let $0 < \delta \leq 0.5$. A* δ-spine *$Y = \{P_1, P_2, ..., P_h\}$ of T is a set of pairwise edge-disjoint δ-paths in T such that $S = \bigcup_{1 \leq i \leq h} P_i$ is a minimal δ-separator of T. Furthermore, for any pair of distinct paths P_i and P_j in the spine, we require that either they do not intersect or, if they do, the intersection point is an endpoint of both paths.*

DEFINITION 5.8. *Let Y be a δ-spine of a tree T. $CAL(Y)$ (which stands for the* cut and leaf set *of Y) is the set of the endpoints of the paths in Y. In the case that Y is empty, the CAL set contains only one node which is the δ-separator of T. Formally $CAL(Y) = \{u | \exists P \in Y, v \in T : P = SP(T, u, v)\}$ if Y is not empty, and otherwise $CAL(Y) = \{u | u$ is the minimal δ-separator $\}$.*

Two trees achieving the maximum value for the size of the minimum CAL set for $\delta = 1/3$ ($|CAL(Y)| = 3$) and $\delta = 1/4$ ($|CAL(Y)| = 5$) are depicted in Figure 5.2. Next, we show that for any tree, there always exists a (1/3)-spine Y_1 such that $|CAL(Y_1)| \leq 3$ and a (1/4)-spine Y_2 such that $|CAL(Y_2)| \leq 5$.

LEMMA 5.9. *For any constant $0 < \delta \leq 0.5$, and spanning tree T of G, there exists a δ-spine Y of T such that $|CAL(Y)| \leq \lceil 2/\delta \rceil - 3$.*

Proof. Let S be a minimal δ-separator of T. S is a tree. Let U_1 be the set of leaves in S, U_2 be the set of vertices which have more than two neighbors in S, and $U = U_1 \cup U_2$. Let $h = |U_1|$. Clearly, $|U| \leq 2h - 2$. Let Y_1 be the set of paths obtained by cutting S at all the vertices in U_2. For example, for the tree on the right side of Figure 5.2, $U_1 = \{2, 3, 4\}$; $U_2 = \{1\}$; Y_1 contains $SP(T, 1, 2)$, $SP(T, 1, 3)$, and $SP(T, 1, 4)$. For any $P \in Y_1$, if $P^c > \delta n/2$ then P is called a *heavy* path. It is easy to check that Y_1 satisfies the requirements of a δ-spine except that there may exist some heavy paths. Suppose P is not a δ-path. We can break it up into δ-paths by the following process. First find the longest prefix of P starting at one of its endpoints and ending at some internal vertex, i, say, in the path, that determines a δ-path. Now we break P at vertex i. Then we repeat the breaking process on the remaining suffix of P starting at i stripping off the next δ-path and so on. In this way P can be cut into δ-paths by breaking it in at most $\lceil 2P^c/(\delta n) \rceil - 1$ vertices. Since there are at least δn nodes hung at each leaf,

$$\sum_{P\in Y_1} P^c < n - h\delta n.$$

Assume U_3 to be the minimal vertex set for cutting the heavy paths to result in a δ-spine Y of T. We have

$$|U_3| \leq \lceil 2\,(n - h\delta n)\,/\,(\delta n)\rceil - 1 = \lceil 2/\delta\rceil - 2h - 1.$$

So, $|CAL(Y)| = |U| + |U_3| \leq \lceil 2/\delta\rceil - 3.$ □

5.3. Lower bound.

DEFINITION 5.10. *The* routing load *of an edge e in T is the number $e^a e^b$ of pairs in T connected by a path containing e.*

The following lemma is immediate.

LEMMA 5.11. *For any spanning tree T of G, $C(T) = \sum_{e\in T} e^a e^b w(e)$.*

LEMMA 5.12. *Let Y be a δ-spine of a spanning tree T of G and $S = \bigcup_{P\in Y} P$ be a minimal δ-separator of T. Then*

$$C(T) \geq (1-\delta)n\sum_{v\in V} d_T(v,S) + \sum_{P\in Y}\left(P^b(P^a+P^c)w(P) + (P^a - P^b)Q(P)\right).$$

Proof. Since $e^a \geq (1-\delta)n$ for any edge $e \in T\setminus S$, we have

$$\begin{aligned}
C(T) &= \sum_{e\in T} e^a e^b w(e)\\
&\geq \sum_{e\in T\setminus S}(1-\delta)ne^b w(e) + \sum_{e\in S} e^a e^b w(e)\\
&\geq (1-\delta)n\sum_{v\in V} d_T(v,S) + \sum_{P\in Y}\sum_{e\in P} e^a e^b w(e).
\end{aligned}$$

Now we simplify the second term. Assume $P = (r_0, r_1, r_2, \ldots, r_h)$ in which $|VB(T,P,r_0)| \geq |VB(T,P,r_h)|$. Let $|VB(T,P,r_i)| = n_i$ for $1 \leq i \leq h-1$ and $e_i = (r_{i-1}, r_i)$ for $1 \leq i \leq h$.

$$\begin{aligned}
&\sum_{e\in P} e^a e^b w(e)\\
&= \sum_{i=1}^{h}\left(P^a + P^c - \sum_{j=i}^{h-1} n_j\right)\left(P^b + \sum_{j=i}^{h-1} n_j\right) w(e_i)\\
&\geq \sum_{i=1}^{h} P^b\left(P^a + P^c\right) w(e_i) + \left(P^a - P^b\right)\sum_{i=1}^{h}\sum_{j=i}^{h-1} n_j w(e_i)\\
&\quad + \sum_{i=1}^{h}\left(\sum_{j=i}^{h-1} n_j\right)\left(P^c - \sum_{j=i}^{h-1} n_j\right) w(e_i)\\
&\geq P^b(P^a+P^c)w(P) + (P^a - P^b)\sum_{j=1}^{h-1} n_j\left(\sum_{i=1}^{j} w(e_i)\right)\\
&= P^b\left(P^a + P^c\right) w(P) + (P^a - P^b)Q(P).
\end{aligned}$$

This completes the proof. □

5.4. From trees to stars.

LEMMA 5.13. *For any constant $0 < \delta \leq 0.5$, there exists a spanning tree $X \in (\lceil 2/\delta \rceil - 3)^*(G)$ such that $C(X) \leq \frac{1}{1-\delta} C(\widehat{T}_G)$.*

Proof. Let $T = \widehat{T}_G = (V, E, w)$ and $n = |V|$. Also, let $Y = \{P_i | 1 \leq i \leq h\}$ be a δ-spine of T in which $|CAL(Y)| \leq \lceil 2/\delta \rceil - 3$. Note that the set of all the edges in Y form a δ-separator S. Assume $P_i = SP(T, u_i, v_i)$ and $|VB(T, P_i, u_i)| \geq |VB(T, P_i, v_i)|$.

We construct a spanning tree whose internal nodes are exactly the CAL set of the δ-spine we just identified. We connect these nodes by short-cutting paths along the spine to include a set of acyclic edges with the same skeletal structure as the spine. All vertices in subtrees hanging off the CAL nodes of the spine are connected directly to their closest node in the spine. Along a δ-path in the spine, all the internal nodes and nodes in subtrees hanging off internal nodes are connected to one of the two endpoints of this path (note that both are in the CAL set of the spine) in such a way as to minimize the resulting routing cost. This is the spanning tree used to argue the upper bound on the routing cost in the proof.

More formally, construct a subgraph $R \subset G$ with vertex set $CAL(Y)$ and edge set $E_r = \{(u_i, v_i) | 1 \leq i \leq h\}$. Trivially, R is a tree. Let $f(i)$ be an indicator variable such that if $\left(P_i^a - P_i^b\right) P_i^c w(P_i) - n\left(2Q(P_i) - P_i^c w(P_i)\right) \geq 0$ then $f(i) = 1$, else $f(i) = 0$. The indicator variable $f(i)$ determines the endpoint of P_i to which all the internal nodes and nodes hanging off such internal nodes will be directly connected. We construct a spanning tree X of G where the edge set E_x is determined by the following rules:

1. $R \subset X$.
2. If $q \in VB(T, S, r)$, then $(q, r) \in E_x$, for any $r \in \{u_i, v_i | 1 \leq i \leq h\}$.
3. For the vertex set $V_i = V - VB(T, P_i, u_i) - VB(T, P_i, v_i)$, if $f(i) = 1$, then $\{(q, u_i) | q \in V_i\} \subset E_x$, else $\{(q, v_i) | q \in V_i\} \subset E_x$. That is, the vertices in V_i are either all connected to u_i or all connected to v_i.

It is easy to see that $X \in (\lceil 2/\delta \rceil - 3)^*(G)$. Let's consider the cost of X.

$$\begin{aligned} C(X) &= \sum_{e \in E_x} e^a e^b w(e) \\ &= \sum_{e \in E_r} e^a e^b w(e) + (n-1) \sum_{e \in E_x - E_r} w(e). \end{aligned}$$

First, for any $e = (u_i, v_i) \in E_r$,

$$\begin{aligned} e^a e^b w(e) &\leq \left(P_i^a + f(i) P_i^c\right)\left(P_i^b + (1 - f(i)) P_i^c\right) w(P_i) \\ &= P_i^a P_i^b w(P_i) + \left(f(i) P_i^b + (1 - f(i)) P_i^a\right) P_i^c w(P_i). \end{aligned}$$

Recall that for subset of edges $S \subset T$, $w_S(T, i, j)$ stands for $w(SP(T, i, j) \cap S)$. Second, by the triangle inequality,

$$\begin{aligned} \sum_{e \in E_x - E_r} w(e) &\leq \sum_{v \in V} d_T(v, S) + \sum_{i=1}^{h} \sum_{v \in V_i} \left(f(i) w_S(T, v, u_i) + (1 - f(i)) w_S(T, v, v_i)\right) \\ &= \sum_{v \in V} d_T(v, S) + \sum_{i=1}^{h} \left(f(i) Q(P_i) + (1 - f(i))\left(P_i^c w(P_i) - Q(P_i)\right)\right). \end{aligned}$$

Thus,

$$\begin{aligned} C(X) &\leq \sum_{i=1}^{h} P_i^a P_i^b w(P_i) + n \sum_{v\in V} d_T(v,S) \\ &\quad + \sum_{i=1}^{h} \min\{P_i^b P_i^c w(P_i) + nQ(P_i), P_i^a P_i^c w(P_i) + n(P_i^c w(P_i) - Q(P_i))\}. \end{aligned}$$

Since the minimum of two numbers is not larger than their weighted mean, we have

$$\begin{aligned} &\min\{P_i^b P_i^c w(P_i) + nQ(P_i), P_i^a P_i^c w(P_i) + n\left(P_i^c w(P_i) - Q(P_i)\right)\} \\ &\leq \left(P_i^b P_i^c w(P_i) + nQ(P_i)\right) \frac{P_i^a}{P_i^a + P_i^b} + \left(P_i^a P_i^c w(P_i) + n\left(P_i^c w(P_i) - Q(P_i)\right)\right) \frac{P_i^b}{P_i^a + P_i^b}. \end{aligned}$$

Then,

$$\begin{aligned} C(X) &\leq \sum_{i=1}^{h} P_i^a P_i^b w(P_i) + n \sum_{v\in V} d_T(v,S) + \sum_{i=1}^{h} \frac{\left(2P_i^a P_i^b P_i^c + nP_i^b P_i^c\right) w(P_i)}{P_i^a + P_i^b} \\ &\quad + \sum_{i=1}^{h} \frac{(P_i^a - P_i^b) nQ(P_i)}{P_i^a + P_i^b} \\ &= n \sum_{v\in V} d_T(v,S) + \sum_{i=1}^{h} \frac{w(P_i)}{P_i^a + P_i^b} \left(\left(P_i^a P_i^b + P_i^b P_i^c\right) n + P_i^a P_i^b P_i^c\right) \\ &\quad + \sum_{i=1}^{h} \frac{(P_i^a - P_i^b) nQ(P_i)}{P_i^a + P_i^b}. \end{aligned}$$

The simplification in the last inequality uses the observation that for any i, we have $P_i^a + P_i^b + P_i^c = n$. By Lemma 5.12,

$$C(X) \leq C(T) \max_{1\leq i\leq h} \left\{ \frac{1}{1-\delta}, \frac{n}{P_i^a + P_i^b} + \frac{P_i^a P_i^c}{(P_i^a + P_i^b)(P_i^a + P_i^c)} \right\}.$$

Since $P_i^c \leq \delta n/2$,

$$\begin{aligned} &\frac{n}{P_i^a + P_i^b} + \frac{P_i^a P_i^c}{(P_i^a + P_i^b)(P_i^a + P_i^c)} \\ &\leq \frac{n}{P_i^a + P_i^b} + \frac{P_i^c}{P_i^a + P_i^b} \\ &= \frac{n + P_i^c}{n - P_i^c} \leq \frac{2+\delta}{2-\delta} \leq \frac{1}{1-\delta}. \end{aligned}$$

This completes the proof. □

In the following section we will show that it is possible to determine the minimum k-star of a graph in polynomial time. In fact, we have the following lemma.

LEMMA 5.14. *The minimum k-star of a graph G can be constructed in time $O(n^{2k})$.*

The proof is delayed to the next section. The following theorem establishes the time-complexity of our PTAS.

THEOREM 5.15. *There exists a PTAS for the* ΔMRCT *problem, which can find a* $(1{+}\varepsilon)$*-approximation solution in* $O(n^\rho)$ *time complexity where* $\rho = 2\lceil 2/\varepsilon \rceil - 2$.

Proof. By Lemma 5.13, there exists a spanning tree $X \in (\lceil 2/\delta \rceil - 3)^*(G)$ such that $C(X) \leq \frac{1}{1-\delta} C(\widehat{T}_G)$. For finding a $(1{+}\varepsilon)$-approximation solution, we set $1/\delta = (1/\varepsilon) + 1$ and find a minimum k-star with $k = \lceil 2/\delta \rceil - 3 = \lceil 2/\varepsilon \rceil - 1$. The time complexity is $O(n^\rho)$ where $\rho = 2\lceil 2/\varepsilon \rceil - 2$ from Lemma 5.14. ◻

The result in Theorem 1.1 is immediately derived from Theorem 5.15 and Corollary 4.5.

6. Finding the best *k*-star.

In this section we describe an algorithm for finding the minimum routing cost k-star in G for a given value of k. As mentioned before, given an accuracy parameter $\epsilon > 0$, we apply this algorithm for $k = \lceil \frac{2}{\epsilon} - 1 \rceil$ and return the minimum routing cost k-star as a $(1+\epsilon)$-approximate solution.

For a given k, to find the best k-star, we consider all possible subsets S of vertices of size k, and for each such choice, find the best k-star where the remaining vertices have degree one.

6.1. A polynomial-time method.

First, we verify that the overall complexity of this step is polynomially bounded for any fixed k. Any k-star can be described by a triple $(S, \tau, \mathcal{L})$, where $S = \{v_1, \ldots, v_k\} \subseteq V$ is the set of k distinguished vertices which may have degree more than one, τ is a spanning tree topology on S, and $\mathcal{L} = (L_1, \ldots, L_k)$, where $L_i \subseteq V \setminus S$ is the set of vertices connected to vertex $v_i \in S$.

Let $l = (l_1, \ldots, l_k)$ be a nonnegative k-vector[2] such that $\sum_{i=1}^{k} l_i = n - k$. We say that a k-star $(S, \tau, \mathcal{L})$ has the configuration (S, τ, l) if $l_i = |L_i|$ for all $1 \leq i \leq k$. For a fixed k, the total number of configurations is $O(n^{2k-1})$ since there are $\binom{n}{k}$ choices for S, k^{k-2} possible tree topologies on k vertices, and $\binom{n-1}{k-1}$ possible such k-vectors. (To see this, observe that every such vector can be put in correspondence with picking $k-1$ among $n-1$ linearly ordered elements and using the cardinalities of the segments between consecutively picked segments as the components of the vector.) Note that any two k-stars with the same configuration have the same routing load on their corresponding edges. We define $\alpha(S, \tau, l)$ to be the minimum routing cost k-star with configuration (S, τ, l).

Note that any vertex v in $V \setminus S$ that is connected to a node $s \in S$ contributes a term of $w(v, s)$ multiplied by its routing load of $n-1$. Since all these routing loads are the same, the best way of connecting the vertices in $V \setminus S$ to nodes in S is obtained by finding a minimum-cost way of matching up the nodes of $V \setminus S$ to those in S which obey the degree constraints on the nodes of S imposed by the configuration, where the costs are the distances w. This problem can be solved in polynomial time for a given configuration (by a straightforward reduction to an instance of minimum-cost perfect matching). The above minimum-cost perfect matching problem, also called the *assignment* problem, has been well studied and several efficient algorithms can be found in [1]. For instance, by using an $O(n^3)$ algorithm for the assignment problem, the overall complexity would be $O(n^{2k+2})$ for finding the best k-star.

6.2. A faster method.

We now show how the minimum k-stars for the different configurations can be computed more efficiently by carefully ordering the matching problems for the configurations and exploiting the common structure of two consecutive problems. In particular, we show how we can obtain the optimal solution of any configuration in this order by performing a single augmentation on the optimal

[2]For any $r \in Z^+$, an r-vector is an integer vector with r components.

solution of the previous configuration. Thus, we show (Lemma 6.2) how to compute $\alpha(S, \tau, l)$ for a given configuration in time $O(nk)$.

Let W_{ab} be the set of all nonnegative a-vectors whose entries add up to a constant b. In $W_{ab} \times W_{ab}$, we introduce the relation $\sim$ as $l \sim l'$ if there exist $1 \leq s, t \leq a$ such that

$$l'_i = \begin{cases} l_i - 1 & \text{if } i = s, \\ l_i + 1 & \text{if } i = t, \\ l_i & \text{otherwise.} \end{cases}$$

For a pair l and l' such as the above, we say that l' *is obtained from* l *by* s *and* t.

Let $r = |W_{ab}| = \binom{a+b-1}{a-1}$. The following proposition shows that the elements of W_{ab} can be linearly ordered as $l^1, \ldots, l^r$ so that $l^{i+1} \sim l^i$ for all $1 \leq i \leq r-1$.

PROPOSITION 6.1. *For all positive integers a, b, there exists a permutation $\pi^{a,b}$ of W_{ab} such that $\pi^{a,b}_1$ is the lexicographic minimum, $\pi^{a,b}_r$ is the lexicographic maximum, and $\pi^{a,b}_{i+1} \sim \pi^{a,b}_i$ for all $i = 1, \ldots, r-1$.*

Proof. By induction. The claim is clearly true when $a = 1$ for any b. Assume the claim is true for all b when $a = m - 1$. For $a = m$ construct the ordering as follows: first, the elements for which $l_1 = 0$, ordered by applying $\pi^{a-1,b}$ to $(l_2, \ldots, l_a)$; then the elements for which $l_1 = 1$, ordered according to *decreasing* $\pi^{a-1,b-1}$. In general each block for which $l_1 = h$ is ordered by applying $\pi^{a-1,b-h}$ to $(l_2, \ldots, l_a)$, forward or backward according to the parity of h. Note that $\pi^{a,b}_{i+1} \sim \pi^{a,b}_i$ within one block. Furthermore, at block boundaries the part $(l_2, \ldots, l_a)$ is either a lexicographic minimum or maximum so that it is feasible to increase by one l_1. Finally, it is obvious that the first and the last of the constructed ordering are the lexicographic minimum and maximum respectively. ∎

According to Proposition 6.1 we can order the elements of $W_{k,(n-k)}$ as $l^1, \ldots, l^r$, where $r = \binom{n-1}{k-1}$. Note that $l^1 = (0, \ldots, 0, n-k)$ and $l^r = (n-k, 0, \ldots, 0)$. In the remainder of this section, we shall prove the following lemma.

LEMMA 6.2. *$\alpha(S, \tau, l^{i+1})$ can be computed from $\alpha(S, \tau, l^i)$ in $O(nk)$ time.*

Proof. We shall show that $\alpha(S, \tau, l^{i+1})$ can be found from $\alpha(S, \tau, l^i)$ by means of a shortest path computation. A similar argument is used in [1, Exercise 10.20]; for solving a minimum cost flow problem given the solution of another minimum cost flow problem which differs by only one unit capacity arc.

For convenience, let us rename the vertices so that $S = \{1, \ldots, k\}$. Let $l^i = (|L_1|, \ldots, |L_k|)$ and $(S, \tau, \mathcal{L}) = \alpha(S, \tau, l^i)$. Let us define an auxiliary weighted digraph $D(\mathcal{L}) = (V, A, \delta)$ in which the arc set is $A = \{(u, v) | u \in V \setminus S, v \in S\} \cup \{(u, v) | u \in S, v \in L_u\}$ and $\delta(u, v) = w(u, v)$ if $u \notin S$, and $\delta(u, v) = -w(u, v)$ if $u \in S$. For a node in S, the weight on an outgoing arc reflects the cost reduction for removing a leaf from its neighbors, and the weight on an incoming arc reflects the increase in cost for connecting a leaf to the node.

It is immediately seen that any cycle (not necessary simple) in the graph describes a way of changing $(S, \tau, \mathcal{L})$ into another k-star with the same configuration, and the difference in cost between the new and the old k-stars is given by $(n-1)$ times the length of the cycle. Because $(S, \tau, \mathcal{L})$ is optimal for its configuration, there is no negative length cycle in $D(\mathcal{L})$.

Similarly, if l^{i+1} is obtained from l^i by s and t, then any path from s to t in $D(\mathcal{L})$ changes $(S, \tau, \mathcal{L})$ into a k-star with configuration (S, τ, l^{i+1}). Conversely, any k-star with configuration (S, τ, l^{i+1}) can be obtained by a path from s to t and possibly some cycles. Since positive length cycles contribute positive cost and there is no negative length cycle, it is clear that there is a path P from s to t, changing $\alpha(S,\tau,l^i)$ into $\alpha(S,\tau,l^{i+1})$, which is simple. To see that it must be a shortest s-t path, let P' be any path from s to t, which changes $\alpha(S,\tau,l^i)$ into a k-star K'. Since K' has the same configuration of $\alpha(S,\tau,l^{i+1})$, the difference between their costs is given by $(n-1)(\delta(P')-\delta(P))$. Therefore, we conclude that P must be the shortest path from s to t in $D(\mathcal{L})$. We now show how such a shortest path can be computed in $O(kn)$ time.

Consider any shortest path $(u_1, v_1, u_2, \dots, v_{h-1}, u_h)$ between two nodes $u_i \in S$ and $v_i \in V \setminus S$ in $D(\mathcal{L})$. Take two consecutive edges (u_i, v_i) and (v_i, u_{i+1}) in the path. Since the path is shortest, v_i must be such as to minimize the sum of the two edge lengths. Recall that $\delta(u_i, v_i) = -w(u_i, v_i)$ and $\delta(v_i, u_{i+1}) = w(v_i, u_{i+1})$. Then, we have that the sum of the two edge lengths is $\min_{v_i \in L_i}\{w(v_i, u_{i+1}) - w(u_i, v_i)\}$. Therefore, to find the shortest path from s to t on $D(\mathcal{L})$, it is enough to construct a complete digraph $D'(\mathcal{L})$ with vertex set S and lengths δ', in which $\delta'(i,j) = \min_{v\in L_i}\{w(v,j) - w(i,v)\}$. It is easy to see that the length of the shortest path from s to t on $D'(\mathcal{L})$ is the same as the one on $D(\mathcal{L})$. Given the graph $D'(\mathcal{L})$, a shortest s-t path (and also the corresponding path on $D(\mathcal{L})$) can be found in $O(k^2)$ time. Finally, to construct $D'(\mathcal{L})$, for each vertex $i \in S$, we have to find $k-1$ minima (one for every other $j \in S$), each over a set of l_i elements. Adding up, the total time complexity is $(k-1)\sum_{i=1}^{k} l_i = (k-1)(n-k) = O(nk)$. $\square$

We are now able to prove Lemma 5.14, i.e., that a minimum k-star can be found in time $O(n^{2k})$.

Proof. When S and τ are fixed, to find an optimum k-star we begin by $\alpha(S,\tau,l^1)$, which is readily obtained by setting $L_k = V \setminus S$. Then, using Lemma 6.2, we compute the optimal k-stars for configurations $l^2, \dots, l^r$, and we report the best overall.

In general, a minimum routing cost k-star in G can be found in time $O(n^{2k})$, given by $\binom{n}{k}$ choices for S, k^{k-2} possible tree topologies, and for each fixed S and τ, $\binom{n-1}{k-1}$ configurations, of cost $O(nk)$ each. $\square$

7. Optimal communication spanning trees. We begin with a few definitions following Bartal [3]. Let V be a set of n points and let M be a metric space defined over V. The distance between i and j in M is denoted by $d_M(i,j)$. A metric N over V dominates another metric M over V if for every pair $i,j \in V$, we have $d_N(i,j) \geq d_M(i,j)$.

Definition 7.1. *A metric N over V α-approximates a metric M over V if it dominates M and for every $i,j \in V$, we have $d_N(i,j) \leq \alpha \cdot d_M(i,j)$.*

Define a tree (or additive) metric over V as a metric space corresponding to paths in a tree which contains all the points of V. Note that we allow the tree defining the additive metric to contain points other than those in V.

We are interested in tree metrics that approximate any given metric M. However, even for the simple metric induced by arranging the nodes of V in a cycle, if we restrict ourselves to approximating this by tree metrics, $\alpha = \Omega(|V|)$ [3, 16]. Hence we turn to the following notion.

Definition 7.2. *Let M be a metric space over V. A set of metric spaces S over V α-probabilistically-approximates M, if every metric space in S dominates M and there exists a probability distribution over metric spaces $N \in S$ such that for every $i,j \in V$, $E(d_N(i,j)) \leq \alpha \cdot d_M(i,j)$.*

Bartal's main result is the following theorem.

Theorem 7.3 (see [3]). *For any metric space on V, it can be $O(\log^2 |V|)$-probabilistically approximated by a set of tree metrics on V. Furthermore, the tree* *metrics and the distribution over them can be computed in polynomial time.*

As has been observed earlier [12], it is not hard to transform the tree metrics in this theorem into spanning tree metrics, namely, those that do not contain any extra points other than those in V. We use the above theorem to approximate the given metric M by spanning tree metrics N. By using a spanning tree N randomly picked from this collection according to the given distribution as the solution, the expected value of its communication cost is $\sum_{ij} r_{ij} d_N(i,j) \leq O(\log^2 |V|) \sum_{ij} r_{ij} d_M(i,j)$ by linearity of expectation. By repeatedly picking a few trees and using the best one, this bound is achieved with high probability, giving the result in Theorem 1.2. As mentioned earlier, this bound has been improved and derandomized in [4, 6].

Acknowledgments. Thanks to Sampath Kannan for describing the relevance of the threshold objective for multiple alignments. We also thank Tao Jiang and Howard Karloff for suggesting the merger of two different works to obtain this joint paper. Finally, we thank the referees for their valuable comments.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows—Theory, Algorithms, and Applications*, Prentice–Hall, Englewood Cliffs, NJ, 1993.

[2] V. Bafna, E. L. Lawler, and P. Pevzner, *Approximation algorithms for multiple sequence alignment*, Proceedings of the 5th Combinatorial Pattern Matching Conference, Lecture Notes in Comput. Sci. 807, Springer, New York, 1994, pp. 43–53.

[3] Y. Bartal, *Probabilistic approximation of metric spaces and its algorithmic applications*, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 184–193.

[4] Y. Bartal, *On approximating arbitrary metrics by tree metrics*, Proceedings of the 30th Annual ACM Symposium on Theory of Computing, Dallas, TX, 1998, pp. 161–168.

[5] H. Carrillo and D. Lipman, *The multiple sequence alignment problem in biology*, SIAM J. Appl. Math., 48 (1988), pp. 1073–1082.

[6] M. Charikar, C. Chekuri, A. Goel, and S. Guha, *Rounding via trees: Deterministic approximation algorithms for group Steiner trees and k-median*, Proceedings of the 30th Annual ACM Symposium on Theory of Computing, Dallas, TX, 1998, pp. 114–123.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1994.

[8] D. Feng and R. Doolittle, *Progressive sequence alignment as a prerequisite to correct phylogenetic trees*, J. Molecular Evol., 25 (1987), pp. 351–360.

[9] D. Gusfield, *Efficient methods for multiple sequence alignment with guaranteed error bounds*, Bull. Math. Biology, 55 (1993), pp. 141–154.

[10] T. C. Hu, *Optimum communication spanning trees*, SIAM J. Comput., 3 (1974), pp. 188–195.

[11] D. S. Johnson, J. K. Lenstra, and A. H. G. Rinnooy Kan, *The complexity of the network design problem*, Networks, 8 (1978), pp. 279–285.

[12] G. Konjevod, R. Ravi, and F. S. Salman, *On Approximating Planar Metrics by Tree Metrics*, manuscript, 1997.

[13] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, *A tool for multiple sequence alignment*, Proc. Nat. Acad. Sci. USA, 86 (1989), pp. 4412–4415.

[14] S. B. Needleman and C. D. Wunsch, *A general method applicable to search the similarities in the amino acid sequences of two proteins*, J. Molecular Biol., 48 (1970), pp. 443–453.

[15] P. A. Pevzner, *Multiple alignment, communication cost, and graph matching*, SIAM J. Appl. Math., 52 (1992), pp. 1763–1779.

[16] Y. Rabinovich and R. Raz, *Lower bounds on the distortion of embedding finite metric spaces in graphs*, Discrete Comput. Geom., 19 (1998), pp. 79–94.

[17] D. Sankoff and J. B. Kruskal, eds. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison–Wesley, Reading, MA, 1983.

[18] D. Sankoff, C. Morel, and R. J. Cedergren, *Evolution of the 5S Ribosomal RNA*, Nature New Biology, 245 (1973), pp. 232–234.

[19] G. D. Schuler, S. F. Altschul, and D. J. Lipman, *A workbench for multiple alignment construction and analysis*, Proteins Structure Function Genetics, 9 (1991), pp. 180–190.
[20] L. Wang and T. Jiang, *On the complexity of multiple sequence alignment*, J. Comput. Biol., 1 (1994), pp. 337–348.
[21] M. S. Waterman, *Introduction to Computational Biology*, Chapman & Hall, London, 1995.
[22] R. Wong, *Worst-case analysis of network design problem heuristics*, SIAM J. Alg. Discrete Methods, 1 (1980), pp. 51–63.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 779–789

COMPLEXITY OF DECIDING SENSE OF DIRECTION[*]

PAOLO BOLDI[†] AND SEBASTIANO VIGNA[†]

Abstract. In this paper we prove that deciding whether a distributed system (represented as a colored digraph with n nodes) has weak sense of direction is in AC^1 (using n^6 processors). Moreover, we show that deciding sense of direction is in P. Our algorithms can also be used to decide in AC^1 whether a colored graph is a Cayley color graph.

Key words. distributed systems, computational complexity, sense of direction, Cayley graphs

AMS subject classifications. 68Q25, 68M14, 68R10, 68Q45

PII. S0097539796310801

1. Introduction.

The theory of distributed computing aims at understanding the nature of cooperation in a distributed environment, where processing is carried on by agents with autonomous computing abilities, which can communicate by exchanging messages along communication links. The topological structure of such systems can be described by a graph, with nodes representing agents and arcs representing links. Each node has a local (partial) view of the system, and it associates a different label (color) with each of its incident links; in general, the color assigned to a certain link by a node may differ from the one assigned to the same link by the partner, because colors are assigned locally.

The solution to many problems in a distributed system can be greatly simplified by using colorings with special properties. In this paper we analyze from the point of view of complexity theory a property known as *(weak) sense of direction* [FMS98]; more precisely, we study the complexity of deciding whether a network with a given coloring has (weak) sense of direction or not.

Sense of direction can be intuitively described as follows. Suppose that every time a message passes through a link, it picks up the color of the link. At destination, the message will possess the entire string of colors of the links it passed through along its route. Our problem is to tell whether two messages have been sent by the same source or not, just by looking at their strings of colors. Since each node assigns a different color to each of its incident links, if every agent has a complete view of the system (and of every local coloring as well), the problem can be solved by an easy local algorithm that uses a linear backtracking technique to single out the source. But is there a way to solve the problem *globally*, with a unique function that also abstracts partially from the overall knowledge of the system? This possibility depends on some consistency property of the coloring.

Properties of global consistency have been known and used informally for a long time (see, for instance, [SAN84, LMW86, AvLSZ89]), but they were not formalized until [FMS98]: that definition, now widely accepted, allowed their systematic study. In the present paper, we will characterize for the first time weak sense of direction in a combinatorial manner and show that it can be decided efficiently in parallel;

[*]Received by the editors October 18, 1996; accepted for publication (in revised form) February 23, 1999; published electronically December 9, 1999. A preliminary version of this paper appeared in *Structure, Information, and Communication Complexity, Proc. 2nd Colloquium SIROCCO '95*, Lefteris M. Kirousis and Evangelos Kranakis, eds., Carleton University Press, 1996.

http://www.siam.org/journals/sicomp/29-3/31080.html

[†]Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41, I-20135 Milano MI, Italia (vigna@acm.org, boldi@dsi.unimi.it).

namely, we will prove that the presence of weak sense of direction can be decided in logarithmic time using n^6 processors.

However, having weak sense of direction is in general not enough for practical purposes. What one usually wants is the possibility of coding the identity of the source locally in each node, without using a complete path to decode it (see, for instance, [FMS97]). In other words, each node should be able to infer the local name of the source simply using the color of the link from which the message has arrived, and the name under which the next-to-last node on the path knows the source. This property is known as *sense of direction*: again, we will characterize it combinatorially, and we will show that it is decidable in sequential polynomial time.

The upper bounds we derive are the first nontrivial ones for these decision problems, even though the algorithms we propose are not practical (due to the high number of processors, in the first case, and to the high degree of the polynomial bound, in the second case). In the last section we give some concluding remarks and notice that our results apply also to the recognition of Cayley color graphs.

2. Definitions. A *directed graph* (or, in short, a graph) G is given by a set V of n nodes and a set $A \subseteq V \times V$ of arcs. We write $P[x, y] \subseteq A^*$ for the set of paths from the node x to the node y.

An *(arc) coloring* of a graph G is a function $\lambda : A \to \mathcal{L}$, where $\mathcal{L}$ is a finite set of colors.[1] We say that λ is *deterministic* iff

$$\lambda(\langle x, y\rangle) = \lambda(\langle x, z\rangle) \implies y = z,$$

that is, if the automaton described by the transition graph G with coloring λ is deterministic. We shall frequently shift between graph-theoretic and automata-theoretic concepts when talking about a colored graph.

Our (colored) graphs will be always represented by (colored) adjacency matrices, that is, $n \times n$ matrices such that the entry indexed by $x, y \in V$ will contain 0 if no arc connects x to y; otherwise, it will contain a positive integer representing the color of the arc (or 1, if the graph has no coloring).

Given a graph G deterministically colored by λ (we shall omit in the future to mention that λ is deterministic; all colorings in this paper are such), let

$$L(x, y) = \{\, \lambda^*(\pi) \mid \pi \in P[x, y]\,\},$$

where $\lambda^* : A^* \to \mathcal{L}^*$ is defined by

$$\lambda^*(\langle x_1, x_2\rangle\langle x_2, x_3\rangle \cdots \langle x_{k-1}, x_k\rangle) = \lambda(\langle x_1, x_2\rangle)\lambda(\langle x_2, x_3\rangle) \cdots \lambda(\langle x_{k-1}, x_k\rangle).$$

In other words, $L(x, y)$ is the language recognized by G when x is the initial state and y is the final state. For all $I \subseteq V^2$ let

$$L_I = \bigcup_{\langle x,y\rangle \in I} L(x, y)$$

(of course, $L(x, y) = L_{\{\langle x,y\rangle\}}$). Notice that $\varepsilon \in L(x, x) \neq \varnothing$.

A *local naming* for G is a family of injective functions $\beta = \{\beta_x : V \to \mathcal{N}\}_{x \in V}$, with $\mathcal{N}$ a finite set, called the *name space*. Intuitively, each node x of G gives to each

[1] We remark that throughout this paper $|\mathcal{L}|$ is polynomially bounded in n. In particular, we can restrict without loss of generality to situations in which $|\mathcal{L}| \leq n^2$, for no more than n^2 colors can be actually "used" by λ.

other node y a name $\beta_x(y)$ taken from the name space. Since we require injectivity, we have that necessarily $|\mathcal{N}| \geq n$. We shall also write $\beta : V \times V \to \mathcal{N}$ for the "unindexed" local naming, that is, $\beta(x, y) = \beta_x(y)$.

Given a colored graph endowed with a local naming, a function $f : L_{V^2} \to \mathcal{N}$ is a (consistent) *coding function* iff

$$\forall x, y \in V \quad \forall \pi \in P[x, y], \qquad f(\lambda^*(\pi)) = \beta_x(y).$$

A coding function translates the coloring of the path along which two nodes x, y are connected into the name that x gives to y. Note that although the resulting name is *local* (i.e., x and z might choose different elements of the name space for the same node y), the coding function is *global*.

A coloring λ is a *weak sense of direction* for a graph G iff for some local naming there is a coding function.[2] We shall also say that a colored graph *has* weak sense of direction, or that λ *gives* weak sense of direction to G.

As an example, consider a $p \times q$ torus, where the links have the standard *compass coloring* (North/South/East/West). Each node with coordinates $\langle i, j\rangle$ (where $i \in \mathbf{Z}_p$, $j \in \mathbf{Z}_q$) gives to a node with coordinates $\langle h, k\rangle$ the local name $\langle i - h, j - k\rangle$. If a message arrives through a path colored by v, the receiver knows the sender under the local name $f(v) = \langle \#_N(v) - \#_S(v), \#_E(v) - \#_W(v)\rangle$, where $\#_N(v)$ is the number of occurrences of the "North" color in v, and so on. On the other hand, if we reverse the North/South coloring at just one node, then the resulting graph has no sense of direction, because there is no global way to detect whether the string "North North North" corresponds to a first-coordinate offset of 1 or 3.

Finally, a (consistent) *decoding function* is a map $h : \mathcal{L} \times \mathcal{N} \to \mathcal{N}$ that satisfies

$$\forall \langle x, y\rangle \in A \quad \forall z \in V \quad \forall \pi \in P[y, z], \qquad h(\lambda(\langle x, y\rangle), f(\lambda^*(\pi))) = \beta_x(z).$$

A decoding function translates the name given by y to z into the name given by x to z, knowing only the color of the arc connecting x and y. Note that, as in the case of the coding function, h is *global*.

A coloring λ is a *sense of direction* for a graph G iff for some local naming there is a coding function and a decoding function. Our previous example has also trivially sense of direction (just add the contribution of the last color).

The model of computation we use is a *common* CRCW PRAM (i.e., concurrent reads and writes are allowed, and all the processors participating to a concurrent write must write the same value). We shall denote with AC^k the class of problems that are solvable in time $O((\log n)^k)$ using a polynomial number of processors. (The classes AC^k were originally defined using boolean circuits, but they can be equivalently characterized in terms of parallel complexity; see [KR90].)

The notions of equivalence relation and partition will be used interchangeably, and equivalence relations will be represented using boolean matrices. Thus, if R is an equivalence relation, we shall indifferently write $x \; R \; y$, $R(x, y) = 1$ or $x, y \in I \in R$ (i.e., x and y belong to the same class I of the partition induced by R). If R and S are equivalence relations, we shall write $R \leq S$ whenever R is finer than S as an equivalence relation, that is, if $R \subseteq S$ (we shall also say that S is coarser than R). We

[2]Elsewhere weak sense of direction has been defined in a slightly different way by considering only nonempty paths. It is easy to check that the algorithms described here can be immediately adapted to that definition just by assuming $\varepsilon \notin L(x, x)$ and, consequently, performing a transitive (nonreflexive) closure of the matrix M in Proposition 4.1.

shall use the same notation, with the same meaning, for the associated partitions and boolean matrices. Composition of relations (or, equivalently, product of matrices) will be denoted by juxtaposition and reflexive-transitive closure of R by R^*.

We remark a difference with the notation of [FMS98]. Instead of using undirected, connected graphs with a coloring that is node dependent, we consider general colored directed graphs,[3] in which an arc from x to y represents the existence of a link *from y to x*; this apparently unnatural convention makes our presentation much simpler,[4] and it allows us to use a more standard way of coloring graphs, which highlights naturally the connections with automata and regular languages.

3. Coding functions and internal monodromy.

We will now give a purely combinatorial condition on the coloring of a graph that will be proved equivalent to being a weak sense of direction. From now onwards we shall work with finite graphs and stick to the notation of section 2 without further remarks.

DEFINITION 3.1. *Given a set X, a partition Π of X^2 is said to be* (internally) monodrome *iff*

$$\forall I \in \Pi, \qquad \langle x, y\rangle, \langle x, z\rangle \in I \implies y = z.$$

The previous definition will be extensively used throughout the paper. Note that an equivalence relation that is finer than a monodrome one is also monodrome.

THEOREM 3.2. *Let T be the equivalence relation on V^2 defined as the transitive closure of*

$$\langle x, y\rangle \sim \langle x', y'\rangle \iff L(x, y) \cap L(x', y') \neq \varnothing.$$

Then λ is a weak sense of direction iff T is monodrome.

The proof of the theorem is preceded by two lemmata. Intuitively, each element I of T will be an element of the name space. The name under which x knows y is precisely the (unique) $I \in T$ such that $\langle x, y\rangle \in I$. The idea behind the proof is that whenever the same string v of colors appears on two different paths of the graph, say from x to y and from x' to y', then necessarily $\beta_x(y) = f(v) = \beta_{x'}(y')$.

LEMMA 3.3. *Let $G = (V, A)$ be a graph with a weak sense of direction given by a coloring λ, and let β and f be the corresponding local naming and coding function; then, there exists a monodrome partition Π of V^2 such that $\Pi \geq T$ and $|\Pi| \leq |\mathcal{N}|$ (recall that $\mathcal{N}$ is the codomain of β).*

Proof. We will show that the nonempty fibers of β (i.e., the nonempty counter-images of singletons) form a monodrome partition Π of V^2 that is coarser than T (and, of course, $|\mathcal{N}| \geq |\operatorname{im}(\beta)| = |\Pi|$). Monodromy is trivial, as it is equivalent to injectivity of the naming functions β_x. Now suppose $\langle x, y\rangle, \langle x', y'\rangle \in I \in T$; then there is a chain

$$\langle x, y\rangle = \langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_k, y_k\rangle = \langle x', y'\rangle$$

such that $w_i \in L(x_i, y_i) \cap L(x_{i+1}, y_{i+1}) \neq \varnothing$ for $1 \leq i \leq k-1$. Thus, $\beta_{x_i}(y_i) = f(w_i) = \beta_{x_{i+1}}(y_{i+1})$, which implies that $\beta_x(y) = \beta_{x'}(y')$, that is, $\langle x, y\rangle, \langle x', y'\rangle \in J \in \Pi$. ∎

LEMMA 3.4. *For every monodrome partition $\Pi \geq T$ there exists a local naming β and a coding function f with name space Π.*

[3]It is easy to extend our results to graphs with parallel arcs if the set of arcs between two nodes is represented by a vector of integers, one for each color.

[4]The case studied in [FMS98] can be recovered by considering graphs that are symmetric, loopless, and strongly connected.

Proof. First of all, note that the nonempty L_I's form a partition of L_{V^2} when I ranges over T (which is monodrome). Indeed, if $w \in L_I \cap L_J$, then $w \in L(x,y) \cap L(x',y')$ for some $\langle x,y\rangle \in I, \langle x',y'\rangle \in J$. But then necessarily $I = J$. Of course, the partition Π enjoys a fortiori the same property.

Now let $\beta_x(y)$ be the unique element of Π containing $\langle x,y\rangle$, and let $f(w)$ be the unique $I \in \Pi$ such that $w \in L_I$. Then, β is a local naming and f is a coding function.

1. If $\beta_x(y) = \beta_x(z)$, then there is an element of Π including $\langle x,y\rangle$ and $\langle x,z\rangle$. But the definition of monodrome partition implies $y = z$.

2. If $\pi \in P[x,y]$ and $\lambda^*(\pi) = w$, then $w \in L(x,y)$. Now, if $I \in \Pi$ is the unique element of Π that contains $\langle x,y\rangle$, then

$$f(\lambda^*(\pi)) = f(w) = I = \beta_x(y). \qquad \square$$

The proof of Theorem 3.2 is now trivial. If G has weak sense of direction, T is the refinement of a monodrome partition by Lemma 3.3, and it is thus monodrome. On the other hand, by taking $\Pi = T$ in Lemma 3.4 we obtain a weak sense of direction from a graph G with T monodrome.

Note that even though monodromy and weak sense of direction are equivalent, it might still be the case that monodrome partitions coarser than T do not give the best possible (i.e., smallest) name spaces. Nonetheless, Lemmas 3.3 and 3.4 together guarantee that optimal naming is equivalent to optimal partitioning: indeed, if a weak sense of direction uses p names, then by Lemma 3.3 there is a monodrome partition coarser than T with $q \leq p$ classes, and in turn this partition gives by Lemma 3.4 a weak sense of direction using q names.

4. The decision problem for weak sense of direction.

Using the results of section 3 we will now establish that deciding whether a given colored graph G with n nodes has weak sense of direction is in AC^1 (using n^6 processors).

4.1. Checking monodromy.

In this section we will show how to build T using reflexive-transitive closures of suitable matrices.

PROPOSITION 4.1. *Let G be a graph with coloring λ. Let M be the $n^2 \times n^2$ boolean matrix defined by*

$$M(\langle x,x'\rangle,\langle y,y'\rangle) = 1 \iff \langle x,y\rangle,\langle x',y'\rangle \in A \ \wedge\ \lambda(\langle x,y\rangle) = \lambda(\langle x',y'\rangle).$$

Then

$$L(x,y) \cap L(x',y') \neq \varnothing \iff M^*(\langle x,x'\rangle,\langle y,y'\rangle) = 1.$$

Intuitively, the matrix M is the noncolored adjacency matrix of the (categorical) *product graph* $G \times G$, which has an arc colored by α between the nodes $\langle x,x'\rangle$ and $\langle y,y'\rangle$ iff G has arcs colored by α between x,y and between x',y'.

Proof. We have $L(x,y) \cap L(x',y') \neq \varnothing$ iff there is a string $w = \alpha_1\alpha_2\cdots\alpha_k$ such that $x \cdot w = y$ and $x' \cdot w = y'$,[5] that is, iff there are paths

$$\langle x, x\cdot\alpha_1\rangle\langle x\cdot\alpha_1, x\cdot\alpha_1\alpha_2\rangle\cdots\langle x\cdot\alpha_1\alpha_2\cdots\alpha_{k-1}, x\cdot w = y\rangle$$

and

$$\langle x', x'\cdot\alpha_1\rangle\langle x'\cdot\alpha_1, x'\cdot\alpha_1\alpha_2\rangle\cdots\langle x'\cdot\alpha_1\alpha_2\cdots\alpha_{k-1}, x'\cdot w = y'\rangle.$$

[5]We are again considering G as an automaton, and we are denoting by the dot operator the right action of the alphabet $\mathcal{L}$ on states; in other words, $x \cdot \alpha$ is the state reached from x along the unique arc colored by α, if it exists.

By the definition of M, $M(\langle z, z'\rangle, \langle z \cdot \alpha, z' \cdot \alpha\rangle) = 1$ whenever both $z \cdot \alpha$ and $z' \cdot \alpha$ are defined. Thus, $L(x, y) \cap L(x', y') \neq \varnothing$ is equivalent to reachability (i.e., reflexive-transitive closure) of $\langle y, y'\rangle$ from $\langle x, x'\rangle$ in the graph having M as adjacency matrix. ◻

The previous proposition implies that $T = N^*$, where N is the matrix obtained from M^* using the following permutation:

$$N(\langle x, y\rangle, \langle x', y'\rangle) = M^*(\langle x, x'\rangle, \langle y, y'\rangle).$$

Once the matrix T has been computed, the monodromy condition is written as follows:

$$\forall x, y, z \in V, \quad T(\langle x, y\rangle, \langle x, z\rangle) = 1 \implies y = z.$$

As a matter of fact, this condition has a simple formulation in terms of submatrices: if we divide T into n^2 submatrices of size $n \times n$, the n submatrices on the diagonal have to be identities.

4.2. Weak sense of direction is in $\mathbf{AC}^1$. We are now going to show that the operations described in the previous section can be realized using a logarithmic number of steps and n^6 processors. We first give a formal definition of our problem.

PROBLEM 4.1. WEAK SENSE OF DIRECTION.

Instance: *A graph G with coloring λ.*

Question: *Is λ a weak sense of direction?*

THEOREM 4.2. WEAK SENSE OF DIRECTION *is in AC^1.*

Proof. Given a colored graph G with n nodes, we can compute the matrix M (using the notation of the previous section) in constant time using n^4 processors (i.e., one processor per entry of M). Then we can compute M^* in logarithmic time using n^6 processors (see [KR90]). But N can be computed from M using n^4 processors and constant time, after which $T = N^*$ can be computed again in logarithmic time using n^6 processors. Monodromy of T can be trivially checked in constant time using n^3 processors. ◻

An $O(n^{4.752} \log n)$ sequential algorithm for weak sense of direction can be easily obtained from the results of section 4.1 by using fast boolean matrix multiplication [CW90]. With respect to this trivial sequential solution, the large number of processors used in the algorithm of Theorem 4.2 causes a nonnegligible $O(n^{1.248})$ loss of work. We also state the following result.

THEOREM 4.3. *The local naming β associated to a given T can be computed in constant time using n^6 processors.*

Proof. Given a row of T, we can compute the index of the column of the first nonzero element using n^4 processors and a constant number of steps.[6] This index identifies uniquely each equivalence class (of course, $\beta_x(y)$ is exactly the index associated to the row $\langle x, y\rangle$), and the thesis follows computing such indices in parallel for the n^2 rows of T. ◻

5. Decoding functions and left regularity. We now attack the problem of characterizing colored graphs that have sense of direction, that is, a decoding function, besides a local naming and a coding function. Note that weakness is not a "fake" notion: there are colored graphs that have weak sense of direction but do not have a decoding function. An example follows.

[6]Given a boolean vector $\mathtt{v}$ of m elements, we can compute the index of its first nonzero entry as follows: we first compute (in constant time and using $m(m+1)/2$ processors) a new vector $\mathtt{w}$ whose ith entry is $v_1 \vee v_2 \vee \cdots \vee v_i$. Then we set up m processors so that the ith processor outputs its index iff $w_{i-1} \neq w_i$ ($w_0 = 0$ by definition).

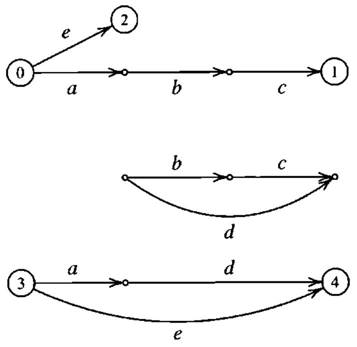


FIG. 5.1. *A graph with strictly weak sense of direction.*

The characterization of section 3 makes it clear that the graph of Figure 5.1 has weak sense of direction. (The only sets $L(x, y)$ with more than one element are $\{bc, d\}$ and $\{ad, e\}$.) Moreover, we are forced to set $f(bc) = f(d)$ for any coding function f. But as soon as we suppose the existence of a decoding function h, we obtain

$$\beta_0(1) = h(a, f(bc)) = h(a, f(d)) = \beta_3(4) = f(e) = \beta_0(2),$$

which is clearly impossible because β_0 is injective. (This example can be easily modified to obtain a symmetric connected graph with the same property—just add all arcs that are necessary assigning to each new arc a new color.)

DEFINITION 5.1. *Let G be a graph with coloring λ. A partition Π of V^2 is said to be* left regular *iff*

$$\forall I \in \Pi \quad \forall \alpha \in \mathcal{L}, \quad \exists J \in \Pi, \qquad \alpha L_I \cap L_{V^2} \subseteq L_J.$$

If a colored graph admits a left-regular monodrome partition $\Pi \geq T$, the language L_I ($I \in \Pi$), when prefixed with α, gives a sublanguage αL_I of some L_J, $J \in \Pi$. In other words, if two strings v, w live in the same language L_I, they cannot be "separated" by an α-prefixing. Note that if $\alpha L_I \cap L_{V^2} \neq \varnothing$, then J is unique.

THEOREM 5.2. *A graph G with coloring λ has sense of direction iff it admits a left-regular monodrome partition $\Pi \geq T$.*

Proof. If G admits a left-regular monodrome partition $\Pi \geq T$, we set up f and β as in Lemma 3.4, and we define $h(\alpha, f(v))$ for $\alpha \in \mathcal{L}$, $v \in L_I$ as the unique $J \in \Pi$ such that $\alpha L_I \cap L_{V^2} \subseteq L_J$ (the choice of v is, of course, irrelevant). Trivially, this definition makes h into a decoding function.

Consider now a graph G for which the coloring λ is a sense of direction. Recalling the notation of Lemma 3.3, we write Π for the monodrome partition obtained by considering the nonempty fibers of β. Thus, we can identify elements in Π with elements of the image of β. We just have to prove that Π is left regular.

Given $\alpha \in \mathcal{L}$ and $I \in \Pi$, consider strings $w_1, w_2 \in L_I$ such that $\alpha w_1, \alpha w_2 \in L_{V^2}$. Thus, we have nodes x_1, x_2 such that $x_1 \cdot \alpha w_1$ and $x_2 \cdot \alpha w_2$ are both defined. Then we have

$$\beta_{x_1}(x_1 \cdot \alpha w_1) = h(\alpha, f(w_1)) = h(\alpha, f(w_2)) = \beta_{x_2}(x_2 \cdot \alpha w_2),$$

so each string in $\alpha L_I \cap L_{V^2}$ belongs to the same L_J, $J \in \Pi$. ◻

The problem we face when using left regularity to check for the existence of a decoding function is that in principle we should check *all* possible monodrome partitions coarser than T, and such a test would clearly lead to a combinatorial explosion. The next few theorems will be helpful in reducing the computational burden of Definition 5.1.

DEFINITION 5.3. *For each $\alpha \in \mathcal{L}$, the relation $C_\alpha \subseteq V^2 \times V^2$ is defined by*

$$\langle x, y\rangle \; C_\alpha \; \langle x', y'\rangle \iff \alpha L(x,y) \cap L(x',y') \neq \varnothing.$$

For each relation $R \subseteq V^2 \times V^2$ and for each $w = \alpha_1\alpha_2 \cdots \alpha_k \in \mathcal{L}^*$, we also set

$$(CR)_w = C_{\alpha_1} R C_{\alpha_2} R \cdots C_{\alpha_k} R.$$

The operator $(-)^{C_\mathcal{L}}$, which transforms relations on V^2, is defined by

$$R^{C_\mathcal{L}} = \Big(\sum_{w \in \mathcal{L}^*} ((CR)_w)^T R (CR)_w \Big)^*.$$

We observe that $(-)^{C_\mathcal{L}}$, when applied to a *symmetric* relation, always gives an *equivalence* relation. Moreover, it is monotone (i.e., $R \leq S$ implies $R^{C_\mathcal{L}} \leq S^{C_\mathcal{L}}$), and $R \leq R^{C_\mathcal{L}}$.

PROPOSITION 5.4. *A partition Π of V^2 coarser than T is left regular iff $C_\mathcal{L}\Pi \leq \Pi$.*

Proof. We have that

$$\begin{aligned}
\Pi^{C_\mathcal{L}} \leq \Pi &\iff \Big(\sum_{w \in \mathcal{L}^*} ((C\Pi)_w)^T \Pi (C\Pi)_w \Big)^* \leq \Pi \\
&\iff \forall w \in \mathcal{L}^* \quad ((C\Pi)_w)^T \Pi (C\Pi)_w \leq \Pi \\
&\iff \forall \alpha \in \mathcal{L} \quad \Pi C_\alpha^T \Pi C_\alpha \Pi \leq \Pi \\
&\iff \forall \alpha \in \mathcal{L} \quad C_\alpha^T \Pi C_\alpha \leq \Pi.
\end{aligned}$$

Moreover, for every $I, J \in \Pi$ and $\alpha \in \mathcal{L}$,

$$\begin{aligned}
&\alpha L_I \cap L_{V^2} \subseteq L_J \\
&\iff \Big(\alpha \bigcup_{\langle x,y\rangle \in I} L(x,y) \Big) \cap L_{V^2} \subseteq L_J \\
&\iff \forall \langle \bar{x}, \bar{y}\rangle \in I \quad \alpha L(\bar{x}, \bar{y}) \cap L_{V^2} \subseteq L_J \\
&\iff \forall \langle \bar{x}, \bar{y}\rangle \in I \quad \forall \langle x, y\rangle \in V^2 \quad \alpha L(\bar{x}, \bar{y}) \cap L(x,y) \subseteq L_J \\
&\iff \forall \langle \bar{x}, \bar{y}\rangle \in I \quad \forall \langle x, y\rangle \in V^2 \quad \alpha L(\bar{x}, \bar{y}) \cap L(x,y) \neq \varnothing \Rightarrow \langle x, y\rangle \in J,
\end{aligned}$$

and the last condition is straightforwardly equivalent to $C_\alpha^T \Pi C_\alpha \leq \Pi$. ◻

We are now ready to characterize graphs that admit monodrome, left-regular partitions coarser than T.

DEFINITION 5.5. *We denote with U the least partition coarser than T and closed under $(-)^{C_\mathcal{L}}$; said otherwise, U is the least partition coarser than T such that $U^{C_\mathcal{L}} \leq U$.*

Note that since $U^{C_\mathcal{L}} \geq U$ by definition, closure under $(-)^{C_\mathcal{L}}$ can be equivalently stated by saying that $U^{C_\mathcal{L}} = U$. The partition U certainly exists because the class of equivalence relations coarser than T and closed under $(-)^{C_\mathcal{L}}$ is nonempty and closed under intersection.

THEOREM 5.6. *A graph G with coloring λ has sense of direction iff U is monodrome.*

Proof. If G has sense of direction then by Theorem 5.2 there is a left-regular monodrome partition coarser than T and, by minimality, coarser than U; thus, U is monodrome. For the other direction, just take $\Pi = U$ and use again Theorem 5.2. ∎

Finally, we show how to compute U explicitly.

THEOREM 5.7. *$U = (\cdots((T)^{C_\mathcal{L}})^{C_\mathcal{L}}\cdots)^{C_\mathcal{L}}$ (n^2 times).*

Proof. By a trivial argument of domain theory, U can be obtained by iterating the $(-)^{C_\mathcal{L}}$ operator over T until a fixed point is reached. However, T can have at most n^2 equivalence classes, and each application of $(-)^{C_\mathcal{L}}$ decreases this number by at least 1, hence the thesis. ∎

6. The decision problem for sense of direction.

Using the results of the previous section we will now establish that one can decide in (sequential) polynomial time whether a given colored graph G with n nodes has sense of direction.

6.1. Checking left regularity.

There are two main steps for computing $(-)^{C_\mathcal{L}}$: determining the matrices (relations) C_α and computing $\sum_{\alpha\in\mathcal{L}} C_\alpha \otimes C_\alpha$ (with $\otimes$ we denote the Kronecker product of matrices). This is sufficient because of the following proposition.

PROPOSITION 6.1. *Given a relation $R \subseteq V^2 \times V^2$, let $\hat{R}$ be defined by*

$$\hat{R} = \left(\sum_{\alpha\in\mathcal{L}} C_\alpha \otimes C_\alpha\right)(R \otimes R).$$

Then

$$\langle x, y\rangle \quad \sum_{w\in\mathcal{L}^*} ((CR)_w)^T R (CR)_w \quad \langle x', y'\rangle \tag{6.1}$$

iff there are $\langle \bar{x}, \bar{y}\rangle, \langle \bar{x}', \bar{y}'\rangle$ such that $\langle \bar{x}, \bar{y}\rangle\ R\ \langle \bar{x}', \bar{y}'\rangle$ and

$$\langle\langle \bar{x}, \bar{y}\rangle, \langle \bar{x}', \bar{y}'\rangle\rangle\ \hat{R}^*\ \langle\langle x, y\rangle, \langle x', y'\rangle\rangle. \tag{6.2}$$

Proof. Since

$$\hat{R} = \left(\sum_{\alpha\in\mathcal{L}} C_\alpha \otimes C_\alpha\right)(R \otimes R) = \sum_{\alpha\in\mathcal{L}} C_\alpha R \otimes C_\alpha R,$$

we have that (6.2) holds iff there is a word $w = \alpha_1\alpha_2\cdots\alpha_k \in \mathcal{L}^*$ such that

$$\langle\langle \bar{x}, \bar{y}\rangle, \langle \bar{x}', \bar{y}'\rangle\rangle\ C_{\alpha_1}RC_{\alpha_2}R\cdots C_{\alpha_k}R \otimes C_{\alpha_1}RC_{\alpha_2}R\cdots C_{\alpha_k}R\ \langle\langle x, y\rangle, \langle x', y'\rangle\rangle,$$

that is, such that

$$\langle \bar{x}, \bar{y}\rangle\ (CR)_w\ \langle x, y\rangle \quad \wedge \quad \langle \bar{x}', \bar{y}'\rangle\ (CR)_w\ \langle x', y'\rangle. \tag{6.3}$$

But (6.1) is equivalent to the existence of a word $w \in \mathcal{L}^*$ such that

$$\langle x, y\rangle\ ((CR)_w)^T R (CR)_w\ \langle x', y'\rangle,$$

which in turn is equivalent to the existence of pairs $\langle \bar{x}, \bar{y}\rangle\ R\ \langle \bar{x}', \bar{y}'\rangle$ satisfying (6.3), or equivalently (6.2). ∎

Thus, given the matrices C_α we can compute $R^{C_\mathcal{L}}$ as follows:

(1) compute $(R \otimes R)\hat{R}^*$;

(2) create a new $n^2 \times n^2$ matrix S such that $S(\langle x, y\rangle, \langle x', y'\rangle) = 1$ iff there is a $\langle \bar{x}, \bar{y}\rangle \in V^2$ satisfying $\langle\langle \bar{x}, \bar{y}\rangle, \langle \bar{x}, \bar{y}\rangle\rangle\ (R \otimes R)\hat{R}^*\ \langle\langle x, y\rangle, \langle x', y'\rangle\rangle$;

(3) compute S^*, which is exactly $R^{C_{\mathcal{L}}}$.

We start by showing how to compute the C_α's by means of the same techniques we used in proving Proposition 4.1. We shall add to each node of G a bunch of $|\mathcal{L}|$ incoming arcs starting from new "artificial" nodes and apply again the product construction.

Proposition 6.2. *Let the graph $H = (W, B)$ be defined as follows: $W = V + V \times \mathcal{L}$, $B = A \cup \{\langle\langle x, \alpha\rangle, x\rangle \mid x \in V, \alpha \in \mathcal{L}\}$. Let us extend the coloring λ of G to a coloring $\mu : B \to \mathcal{L}$ by setting $\mu(\langle\langle x, \alpha\rangle, x\rangle) = \alpha$. Let D be the $(n+n|\mathcal{L}|)^2 \times (n+n|\mathcal{L}|)^2$ matrix*

$$D(\langle s, s'\rangle, \langle t, t'\rangle) = 1 \iff \langle s, t\rangle, \langle s', t'\rangle \in B \ \wedge\ \mu(\langle s, t\rangle) = \mu(\langle s', t'\rangle),$$

where s, s', t, t' range over W. Then, for all $x, x', y, y' \in V$,

$$D^*(\langle\langle x, \alpha\rangle, x'\rangle, \langle y, y'\rangle) = 1 \iff \alpha L(x, y) \cap L(x', y') \neq \varnothing.$$

Proof. As in the proof of Proposition 4.1, we have that $D^*(\langle\langle x, \alpha\rangle, x'\rangle, \langle y, y'\rangle) = 1$ iff there is a word $w = \alpha_1\alpha_2 \cdots \alpha_k$ such that $\langle x, \alpha\rangle \cdot w = y$ and $x' \cdot w = y'$. But by construction of H this happens iff $n > 0$, $\alpha_1 = \alpha$, $\alpha_2 \cdots \alpha_k \in L(x, y)$ and $\alpha_1\alpha_2 \cdots \alpha_k \in L(x', y')$—in other words, iff $\alpha L(x, y) \ni w \in L(x', y')$. ◻

Corollary 6.3. *$C_\alpha(\langle x, y\rangle, \langle x', y'\rangle) = D^*(\langle\langle x, \alpha\rangle, x'\rangle, \langle y, y'\rangle)$.*

We finally have the following theorem.

Theorem 6.4. *Given a relation $R \subseteq V^2 \times V^2$, $C_{\mathcal{L}}R$ can be computed in logarithmic time using a polynomial number of processors.*

Proof. Recalling that $|\mathcal{L}| \leq n^2$ it is easy to see that the matrix D (with the notation of Proposition 6.2) can be built in constant time using a polynomial number of processors. Then, we just have to perform a series of transitive closures, products, and permutations, which can be computed in logarithmic time using a polynomial number of processors. The details are left to the reader. ◻

6.2. Sense of direction is in P. We now come to the main theorem of this section. We start again by formalizing our problem.

Problem 6.1. Sense of Direction.

Instance: *A graph G with coloring λ.*

Question: *Is λ a sense of direction?*

Theorem 6.5. Sense of Direction *is in P.*

Proof. We can compute T in logarithmic time using n^6 processors; thus, we can a fortiori compute it sequentially in polynomial time. To compute U, we just have to iterate n^2 times $(-)^{C_{\mathcal{L}}}(-)$ over T (see Definition 5.5), and this task can be carried out in polynomial time, since each iteration requires a fortiori polynomial sequential time by Theorem 6.4, and the number of iterations is polynomial. By Theorem 5.6, we can decide whether our graph has sense of direction by checking the monodromy of U, which can be done in $O(n^3)$ steps. ◻

Note that since the operator $(-)^{C_{\mathcal{L}}}$ is computable in parallel logarithmic time, and the bound on the number of iterations provided by Theorem 5.7 is very rough, it is an interesting open problem to characterize large classes of graphs for which a (poly)logarithmic number of iterations suffices. The number of processors that are necessary to compute $(-)^{C_{\mathcal{L}}}$ is, however, rather large, and so is, correspondingly, the polynomial bound on sequential time.

7. Conclusions.

The positive results we presented about the decision problem for (weak) sense of direction of a *colored* graph should not obscure the fact that the next and most important open problem is to decide how many colors are really sufficient to give (weak) sense of direction to a graph. More precisely, we could ask, given a graph G and a positive integer k, whether there is a (weak) sense of direction λ for G using at most k colors. Note that k is always bounded from below by the maximum outdegree of the graph, so in particular we could also ask which graphs can be given a (weak) sense of direction using a number of colors equal to the maximum outdegree.

As shown in [BV97], the outregular graphs with such a (weak) sense of direction are exactly the Cayley color graphs (i.e., Cayley graphs with the natural coloring obtained associating to each arc the element of the group that generated it). Thus, we can use the algorithm described in section 4 to recognize Cayley color graphs in parallel logarithmic time. (To our knowledge, this is the first complexity result on the recognition of Cayley color graphs.) The result above implies also that the problem of deciding whether a given graph is a Cayley graph can be reduced to deciding whether the graph is outregular and can be given a (weak) sense of direction using a number of colors equal to its outdegree; surprisingly, however, no results (in particular, no hardness results) are known about the recognition of Cayley graphs.

Acknowledgments. We would like to thank Carlo "Mere" Mereghetti for many useful discussions and Gianfrancesco "Frank" Ravasio for his careful proofreading.

REFERENCES

[AvLSZ89] H. Attiya, J. van Leeuwen, N. Santoro, and S. Zaks, *Efficient elections in chordal ring networks*, Algorithmica, 4 (1989), pp. 437–446.

[BV97] P. Boldi and S. Vigna, *Minimal sense of direction and decision problems for Cayley graphs*, Inform. Process. Lett., 64 (1997), pp. 299–303.

[CW90] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[FMS97] P. Flocchini, B. Mans, and N. Santoro, *On the impact of sense of direction on message complexity*, Inform. Process. Lett., 63 (1997), pp. 23–31.

[FMS98] P. Flocchini, B. Mans, and N. Santoro, *Sense of direction: Definitions, properties, and classes*, Networks, 32 (1998), pp. 165–180.

[KR90] R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science vol. A, J. V. Leeuwen, ed., North–Holland, Amsterdam, 1990, pp. 869–941.

[LMW86] M. C. Loui, T. A. Matsushita, and D. B. West, *Election in complete networks with a sense of direction*, Inform. Process. Lett., 22 (1986), pp. 185–187. See also [LMW88].

[LMW88] M. C. Loui, T. A. Matsushita, and D. B. West, *Corrigendum: "Election in a complete network with a sense of direction,"* Inform. Process. Lett., 28 (1988), p. 327.

[SAN84] N. Santoro, *Sense of direction, topological awareness and communication complexity*, SIGACT News, 2 (1984), pp. 50–56.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 3, pp. 790–803

RIGOROUS TIME/SPACE TRADE-OFFS FOR INVERTING FUNCTIONS*

AMOS FIAT† AND MONI NAOR‡

Abstract. We provide rigorous time/space trade-offs for inverting any function. Given a function f, we give a time/space trade-off of $TS^2 = N^3q(f)$, where $q(f)$ is the probability that two random elements (taken with replacement) are mapped to the same image under f. We also give a more general trade-off, $TS^3 = N^3$, that can invert *any* function at *any* point.

Key words. cryptography, cryptanalysis, one-way functions, randomized algorithms, random graphs, hashing data encryption standard

AMS subject classifications. 68M10, 68Q20, 68Q22, 68R05, 68R10

PII. S0097539795280512

1. Introduction.

Time/space trade-offs occur in many searching tasks. Typical examples are a $TS^2 = \tilde{O}(2^n)$ time/space trade-off for knapsack-like problems [15, 17, 10], algorithms for solving the discrete log problem [16], etc. In this paper we investigate time/space trade-offs for inverting functions.

Hellman [12] was the first to study this problem. He suggests a general time/space trade-off to invert one-way functions. Let the domain be $D = \{1, \ldots, N\}$, and let $f : D \mapsto D$ be the function to be inverted. Suppose that f is given in a black box manner, i.e., on input x the value $f(x)$ appears as output. To invert $f(x)$, a simple exhaustive search requires time $\tilde{O}(N)$ and constant space. (Throughout, we ignore low-order polylogarithmic factors in time/space requirements and use the $\tilde{O}$ notation. We say that $f \in \tilde{O}(g)$ if there exists some c such that $f \in O(g \cdot \log^c(g))$.) An extreme alternative is to precompute a table with N entries whose ith location contains the preimage of i. Constructing such a table requires preprocessing time and space $\tilde{O}(N)$, but later requires only $O(1)$ time per function inversion.

Hellman's motivation is a chosen cleartext attack on block ciphers. Given a block encryption algorithm, $E_k(x)$, consider the function f that maps the encryption key k to the encryption of a fixed cleartext block B under k, $f(k) = E_k(B)$. Inverting such a function is equivalent to breaking the encryption scheme under a fixed cleartext attack. An eavesdropper that wants to listen to many communications may find that $\tilde{O}(N)$ time per transmission is too expensive or causes an unacceptable delay. Because of the differing costs of time and memory, a memory of size $O(N)$ might be absolutely too large, while a preprocessing time of $O(N)$ may be feasible. This

*Received by the editors January 27, 1995; accepted for publication (in revised form) June 9, 1997; published electronically December 15, 1999. A preliminary version of this paper appeared in Proceedings of the 23rd ACM Symposium on Theory of Computing (STOC), New Orleans, LA, 1991, pp. 534–541.

http://www.siam.org/journals/sicomp/29-3/28051.html

†Department of Computer Science, Tel Aviv University, Tel Aviv, Israel (fiat@math.tau.ac.il). Part of this work was done while the author was visiting the International Computer Science Institute and the IBM Almaden Research Center. The research of this author was supported by a grant from the Israel Science Foundation administered by the Israel Academy of Sciences.

‡Incumbent of the Morris and Rose Goldman Career Development Chair, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel (naor@wisdom.weizmann.ac.il). Most of this work was performed while the author was at the IBM Almaden Research Center. The research of this author was supported by a grant from the Israel Science Foundation administered by the Israel Academy of Sciences.

motivates the search for time/space trade-offs that require less than $\tilde{O}(N)$ time per function inversion and also require less than $\tilde{O}(N)$ space.

Hellman assumes in his analysis that the function f that is to be inverted is a random function. He also postulates that functions obtained by minor modifications to f (permuting the output bits) can be modeled as independently chosen random functions. Under these presuppositions, Hellman's scheme [12] requires $\tilde{O}(N)$ time for preprocessing and time/space T/S per function inversion, where T and S satisfy $TS^2 = \tilde{O}(N^2)$.

We find it difficult to give an exact characterization of the assumptions actually required for Hellman's scheme to work. We cannot characterize the set of functions for which the Hellman time/space trade-off is applicable. While the construction may work for specific functions, such as the Data Encryption Standard (DES) key to ciphertext, it seems to be very difficult to prove this. We note that taking f to be polynomial time indistinguishable from a random function is insufficient.

It is possible to give functions for which Hellman's time/space construction will fail. In fact, it is relatively simple for a cryptanalyst to design a cryptographic scheme that causes the Hellman scheme to fail completely, paying for this through the inconvenience that the encryption scheme is not invertible for some negligible fraction of the keys. Our goal in this paper is to give a rigorous time/space trade-off construction that works for any function.

In this paper we give an algorithm for constructing time/space trade-offs, denoted A. Algorithm A consists of a preprocessing phase and an online inversion phase.

Our main result in this paper can be summarized in the following theorem.

THEOREM 1.1. *For any function $f : D \mapsto D$, $|D| = N$, and for any choice of values (S,T) that obey $TS^3 = N^3$, the preprocessing phase of A requires time $\tilde{O}(N)$ and space $\tilde{O}(S)$, producing a time space data structure of space $\tilde{O}(S)$.*

With probability $1 - 1/N$, over the coin tosses of the preprocessing phase, the data structure produced by the preprocessing phase allows the online inversion phase to invert f at any point y in the range of f in time $\tilde{O}(T)$.

An interesting point in this trade-off is that $T = S = N^{3/4}$. We also show that this time/space trade-off can be improved for some functions where we can bound the average number of domain elements that map to the same range value. In particular, for functions where the maximum number of domain elements that map to any range value is polylogarithmic in N, we can give a trade-off of $TS^2 = N^2$; an interesting point in this trade-off is that $T = S = N^{2/3}$.

The principle ideas used in this paper are as follows:

1. We build a table of high indegree points in the graph induced by f; this table is used to construct functions that bypass these points. This allows a better cover of the domain by random chains.
2. We use k-wise independent functions to substitute for the random functions assumed by the Hellman trade-off. An appropriate choice of parameters allows us to do so without harm.
3. We choose a specific order of events during the inversion process and allow dependencies between different k-wise independent functions. These changes are harmless and allow us to compute these k-wise independent functions in nearly constant *amortized* time.

Section 2 of this paper gives an overview of the Hellman trade-off; section 3 introduces the rigorous trade-off; section 4 describes our scheme and its analysis, while restricting the function f to be not "trivial to invert in space S." (Intuitively, if a function f is trivial to invert in space S, it implies that f is generally not useful for cryptographic applications in a scenario where the cryptanalyst has space S.) For completeness, we describe how to modify our construction to deal with trivial to invert functions in section 5; concluding the proof of Theorem 1.1, we also consider various extensions to the scheme. The last section gives some open problems associated with this work.

2. The Hellman construction.

We now review the Hellman construction, which is the starting point of our scheme. First suppose that the function $f : D \mapsto D$, $D = \{1, \ldots, N\}$ is a permutation. In this case there is an elegant time/space trade-off of $T \cdot S = \tilde{O}(N)$: consider the graph induced by f, where vertices are points in the domain and directed edges describe f's operation on the point. Since f is a permutation, this graph splits into disjoint cycles. If a cycle is less than T in length, do nothing. If the cycle is longer than T, store a set of "shortcut elements" along the cycle so that no two are farther apart than T vertices. Each one of these shortcut elements points to its closest predecessor; that is, if the shortcut elements are T apart, then z points to $f^{-T}(z)$. Now, given $y = f(x)$, it can be inverted by following the directed graph. If a pointer element is found, then follow the pointer and continue as before; stop when y is reached again. The immediate predecessor to y is its inverse.

Unfortunately, this does not work for functions f that are not permutations, and the time/space trade-off is worse. Hellman's breakthrough was to consider many related functions such that if any one of them can be inverted, then so can f. Hellman dealt with a random function $f : D \mapsto D$ and postulated that functions obtained by minor modifications to f (permuting the output bits) can be modeled as independently chosen random functions. We describe Hellman's functions differently, as the composition of f with a random function.

The Hellman trade-off curve allows one to choose space S and inversion time T subject to $TS^2 = N^2$, $|D| = N$. The values S and T are a function of three scheme parameters ℓ, m, and t, where $S = \ell m$, $T = \ell t$, and are subject to the constraints $t^2 m \leq N$ and $mt\ell = N$.

For concreteness we'll assume $\ell = m = t = N^{1/3}$, giving the most interesting trade-off point $T = N^{2/3}$, $S = N^{2/3}$. Choose $\ell = N^{1/3}$ different functions $h_i : D \mapsto D$, where $h_i(x)$ has the form $g_i(f(x))$ and g_i is a random function. For every h_i function, $1 \leq i \leq \ell$, Hellman suggests taking $m = N^{1/3}$ randomly selected points $x_{i1}, x_{i2}, \ldots, x_{i,m}$ and storing a table with m entries. Let $t = N^{1/3}$; every entry is a pair: x_{ij} and the tth iterate of h_i on x_{ij}, denoted $h_i^t(x_{ij})$ (see Figure 2.1). The table is sorted (or hashed) by $h_i^t(x_{ij})$ so that table lookup of x_{ij}, given $h_i^t(x_{ij})$, takes no more than $O(\log N)$ time. This table requires space $\tilde{O}(m) = \tilde{O}(N^{1/3})$ per h_i function and thus space $\tilde{O}(m \cdot \ell) = \tilde{O}(N^{2/3})$ for all $\ell = N^{1/3}$ tables.

To invert f at a point y, Hellman's trade-off repeats the following process for all $1 \leq i \leq \ell$. Compute the values $h_i(y), h_i^2(y), \ldots$; if you find a value $h_i^j(y)$ in the table associated with function h_i, then follow the link (the appropriate table entry) and continue applying h_i. If during this process you find a value z such that $h_i(z) = g_i(y)$, then check to see if $f(z) = y$. It can be shown that this will indeed be the case with constant probability under the appropriate idealized assumptions.

If f and the functions g_i are all independent random functions, $O(1)$ time computable, then f can be inverted at a random point with $\Omega(1)$ probability in time $\tilde{O}(\ell \cdot t) = \tilde{O}(N^{2/3})$. Under the same set of assumptions, f can be inverted at a random point with constant probability and time/space requirements obeying $TS^2 = \tilde{O}(N^2)$.

The difficulty with Hellman's construction is that it requires completely random

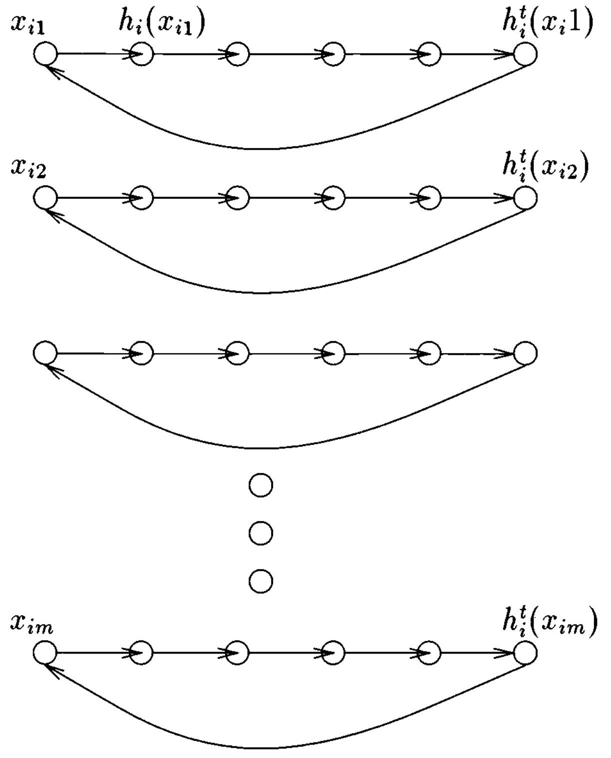


FIG. 2.1. *The chains for a specific function h_i.*

and independent g_i's, but it is not clear of course how to come up with such functions.

3. Outline of results and methodology. We do not want to assume that f has any particular structure; specifically, we do not want to assume that f is random. An adversary may devise f so as to cause the Hellman's basic cryptanalytic time/space trade-off to fail. There exist cryptographically sound functions f, e.g., polynomial time indistinguishable from a truly random function, for which Hellman's time/space trade-off fails. This occurs since the entire trade-off scheme deals with *super-polynomial* values.

Consider a function f with the property that some set of $N^{1-\epsilon}$ domain elements map to the same image, $\epsilon < 1/3$. Such a function may be polynomial time indistinguishable from a random function. Even if one assumes that random $O(1)$ time computable g_i functions are available, the Hellman time/space scheme fails with overwhelming probability. A cryptographer might devise the cryptographic scheme so that only $N-N^{1-\epsilon}$ of the keys induce a permutation, while the other keys map all cleartext values to zero. The Hellman attack on this scheme will fail.

Another assumption we wish to remove is that the random functions g_i are available. If every g_i function is described by a long table of truly random entries, then the time/space trade-off above becomes meaningless. We will use k-wise independence to limit the storage requirements of our g_i functions. We will show that an appropriate choice of k will ensure that the cryptanalytic scheme works. However, for any construction of k-wise independent functions, computing the function requires more than $\tilde{O}(1)$ time. We then show how to modify the scheme so that the FFT algorithm can be used to reduce the *amortized* computation cost of our k-wise independent g_i.

Since f can be determined by an adversary, we cannot assume anything about the structure of f. Let $I(y)$ denote the number of preimages for y under f, $I(y) = |\{x \in D | f(x) = y\}|$. We call $I(y)$ the *indegree* of y.

Images y with many preimages under f imply images $g_i(y)$ with many preimages under h_i.

Consider the sequence $I(0), I(1), \ldots, I(N)$. We are interested in the probability that two randomly chosen elements from the domain D have the same image. We denote this probability by

$$q(f) = \frac{\sum_{i=0}^{N} I(i)^2}{N^2}.$$

Under the assumptions of random g_i functions, and given $q(f)$, we can modify the parameters of the Hellman time/space trade-off of section 2 to obtain a generalized trade-off. If the g_i's are random functions, then $q(h_i) = q(g_i \circ f) \leq q(f) + 1/N$ with high probability. Any point on the time/space trade-off curve

$$T \cdot S^2 = N^3 \cdot q(f) \tag{3.1}$$

can be obtained.[1]

We now sketch the argument as to why the above trade-off is possible. Let ℓ denote the number of functions, m the number of chains per function, and t the length of each chain. Set m and t such that $m \cdot t^2 \cdot q(f) \leq 1$. We can argue that for all i, the union of all m chains contains $\Theta(m \cdot t)$ elements with high probability. Setting $\ell \cdot t \cdot m \geq N$ implies that the union of all $m \cdot \ell$ chains contains a constant fraction of all images with high probability.

In section 4 we show that we can obtain the trade-off of (3.1) without recourse to unprovable assumptions on the functions g_i. Instead, we give explicit provable constructions.

We give an alternative trade-off, which works for any function f, of arbitrary $q(f)$ such that any point on the time/space trade-off curve

$$T \cdot S^3 = N^3 \tag{3.2}$$

can be obtained.

The trade-off of (3.2) is better than the trade-off of (3.1) whenever $S > 1/q(f)$. This trade-off makes use of $O(S)$ memory so as to reduce the "effective probability of collision" to $1/S$ instead of $1/q(f)$. Thus if we modify trade-off (3.1) by replacing $q(f)$ with the "effective probability of collision," trade-off (3.2) becomes a special case. As mentioned above, we will get a low "effective probability of collision" by designing the h_i functions so as to avoid high indegree nodes under f. We give a construction to a single data structure that obeys the above trade-offs with respect to space and time, but each element can be inverted with constant probability. By repeating the construction we get that *all* of the elements can be inverted with high probability.

[1]This trade-off is not the best possible at the endpoints $T = N$ or $S = N$, in which case we can do better.

4. The construction.

4.1. Preliminaries. The function to be inverted, f, maps the domain D of size N onto itself. We may consider $D = \{1, \ldots, N\}$.

DEFINITION 4.1. *A function $f : D \mapsto D$ is said to be trivial to invert in space S if storing the S highest indegree images in a table allows one to invert $f(x)$ by table lookup for all $x \in D' \subset D$ such that $|D'| \geq N/2$.*

By definition, all functions are trivial to invert in space N. Throughout this section, we always assume that f is *not* trivial to invert in space S. If f is trivial to invert, table lookup takes care of most of the cases. We can handle trivial to invert functions as well. (See section 5.)

In the construction, we use several tables in which pairs of the form (x, y) are stored. Given a value y we need to search to see whether a pair with y is stored in the table. This can be implemented by either binary search or by hashing (see, e.g., [2, 6, 11, 9]). In any case, this is a low-order multiplicative factor that vanishes in the $\tilde{O}$ notation.

DEFINITION 4.2. *Let G be a family of functions such that for all $g \in G : g : D \mapsto R$ (where D and R are some finite sets). Let g be chosen uniformly at random from G. We call G k*-wise independent *if for all* distinct $x_1, x_2, \ldots x_k \in D$ *the random variables $g(x_1), g(x_2), \ldots, g(x_k)$ are independent and uniformly distributed in R.*

Families of k-wise independent functions have been studied and applied extensively in recent years (cf. [3, 6, 7, 8, 14, 19]). A useful property is that if G is k-wise independent, then for all $1 \leq i \leq k$, for all $x_1, x_2, \ldots, x_i \in D$ and $v_1, v_2, \ldots v_{i-1} \in R$, the distribution of $g(x_i)$ given that $(g(x_1) = v_1, g(x_2) = v_2, \ldots, g(x_{i-1}) = v_{i-1})$ is uniform in R. For our application we will need a family of functions $g : \{1, \ldots, N\} \times \{1, \ldots, N\} \mapsto \{1, \ldots, N\}$.

4.2. The scheme. Our scheme uses the parameters ℓ, t, and m to define a point on the time/space trade-off curve. We have space $S = \tilde{O}(\ell \cdot m)$ and inversion time $T = \tilde{O}(\ell \cdot t)$. The restrictions on t, ℓ, and m are that $t \leq \ell$, $t \geq 8 \log(4m)$, and $t \cdot \ell \cdot m = N$. We also use another parameter k, which is set $8 \cdot t$ in this section. This parameter is one of the few things we have to change in order to also handle trivial to invert functions in section 5. In order to achieve the time/space trade-off given in (3.2) we also use the constraint $mk^2 \leq S$, and to achieve the time/space trade-off given in (3.1) we use the constraint $mk^2 \leq 1/q(f)$.

To simplify explanations, we define $\tilde{S}$ to be $3S \cdot \log^2 N$; we actually use $\tilde{S}$ space, not S. For $S = N^\delta$, $0 < \delta < 1$, the difference between S and $\tilde{S}$ disappears in the $\tilde{O}$. The other values in the trade-off follow trivially.

Set $k = 8 \cdot t$. (This is one of the few things we have to change in order to handle trivial to invert functions as well.) The scheme uses a family G of k-wise independent functions $g : D \times \{1, \ldots, N\} \mapsto D$. In all ℓ functions, $g_1, g_2, \ldots, g_\ell$ are applied, where each g_i is uniformly distributed in G. The scheme requires that for any $1 \leq i < j \leq \ell$ the random variables g_i and g_j are independent, i.e., given the value of g_i at all the points we learn nothing about g_j. The family G and the method by which the ℓ functions are chosen is described in section 4.4. As we shall see, $g_1, g_2, \ldots, g_\ell$ are *not* completely independent.

The scheme described in the following section yields only that any element has a constant probability (over the coin flips of the preprocessing phase) of being inverted successfully. However, by repeating the full scheme (both preprocessing and the on-line inversion) $O(\log N)$ times, we can get that with probability of at least $1 - 1/N$

all range members can be inverted successfully in one of the schemes (this $O(\log N)$ term is absorbed in the $\tilde{O}$ notation).

4.2.1. Preprocessing for one data structure. The preprocessing phase consists of two stages: a choice of table A and a choice of the chains. Table A will contain pairs of values of the form $\langle x, f(x)\rangle$, sorted or hashed so that it is easy to see if $f(x)$ is in the table and to find the corresponding entry x. Table A is constructed by a random process: choose $\tilde{S}$ random values $x_1, \ldots, x_{\tilde{S}} \in D$, and store pairs $\langle x_i, f(x_i)\rangle$ in A.

Remark. Table A is used to check if a point y has high indegree. If y has high indegree, then there is high probability that $y = f(x)$ will appear in A. The preimage of y under f is used only to invert y itself, but the main use of table A is to avoid the high indegree points during the search. We use the notation $y \in A$ to mean that there is an entry of the form $\langle x, y\rangle$ in A.

The choice of A defines the functions g_i^* and h_i: For $1 \le i \le \ell$, define $g_i^*(x) = g_i(x, j)$, where $1 \le j \le k/2$ is the smallest value such that $f(g_i(x, j)) \notin A$.[2] If there is no such j, then $g_i^*(x)$ is undefined. Define $h_i(x) = g_i^*(f(x))$ (or leave undefined if $g_i^*(f(x))$ is undefined). Evaluating $h_i(x)$ requires j evaluations of f and g_i. Note that for any $x \in D$ the expected value of j (assuming that f is not trivial to invert) is $O(1)$.

The second preprocessing stage consists of choosing the chains that are supposed to cover all of the range. It is done in the following way. For all $1 \le i \le \ell$:

1. For all $1 \le j \le m$ pick x_{ij} at random from D.
2. Compute pairs of the form $(h_i^t(x_{ij}), x_{ij})$. If $h_i^t(x_{ij})$ is undefined or if computing $h_i^t(x_{ij})$ requires more than $k/2$ invocations of g_i, then discard x_{ij}. (Recall that evaluation of h_i at different points may result in a different number of invocations of g_i. Here we are interested in the total number of invocations.)
3. Store a table T_i with m entries $(h_i^t(x_{ij}), x_{ij})$.

Define a *chain* rooted at x_{ij} to be the set $\{h_i^p(x_{ij}) | 1 \le p \le t\}$. Define the ith *cluster*, $1 \le i \le \ell$, to be the set

$$C_i = \{h_i^p(x_{ij}) | 1 \le p \le t, 1 \le j \le m\}.$$

Every cluster C_i is the union of m chains of length t. The chains may overlap one another. (Two chains are said to overlap if they share one or more elements.) We say that an image $y = f(x)$ is *contained* in the chain rooted at x_{ij} if for some $w \in \{h_i^p(x_{ij}) | 0 \le p \le t-1\}$ we have $f(w) = y$. We say that y is in the ith cluster if for some $1 \le j \le m$ it is contained in the chain rooted at x_{ij}.

4.2.2. On-line inversion. We now explain how the tables constructed in the preprocessing phase are used to invert f. Given $y = f(x)$ we apply the following procedure to invert it:

- Search if $y \in A$. If found, then we're done; the inverse is the x such that $\langle x, y\rangle$ is in A.
- Otherwise, for all $1 \le i \le \ell$,
    1. Set $u_i = g_i^*(y)$ and h_i (i.e., they do not participate in the online inversion).
    2. Repeat for $p = 0$ to $t - 1$:
        (a) Search for u_i in T_i. If found ($u_i = h_i^t(x_{ij})$ for some j), then set z to be $h_i^{t-p}(x_{ij})$. If $f(z) = y$, then we have found a preimage for y.

[2]Recall that each $g_i : D \times \{1, \ldots, N\} \mapsto D$.

(b) Set $u_i \leftarrow h_i(u_i)$ if defined. Otherwise, discard u_i.

This concludes the description of the scheme, except for how the g_i's are chosen, represented, and computed.

4.3. Analysis of the scheme. We now turn to the analysis of the scheme. In this section, we use the assumption that computing the g_i functions can be done in $O(1)$ time. This will be justified in section 4.4. Since f is arbitrary, all we know about $q(f)$ is that $1/N \leq q(f) \leq 1$. The net result of our construction of table A is to reduce the effective $q(f)$ so that $q(h_i) \in O(1/S)$ with high probability. That is, given a choice of table A and given that x_1 and x_2 are chosen at random in D, we consider

$$Q_A = \text{Prob}[f(x_1) = f(x_2) \text{ and } f(x_1), f(x_2) \notin A].$$

We know that $Q_A \leq q(f)$. However, we can treat Q_A as a random variable over the probability space defined by choices of A according to the preprocessing phase of the algorithm $\mathcal{A}$. What we show is that with high probability Q_A is less than $1/S$ over this probability space.

LEMMA 4.1. *For any function f with probability at least $1 - 1/\log N$ over the random choices of A,*

$$Q_A \leq 1/S.$$

Proof. Let $I(1), I(2), \ldots, I(N)$ be the sequence of indegrees under f, i.e., $I(j) = |\{x | f(x) = j\}|$. The probability that $f(x) \notin A$ is $(1 - I(f(x))/N)^{\tilde{S}}$. Thus,

$$\begin{aligned}
E[Q_A] &= \text{Prob}[f(x_1) = f(x_2) \text{ and } f(x_1) \notin A] \\
&= \sum_{i=1}^{N} \frac{I(i)^2}{N^2} \cdot \left(1 - \frac{I(i)}{N}\right)^{\tilde{S}} \\
&= \sum_{\{j | I(j) > \frac{N}{S \log N}\}} \frac{I(j)^2}{N^2} \cdot \left(1 - \frac{I(j)}{N}\right)^{\tilde{S}} + \sum_{\{j | I(j) \leq \frac{N}{S \log N}\}} \frac{I(j)^2}{N^2} \cdot \left(1 - \frac{I(j)}{N}\right)^{\tilde{S}}.
\end{aligned}$$

We deal with each term separately. Recall that $\tilde{S} = S \log^2 N$.

$$\begin{aligned}
\sum_{\{j | I(j) > \frac{N}{S \log N}\}} \frac{I(j)^2}{N^2} \cdot \left(1 - \frac{I(j)}{N}\right)^{\tilde{S}} &\leq N \cdot 1 \cdot \left(1 - \frac{1}{S \log N}\right)^{\tilde{S}} \\
&\leq N \cdot 1/N^3 \\
&= 1/N^2.
\end{aligned}$$

The sum $\sum_{i=i}^{j} x_i^2$ subject to the constraints $\sum_{i=1}^{j} x_i \leq c$ and $x_i \leq b$ is maximized by taking c/b x_i's equal to b and setting the rest equal to zero. This is used in the following analysis:

$$\begin{aligned}
\sum_{\{j | I(j) \leq \frac{N}{S \log N}\}} \frac{I(j)^2}{N^2} \cdot \left(1 - \frac{I(j)}{N}\right)^{\tilde{S}} &\leq \sum_{\{j | I(j) \leq \frac{N}{S \log N}\}} \frac{I(j)^2}{N^2} \\
&\leq S \log N \frac{N^2}{N^2 S^2 \log^2 N} \\
&= \frac{1}{S \log N}.
\end{aligned}$$

Thus, from Markov's inequality we get that $\text{Prob}[Q_A > 1/S]$ is at most $\frac{1}{\log N}$, which suffices for our purposes. By a more careful analysis we can get a much smaller probability. $\square$

Fix $y = f(x)$; if $y \in A$, we can invert f on y via table lookup in A. Thus, we should handle only the case that $I(y) \leq N/S$ since otherwise the probability that $y \notin A$ is at most $1/N$. We assume from now on that $y \notin A$. Let V_i be the event that $f(x)$ is in the ith cluster. The main lemma we require is as follows.

LEMMA 4.2. *For all functions f, for all $y = f(x)$, and for all choices of m, t, and S such that $mt^2 < S/2$, assuming $Q_A \leq 1/S$, then either $y \in A$ or*

$$\text{Prob}[V_i] \geq \frac{mt}{2N}.$$

To prove the lemma we need the following claims.

CLAIM 4.1. *Assuming $Q_A \leq 1/S$, the probability that a chain rooted at x_{ij} contains less than t different elements is at most* $1/2m$.

Proof. We first compute the probability that the chain rooted at x_{ij} is discarded at step 2 of the preprocessing phase and show that it is negligible. Then we bound the probability that the chain cycles on h.

For a chain to be discarded at step 2 of the preprocessing phase it may be the case that more than $k/2$ applications of g_i were invoked. This occurs if through the attempted computation of h_i, $k/2$ invocations of g_i were performed altogether, while less than t of them gave images x such that $f(x)$ was not in table A. Since g_i is $(k = 8 \cdot t)$-wise independent, it follows that the probability that this occurs is at most the probability that a sequence of $k/2$ tosses of a coin contains at least $k/2 - t$ tails (the event g_i maps to a value x such that $f(x) \in A$) and less than t heads (the event g_i maps to a value x such that $f(x) \notin A$) with probabilities $p < 1/2$ for tails (by the assumption that f is not trivial to invert). By the Chernoff bound (see Theorem A.1 in [4]) this probability is at most $e^{-t^2/k} = e - t/8 \leq 1/4m$.

Even if the chain rooted at x_{ij} is not discarded, it may contain less than t different elements in case h_i cycles. This occurs if $f(z_1) = f(z_2) \notin A$, where z_1 and z_2 are two values encountered in the development of the chain either as x_{ij} or as the outcome of g_i. From the k-wise independence of g_i and the fact that g_i was invoked at most $k/2$ times in each chain we have that z_1 and z_2 are uniformly distributed at D. The probability that two such elements collide under f and not in A is at most $1/S$ by the assumption that $Q_A \leq 1/S$ is at most $1/S$. There are at most $\binom{k/2}{2}$ candidates for such pairs, so the probability that this is the case is bounded by

$$\frac{\binom{k/2}{2}}{S} < \frac{k^2}{8S} \leq \frac{1}{8m}.$$

Note that if $g_i(z_1, j_1) = g_i(z_2, j_2)$ but $f(g_i(z_1, j_1)) \in A$, where $z_1 \neq z_2$ are two values encountered in the development of the chain as the outcome of f and $1 \leq j_1, j_2 \leq N$ are the appropriate indices, then a cycle does *not* occur.

The probability that the chain is *not* discarded and that it does *not* cycle on h_i is thus at least $1 - /2m$. $\square$

Let E_{ij}^y be the event that the chain rooted at x_{ij} contains y, and let $p_y = \text{Prob}[E_{ij}^y]$.

CLAIM 4.2. *For any $y = f(x)$ such that $I(y) \leq N/S$, and for parameters as defined in Lemma* 4.2, *either $y \in A$ or*

$$p_y \geq (1 - 1/2m) \cdot \frac{t}{N}$$

under the assumption that $Q_A \leq 1/S$.

Proof. Let $w_1, w_2, \ldots, w_{k/2}$ be independent random variables uniformly distributed in D. These random variables define the random choices of the chain: each time g_i is invoked at a new point we treat its value as the next w_a (here we are using the k-wise independence of g_i). In case the chain cycles, the rest of the variables are never used, as is the case following t successful hits in the set outside A. We are interested in the event that *the chain defined by* $w_1, w_2, \ldots, w_{k/2}$ *is not discarded* and $f(w_a) = y$ *for* $1 \leq a \leq k/2$*, which is among the first* t *elements of* $w_1, w_2, \ldots, w_{k/2}$ *such that* $f(w_i) \notin A$. There are t possible places where y can appear in the chain. For each such location the probability that y is hit is at least $1/N$. Given that y is hit in the jth position, the probability that the chain is discarded or contains less than t elements is at most $1/2m$, from the analysis of Claim 4.1 (using the facts that $Q_A \leq 1/S$ and $I(y) \leq N/S$). Therefore with probability of at least $(1 - 1/2m) \cdot \frac{t}{N}$ we have that the chain contains y and is not discarded. □

From the inclusion/exclusion principle,

$$\text{Prob}[V_i] \geq \sum_{j=1}^{m} \text{Prob}[E_{ij}^y] - \sum_{j<j' \in \{1,\ldots,m\}} \text{Prob}[E_{ij}^y \cap E_{ij'}^y]. \tag{4.1}$$

The next claim says that although E_{ij}^y and $E_{ij'}^y$ are *not* independent, we can bound their correlation.

CLAIM 4.3. *Assuming $Q_A \leq 1/S$ and $I(y) \leq N/S$,* $\text{Prob}[E_{ij'}^y \cap E_{ij}^y] \leq \frac{1}{2m-1} \cdot p_y$.

Proof. The probability of the event $E_{ij'}^y \cap E_{ij}^y$ is dominated by the following: Let $w_1, w_2, \ldots, w_{k/2}$ and $w'_1, w'_2, \ldots, w'_{k/2}$ be random variables uniformly distributed in D. Let q_y be the probability that the following two conditions hold:

1. There exists $1 \leq a \leq k/2$ such that $f(w_a) = y$ and w_a is among the first t elements of $w_1, w_2, \ldots, w_{k/2}$ such that $f(w_i) \notin A$.
2. There are $1 \leq b, c \leq k/2$ such that $f(w_b) = f(w'_c) \notin A$.

That is, we are adding to $E_{ij'}^y \cap E_{ij}^y$ all the cases that chain x_{ij} hits y but cycles and the cases where the two chains collide after y is hit. By Claim 4.2 the probability that condition 1 is satisfied is at most $p_y/(1-1/2m) = p_y \frac{2m}{2m-1}$. Given that condition 1 is satisfied, the probability that condition 2 holds is at most $(k/2)^2/S \leq 1/4m$. Therefore

$$\text{Prob}[E_{ij'}^y \cap E_{ij}^y] \leq q_y \leq \frac{2m}{4m(2m-1)} \cdot p_y < \frac{1}{2m-1} \cdot p_y. \qquad \square$$

From the last claim, it follows that (4.1) is bounded from below by

$$m \cdot p_y - \binom{m}{2} \cdot p_y \cdot \frac{1}{2m-1} \geq m p_y \left(1 - \frac{m-1}{2m-1}\right) \geq \frac{mt}{2N},$$

where the last inequality holds from Claim 4.2. It follows that $\text{Prob}[V_i] \geq \frac{mt}{2N}$. □

LEMMA 4.3. *Assuming $Q_A \leq 1/S$,* $\text{Prob}[\cup_{i=1}^{\ell} V_i] \geq \frac{1}{4}$.

Proof. Given table A, the events V_i and V_j are independent since the functions g_i and g_j are independent. (Note that prior to the choice of A, the events may not be independent since V_i's imply a successful choice of A, which effects $\text{Prob}[V_j]$.) We therefore have ℓ identical experiments where each is successful with probability $p \geq \frac{mt}{2N}$, and we are interested in the probability that at least one of them is successful.

This is minimized when $p = \frac{mt}{2N}$ and from inclusion/exclusion

$$\begin{aligned}\text{Prob}[\cup_{i=1}^{\ell} V_i] &\geq \sum_{i=1}^{\ell} \text{Prob}[V_i] - \sum_{1\leq i<j\leq \ell} \text{Prob}[V_i \cap V_j] \\ &\geq \frac{\ell \cdot m \cdot t}{2N} - \binom{\ell}{2} \cdot \frac{m^2 t^2}{4N^2} \\ &\geq \frac{1}{2} \cdot \frac{\ell \cdot m \cdot t}{N} - \frac{1}{8} \cdot \frac{\ell^2 m^2 t^2}{N^2} \\ &\geq \frac{1}{4}. \qquad \square\end{aligned}$$

We now turn to the analysis of the run time of the scheme. For each cluster $1 \leq i \leq \ell$ we have to apply g_i and f at most $k/2$ times at step 2(b). In addition we must consider the complexity of step 2(a), i.e., the number of times a chain is traced. Fix $y = f(x)$. Consider stage 2(a) of the inversion procedure above: For cluster i, F_i is defined to be a random variable that counts the number of times the test "$f(C) = y$" fails. This failure is called a false alarm.

LEMMA 4.4. $E[F_i] \leq 1$.

Proof. Any y and every h_i induces a chain $u_i = g_i^*(y), h_i(u_i), h_i^2(u_i), \ldots, h_i^t(u_i)$. A false alarm at the ith cluster occurs if for some $1 \leq j \leq m$ the chain rooted at y and one of the chains rooted at x_{ij} overlap. There are m such chains, and from Claim 4.3 the probability that two chains merge is bounded by $(k/2)^2/S \leq 1/4m$. Hence the expected number of false alarms is bounded by 1. $\square$

Thus, false alarms increase the time required per cluster by at most a factor of 2. Therefore the total expected complexity of online inversion is $O(k \cdot \ell)$, and the following precursor to the main theorem holds.

THEOREM 4.5. *For any function f not trivial to invert in space S, and for any image $y = f(x)$, the probability that $f^{-1}(y)$ is found is $\Omega(1)$ and the expected number of evaluations of f and the g_i's required to invert $f(x)$ is $\tilde{O}(t \cdot \ell)$. The space $S = \tilde{O}(\ell \cdot m)$ and the preprocessing time is $\tilde{O}(N)$.* $\square$

4.4. Construction of the g_i's. We now specify how to choose the g_i's so that they can be computed efficiently. We do not know how to construct a family of k-wise independent functions so that evaluating a function in the family at a given point requires only $\tilde{O}(1)$ time. Instead we show how to construct a family so that the total computation can be amortized to $\tilde{O}(1)$ time. We change the order of operations in the basic time/space scheme. We do not process cluster by cluster but rather advance one step in every cluster simultaneously. In order to simplify the definition of the g_i's, encode their domain (which is $D \times \{1, \ldots, N\}$) as $\{1, \ldots, N^2\}$ and the range as $\{1, \ldots, N\}$. We actually construct functions $g_i : \{1, \ldots, N^2\} \mapsto \{1, \ldots, N^2\}$, but by simply chopping part of the output we can obtain $g_i : \{1, \ldots, N^2\} \mapsto \{1, \ldots, N\}$ and then consider it as an element of D.

Construct $g_1, g_2, \ldots, g_\ell$ as follows: We work in the finite field $GF[N^2]$ (N is a power of 2), and all computations are in this field. For $0 \leq j \leq k-1$ pick a_j and b_j randomly and independently in $GF[N^2]$. Encode i as an element of $GF[N^2]$ in some arbitrary manner and let g_i be the polynomial of degree $k-1$ whose jth coefficient is $a_j \cdot i + b_j$. That is,

$$g_i(x) = \sum_{j=0}^{k-1} (a_j \cdot i + b_j) x^j.$$

PROPOSITION 4.6. *A function g_i chosen in this manner is k-wise independent.*

Proof. Because each g_i is a random polynomial of degree $k-1$, the proposition follows from [3]. □

From the pairwise independence of polynomials of degree 1, and the random choice of the a and b values above, the set of the $2k$ coefficients of g_i and g_j are independent and we have the following proposition (which was used in Lemma 4.3).

PROPOSITION 4.7. *For all $1 \le i < j \le \ell$, the polynomials g_i and g_j are independent as defined in section* 4.2.

To compute $g_1(x_1), g_2(x_2), \ldots, g_\ell(x_\ell)$, note the following:

$$\begin{aligned} g_i(x) &= \sum_{j=0}^{k-1} (a_j \cdot i + b_j) x^j \\ &= i \cdot \sum_{j=0}^{k-1} a_j x^j + \sum_{j=0}^{k-1} b_j x^j. \end{aligned}$$

Define

$$A(x) = \sum_{j=0}^{k-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{k-1} b_j x^j.$$

Thus, $g_i(x) = i \cdot A(x) + B(x)$. Using FFT we can amortize the cost of computing A and B. We can compute $A(x_1), A(x_2), \ldots, A(x_\ell)$ and $B(x_1), B(x_2), \ldots, B(x_\ell)$ at a cost of $O(k \log \ell + \ell)$. (See [1, Chapter 7].) Given $A(x_1), A(x_2), \ldots, A(x_\ell)$ and $B(x_1), B(x_2), \ldots, B(x_\ell)$, computing $g_i(x_i) = i \cdot A(x_i) + B(x_i)$ for all $1 \le i \le \ell$ can be done at a cost of ℓ.

Thus, the total number of operations required to advance one function computation in all g_i is $O(k \log \ell + \ell) = O(t \log \ell + \ell)$.

LEMMA 4.8. *Computing $g_1, g_2, \ldots, g_\ell$ at points $x_1, x_2, \ldots, x_\ell$ can be done in time $O(t \log \ell + \ell)$.*

In order to achieve the time/space trade-off (3.2), given any T and S, set $t = N/S$, $m = N/T$, and $\ell = TS/N$. If $TS^3 = N^3$, then the two constraints of section 4.2, $t \le \ell$ and $mk^2 \le S$, are satisfied, and since we may assume that $S < N/8 \log N$, we also have that $t \ge 8 \log(4m)$. Therefore by Theorem 4.5 we have the desired time and space.

In order to achieve the time/space trade-off (3.1), given any T and S such that $T \cdot S^2 = N^3 q(f)$, set $t = N/S$, $m = N/T$, and $\ell = TS/N$. Note that we have $t \le \ell$, $mk^2 \le 1/q(f)$, and $k/S \le k^2 q(f)$. The conditions $mk^2 \le 1/q(f)$ and $k/S \le k^2 q(f)$ are sufficient to replace the constraint $mk^2 \le S$ in the proofs of Claims 4.1, 4.2, and 4.3. Therefore we have the following corollary.

COROLLARY 4.9. *Any function f which is not trivial to invert in space S can be inverted at any point with high probability using either time/space trade-offs:*

$$\begin{aligned} &T \cdot S^2 = N^3 \cdot q(f) \text{ or} \\ &T \cdot S^3 = N^3. \end{aligned}$$

5. Extensions. For simplicity we have assumed in section 4 that the function f is not trivial to invert in space S. To complete the proof of Theorem 1.1 we must also deal with this case.

If the S highest indegree images cover a substantially large fraction of the domain, then our scheme may suffer degradation in performance since we cannot claim that when evaluating $h_i(x)$ the expected number of evaluations of f and g_i is $O(1)$. On the other hand, in this case table A covers a larger fraction of images and thus the effective domain size is smaller. We will see how to exploit this fact.

Suppose that following the first stage of the preprocessing phase it turns out that the number of elements in $D' = \{x \in D | f(x) \notin A\}$ is $N' < N/2$ (i.e., the function is trivial to invert in space S). Since all $x \notin D'$ are handled by A, we need to construct a scheme that covers only D', thus raising the possibility of improving only the complexity. Our problem is that we do not know how to construct g_i's that map into D' and not D, so we may need more applications of the g_i's in order to assure that we land in D'. For S and T set $m = N/T$ and $\ell = TS/N$ as before, but set $t = N'/S$ and $k = 8N/N' \cdot t = 8N/S$. Note that $t \cdot \ell \cdot m = N'$ and $mk^2 \leq S$ and given that $TS^3 \geq N^3$, then $k \leq \ell$ (this last point is significant for the amortization of computing the g_i's).

Therefore almost all the analysis of sections 4.3 and 4.4 is applicable and in particular the space is $\tilde{O}(S)$ and the time is $\tilde{O}(T)$. The only point we should change is in the proof of Claim 4.1. We must recompute the probability that a chain rooted at x_{ij} is discarded since g_i was invoked more than $k/2$ times. However, this means that in order to find t members of D' more than $k/2 = 4 \cdot N/N' \cdot t$ trials were needed, where in each time the probability of hitting a member of D' is N'/N. The probability that this occurs is at most the probability that a sequence of $k/2$ tosses of a coin contains at least $k/2 - t$ tails (i.e., the event g_i maps to a value x such that $f(x) \in A$) and less than t heads (the event g_i maps to a value x such that $f(x) \notin A$) with probability $p = N'/N$ for heads (by the assumption that $|D'| = N'$). By the Chernoff bound (see Theorem A.13 of [4]) this value is at most $e^{-9t^2/4t}$, which is smaller than $1/m$. This concludes the proof of Theorem 1.1.

Finally, we note that $q(f)$ need not be given explicitly in order to choose the best time/space trade-off. We can consider an additional "proposal" stage in which a function f is studied to determine what time/space trade-offs can be used to invert it. By estimating $q(f)$ we can check which trade-off to use: if $S > 1/q(f)$, we use the trade-off $T \cdot S^3 = N^3$, otherwise we use $T \cdot S^2 = N^3 \cdot q(f)$. Note that estimating the ith bit of $q(f)$ requires 2^i time. We can obtain a slightly pessimistic trade-off by taking a slightly higher value for $q(f)$.

The "proposal" stage should also consider how $q(h_i)$ drops as a function of S. We can estimate $q(h_i)$ by estimating the probability that two elements collide under f given that they *do not* map to any of the S images with largest indegrees. Even if $q(f)$ is big, $q(h_i)$ might drop very quickly as a function of S, becoming much smaller than $1/S$.

6. Open problems. The most challenging problem in this area of investigation is whether the time/space trade-off for inverting functions can be reduced to $T \cdot S = N$. For permutations this is true and Yao [20] has shown that this is tight for permutations. Shamir and Spencer [18] consider schemes of the basic Hellman form, where functions are represented as a collection of chains indicated by their start and end locations, and show that $TS^2 = N^2$ is optimal for these algorithms.

Acknowledgments. We thank Noga Alon and Uri Feige for fruitful discussions. We thank the diligent referees for their many helpful comments.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[2] A. Aho and D. Lee, *Storing a dynamic sparse table*, in Proceedings 27th IEEE Symposium on Foundations of Computer Science, Chicago, 1986, pp. 55–60.

[3] N. Alon, L. Babai, and A. Itai, *A fast and simple randomized algorithm for the maximal independent set problem*, J. Algorithms, 7 (1987), pp. 567–583.

[4] N. Alon and J. Spencer, *The Probabilistic Method*, John Wiley, New York, 1992.

[5] H. R. Amirazizi and M. E. Hellman, *Time-memory-processor trade-off*, IEEE Trans. Inform. Theory, 34 (1988), pp. 505–512.

[6] J. L. Carter and M. N. Wegman, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[7] B. Chor and O. Goldreich, *On the power of two-point based sampling*, J. Complexity, 5 (1989), pp. 96–106.

[8] B. Chor, O. Goldreich, J. Hastad, J. Friedman, S. Rudich, and R. Smolensky, *The bit extraction problem or t-resilient functions*, in Proceedings 26th IEEE Symposium on Foundations of Computer Science, Portland, OR, 1985, pp. 396–407.

[9] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[10] A. Fiat, S. Moses, A. Shamir, I. Shimshoni, and G. Tardos, *Planning and learning in permutation groups*, in Proceedings 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 274–279.

[11] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. ACM, 31 (1984), pp. 538–544.

[12] M. E. Hellman, *A cryptanalytic time memory trade-off*, IEEE Trans. Inform. Theory, 26 (1980), pp. 401–406.

[13] M. E. Hellman and J. M. Reyneri, *Drainage and the DES, summary*, in Advances in Cryptology–Proceedings of Crypto '82, Plenum Press, New York, 1982, pp. 129–131.

[14] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.

[15] R. C. Merkle and M. E. Hellman, *Hiding information and signatures in trapdoor functions*, IEEE Trans. Inform. Theory, 24 (1978), pp. 525–530.

[16] S. C. Pohlig and M. E. Hellman, *An improved algorithm for computing logarithms over $GF[p]$ and its cryptographic significance*, IEEE Trans. Inform. Theory, 24 (1978), pp. 106–110.

[17] R. SchroepPel and A. Shamir, *A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems*, SIAM J. Comput., 10 (1981), pp. 456–464.

[18] A. Shamir and J. Spencer, manuscript.

[19] A. Siegel, *On universal classes of fast high performance hash functions, their time-space tradeoff and their applications*, in Proceedings 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 20–25.

[20] A. C. Yao, *Coherent functions and program checkers*, in Proc. 22nd ACM Symposium on Theory of Computing, New York, 1990, pp. 84–94.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 804–833

BUBBLES: ADAPTIVE ROUTING SCHEME FOR HIGH-SPEED DYNAMIC NETWORKS*

SHLOMI DOLEV†, EVANGELOS KRANAKIS‡, DANNY KRIZANC‡, AND DAVID PELEG§

Abstract.
This paper presents the first dynamic routing scheme for high-speed networks. The scheme is based on a hierarchical *bubbles* partition of the underlying communication graph. Dynamic routing schemes are ranked by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. An advantage of our scheme is that it implies a small number of updates upon a topology change. In particular, for the case of a bounded degree network it is proved that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Our bubble routing scheme is a combination of a distributed routing database, a routing strategy, and a routing database update. It is shown how to perform the routing database update on a dynamic network in a distributed manner.

Key words. adaptability, dynamic, fiber optics, communication network, routing

AMS subject classification. 68Q99

PII. S0097539797316610

1. Introduction.

1.1. Motivation.

The advent of fiber-optic technology dramatically changes the characteristics of distributed networks. It also improves the capabilities of distributed networks because it gives them the potential of supporting new services such as multimedia and real-time applications. Traditional algorithms designed for the point-to-point classical model of distributed networks may neither fit the new characteristics of the high-speed network nor support the new tasks it is capable of achieving. The relation between the bandwidth of a fiber-optic cable (on the order of hundreds of Gigabits per second) and the speed of a processor implies a bottleneck in the process time rather than in the communication time. Therefore, high-speed networks use a fast switching subsystem in order to utilize the power of the fiber-optic cables.

Algorithms for basic tasks that match the new network structure are of interest. In (high-speed) distributed networks, messages are used for communication between different sites. A message sent from one site to another is transferred through the network according to a *routing scheme*. The routing scheme ensures that the message

*Received by the editors February 14, 1997; accepted for publication (in revised form) January 20, 1998; published electronically December 15, 1999. An extended abstract of this work was presented at the Twenty-Seventh Annual ACM Symposium on the Theory of Computing, May 1995.
http://www.siam.org/journals/sicomp/29-3/31661.html

†Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel (dolev@cs.bgu.ac.il). The research of this author was performed at the School of Computer Science, Carleton University, and the Department of Computer Science, Texas A&M University, College Station, TX. Part of the research of this author was supported by NSERC (Natural Sciences and Engineering Research Council of Canada), TAMU Engineering Excellence funds, and NSF Presidential Young Investigator Award CCR-9396098.

‡School of Computer Science, Carleton University, Ottawa, ON, K1S 5B6, Canada (kranakis@scs.carleton.ca, krizanc@scs.carleton.ca). Part of the research of these authors was supported by NSERC.

§Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel (peleg@wisdom.weizmann.ac.il). The research of this author was supported in part by a grant from the Israel Science Foundation.

is forwarded toward its destination. The routing scheme serves the basic communication primitive in the network—message delivery. Being such a basic component, the performance of the distributed network as a whole may be dominated by the quality of the routing scheme. Thus, finding an efficient routing scheme is one of the most important tasks in dealing with distributed networks.

Imagine a network in which users may be connected and disconnected upon request. Users may migrate from one geographic region to another, causing a change in the demand for services at different parts of the network. Assume further that the network spans the entire world and a single user (or a network junction) changes its location from one street in New York to another. Is it reasonable to update the *entire* network with a new routing database? We would not like the *entire* distributed network to be updated upon each such dynamic change. In fact we would like to *minimize* the effect of a topology change as much as possible.

Beyond planned topology changes, such as user migration, some transient topology changes may take place due to a failure of communication links or processors. One would like the network to automatically change the routing database to reflect the new topology upon the change. The resources used by such a distributed routing database update (messages and time) have an inherent relation with the number of sites that have to change their portion of the distributed routing database.

We distinguish between *static* and *dynamic* routing schemes. A *static* routing scheme is a combination of a distributed routing database and a fitting routing strategy. The routing database is tailored to the network topology. Whenever the network changes its topology, a new distributed routing database is assigned to the network, possibly changing the routing database portion of each processor. A *dynamic* routing scheme has in addition a fitting routing database update. Upon a topology change (e.g., link addition or removal) this fitting routing database update would change the distributed database only in a *limited* number of sites. We rank the dynamic routing schemes by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. By this definition static routing schemes are associated with adaptability that is in the order of the number of nodes in the network.

The efficiency of a dynamic routing scheme is measured not only by its adaptability. It is also measured by the time and memory complexities associated with it. The time performance is measured by a *super-hop count*, defined as the maximum number of super-hops, or routing steps, needed to route a message from a given source to a given destination, where the maximum is taken over all possible origin–destination pairs. A more precise definition of the routing process in high-speed networks and the notion of super-hops will be given later. The memory complexity is the total number of bits used for the routing database. We remark that there is an interesting relation between the adaptability and the memory requirement; i.e., it is clear that the worst case in terms of adaptability and memory requirement is when the entire topology is stored in the memory of each processor. Roughly speaking, both the adaptability and the memory requirements gain when the processors have less information on the system.

1.2. Previous work. Many clever routing schemes and lower bounds for the resources required for routing in point-to-point networks (e.g., wide area networks) were presented in the past. Following the pioneering work of [13, 14], which originated the approach of using hierarchical clustering [2] strategies for memory-efficient routing, came a number of papers which attempted to characterize and bound the resource tradeoffs involved. The first set of studies along this line were mostly designed for special classes of networks like trees [16], complete networks [18], and grids [19]. Routing schemes for general networks were presented in [15, 4, 6]. These studies focused on the design of routing schemes with compact routing tables and low stretch factors. The *stretch factor* of a routing scheme is defined as the maximum ratio, over all pairs of nodes in the network, between the length of the route provided for them by the routing scheme, and the actual distance between them in the network.

Unfortunately, most of the success in this field is for static networks. Few papers consider the dynamic property of the network. The following quotation is taken from [16]: “In actual network the topology may vary in time; in particular, nodes or links may be added or deleted.” In [16] a partial solution is present for limited cases of topology changes that keep the network in a tree structure. In [7], the routing scheme of [6] is extended to the dynamic case for general networks. However, as argued in [1], network changes in static routing schemes (such as [15, 3, 6] or the derived dynamic scheme [7]) “require expensive pre-processing to reconstruct the routing scheme over the whole network. The newly constructed structure is used until the next change....” In contrast, [1] contains a routing scheme for the restricted case of dynamic growing trees. The solution of [1] can handle neither link nor processor failures, nor can it be applied to the case of general graphs.

These papers all deal with the traditional point-to-point communication model. Those solutions are not applicable for fiber-optic high-speed networks, since the stretch measure (based on actual distances) is no longer the predominant cost parameter relevant to such networks, given their specific characteristics. Furthermore, the issue of adaptability is not handled by any of the aforementioned papers. In particular, the solutions provided in the these papers for the dynamic version of the problem might require database updates in all the nodes of the network following a single topology change.

Recently, static routing schemes for high-speed networks were presented in [12, 10]. The schemes statically assign links along a path to act as a virtual long link. Both papers attempt to simultaneously optimize three cost parameters, namely, the super-hop count, the memory size, and the link load. However, in [12, 10] adaptability is not addressed explicitly, and in the resulting solutions a single topology change may in some cases require the entire routing database to be updated. Hence the solution might be impractical in a dynamically changing environment.

1.3. Contributions of this paper. In this work we present the first dynamic routing scheme for high-speed networks. We present a family of hierarchical *bubbles* schemes. The intuition behind this structure is the behavior of natural bubbles. Assume a partition of a space into bubbles. Whereas bubbles may shrink (members are disconnected) or expand and/or blow up (new members are inserted), we want to avoid total change of the bubble structure upon a change at a single bubble. We define an upper and lower threshold for the size of a bubble. When the upper threshold is reached, the bubble is split into two bubbles. When the lower threshold is reached, the bubble is combined with a single neighboring bubble which “swallows” the small bubble (and then is split if necessary).

The bubbles scheme exhibits a tradeoff between the number of super-hops needed for routing on the one hand, and the adaptability and memory requirement parameters on the other hand. Namely, the more super-hops are allowed, the less memory is needed to store the routing database and the more efficient it becomes to adapt the scheme to dynamic changes. Denote by δ the maximum number of neighboring nodes with which each node in the network may be directly connected. For the case of

	Super-hop count	Memory required	Adaptability	Model
Multiple trees	1	$O(n^2 \log n)$	n	
Single leader	2	$O(n^2 \log \delta)$	n	
Basic bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta^2 n^{1/k})$	Node failures
Basic bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Link failures
Edge-bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Node failures
Edge-bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k n^{1/k})$	Link failures

FIG. 1. *Comparison of routing schemes.*

constant δ we prove that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Note that this is the case in many settings in reality, where the number of communication ports of a single processor is limited. Our bubble routing scheme is a combination of a distributed routing database, a routing strategy, and a routing database update. We present a distributed routing database update strategy for dynamic changing networks.

The rest of the paper is organized as follows. The definition of the problem and two examples for basic routing schemes appear in section 2. These two schemes are given as examples for our complexity measures and a basis for comparison with the bubbles scheme, which is presented in section 4. Section 5 presents an improved scheme, based on a somewhat more involved bubble partitioning algorithm. See Figure 1 for a summary of the comparison among the different schemes. Note that we typically choose k to be a small constant, with a value much smaller than $\log n$. Section 6 presents a lower bound on the adaptability of any routing scheme. The distributed update of the bubbles routing database appears in section 7. Concluding remarks are given in section 8.

2. Definition of the problem.

2.1. High-speed dynamic network. We consider the high-speed model of a communication network as described in [8, 5, 9]. The network is described by an undirected graph $G = (V, E)$. The nodes, $V = \{1, \ldots, n\}$, represent the processors of the network (where n is essentially an upper bound on the number of processors in a connected component). The edges of the graph represent bidirectional communication channels between the processors.

The network is *dynamic* in the strong sense: processors and links may crash and recover arbitrarily. However, the number of edges connected to a node P, which we call the *degree* of P, does not exceed some predefined value. We denote the maximum degree of nodes in the network by δ.

Each processor consists of two components, a *switching subsystem* and a *node control unit*, which are connected by a virtual link. The *switching subsystem* is a fast and simple hardware device that switches arriving packets to the appropriate communication link or to its node control unit. Within the switching subsystem each communication link (including the virtual link to the switching subsystems' node control unit) has a unique label. When a packet p arrives at a switching subsystem and the header of p contains at least one label, then the switching subsystem removes the first label, l, and the shortened packet is sent on the link with label l. The *node control unit* of each processor contains the processing hardware and software necessary to extract the information content of messages (delivered in the packets), do internal computation, and generate packets to be forwarded to other nodes via the processor's switching subsystem.

In this model a packet consists of the entire path from the sender to the receiver. For example, when the packet length is 50 bytes and the degree of nodes in the network is $\delta = 5$, then a packet may contain 5^{50} different addreses. This is more than enough for every existing network. Moreover, note that the first packet in a connection may mark the path (by the connection identifier) for the next packets that are related to the same connection—thus not every packet contains the entire path. Due to the above network architecture, it is assumed that a message sent from any processor, P, to any destination, Q, in the network may arrive in one time unit provided the labels along the entire path from P to Q are known to P. We refer to such a routing step as a *super-hop*.[1] In case the entire path is not known to P, the message may be sent through a number of super-hops. In each super-hop, the sender packs the message in a packet whose header contains the route description to the next intermediate destination, and advancing the message to that destination. Since the overhead of preparing the packets dominates the transmission time, the cost of the entire routing process is proportional to the number of super-hops it involves.

2.2. Complexity measures for routing schemes. The routing of packets in the network is done according to a *distributed routing database* maintained by the node control unit and a fitting *routing strategy* and *database update*.

Super-hop count. The *super-hop count* of a routing scheme bounds the maximum number of super-hops (or, packet generation and transmission steps) required in order to send a message from one node control unit to another. Obviously, a single packet transmission is sufficient when every processor knows the entire topology (including the link labels). Thus, the super-hop count can be thought of as measuring the ratio between the maximum number of packets *generated* by the routing scheme and the optimal number of packets that must be generated in order to deliver a message sent from one node control unit to another. It is interesting to note that in that sense, the super-hop count of a routing scheme in the high-speed model essentially plays a similar role to that played by the *stretch factor* measure in routing schemes for traditional computer networks (cf. [15, 3, 6]).

Memory. The memory complexity is the total number of bits maintained by the node control units for the routing data-structure.

Adaptability. A single topology change $\mathcal{C}$ occurs when a single processor or a single link joins (or recovers) or leaves (or crashes) the system. Note that for any two topologies G_1 and G_2 there exist a finite sequence of single topology changes $\mathcal{C}_1, \mathcal{C}_2, \ldots$ that transfer G_1 into G_2. For example, the sequence can start with adding all the processors that do not appear in G_1 but do appear in G_2, one processor at a time. Then links are added in a similar fashion. Finally, links and then processors are removed to form G_2. Ideally a topology change causes only a very limited number of processors to change their portion of the routing distributed database. Thus, we choose the *adaptability measure* to be the maximum number of processors that have to change their portion of the routing distributed database upon a single topology change. A similar measure, called the *adjustment measure*, is proposed in [11].

We shall first ignore the issue of *how* the routing database is updated upon a topology change and count only the number of processors that have to change their routing

[1]Certain routing models devised for asynchronous transfer mode networks use more elaborate mechanisms and distinguish between several types of "super-hops," e.g., routing along virtual channels or virtual paths. Here, we follow the simple, unified model of automatic network routing (ANR) of [8, 9].

database. Then we present a distributed update for the bubble routing scheme, which we call *bubbles update*.

Distributed adaptability. This is the maximal number of node control units that have to participate in the distributed routing database update upon a single topology change. Clearly, the distributed adaptability is greater than the adaptability since every node control unit that has to change its portion of the routing distributed database participates in the distributed database update, and in addition, other node control units may participate in the distributed update without changing their routing database portion.

3. Examples of basic routing schemes.

In this section we present two basic routing schemes with adaptability n.

3.1. The multiple spanning trees scheme.

The simplest routing scheme for our network is the *multiple spanning trees* routing scheme.

Routing distributed database. The routing distributed database is a description of a spanning tree of the entire topology at each processor.

Routing strategy. The routing strategy is to use the entire path given by the routing distributed database.

Routing update. The routing update has to change the distributed routing database upon every processor addition and removal as well as upon removal of a link used as part of a spanning tree by some processor. The update would change the spanning trees representation in an obvious way.

The following lemma states the super-hop count, memory, and adaptability properties of the multiple trees routing scheme.

LEMMA 3.1. *The multiple trees routing scheme has the following properties:*

1. *super-hop count* $= 1$,
2. *memory requirement* $= n^2(\log n + \log \delta)$,
3. *adaptability* $= n$.

Proof. The super-hop count of this scheme is clearly 1, since a single packet is generated for the delivery of a message.

Property 2 holds since $n(\log n + \log \delta)$ bits are required in order to describe a spanning tree with link labels for every processor (say, in parenthesized form).

Property 3 follows from the fact that a single recovery of a processor or a link requires the update of the tree of every processor. □

3.2. The single leader scheme.

Routing distributed database. Only a single processor, L, has a spanning tree of the entire topology. Every other processor, P, has the path description to L, $path_L$. Thus, $path_L$ is a list of link labels that defines a path from P to L.

Routing strategy. To deliver a message m to a processor Q, a packet $(path_L, Q, m)$ is sent to the leader L, and then L sends a packet $(path_Q, m)$ to Q.

Routing update. The routing update has to change the distributed routing database upon leader failure, processors' addition, processors' removal, failure of a link along a path used by some processor to reach the leader, and addition of a link that connects two connected components. For a given choice of a single leader, the update is defined in a natural way.

The following lemma states the super-hop count, memory requirements, and adaptability properties of the single leader routing scheme.

LEMMA 3.2. *The single leader routing scheme has the following properties:*

1. *super-hop count* $= 2$,

2. *memory requirement* $= n(\log n + \log \delta) + n(n-1)\log \delta$,
3. *adaptability* $= n$.

Proof. The super-hop count is 2 since at most two packets are generated for each message.

The memory requirements of the scheme are as follows. The description of a spanning tree of the entire topology requires $n(\log n + \log \delta)$ bits, as in the proof of Lemma 3.1. The description of a path from a processor P to L includes at most n link labels. Each link label requires $\log \delta$ bits. Thus, each processor but L uses at most $n \log \delta$ bits for the routing database.

Finally, if all the paths to the leader L, as well as the spanning tree stored by L, include a certain link e (say, adjacent to L), then the failure of e (while preserving the connectivity of the communication graph) is a topology change that will cause every processor to change its topology database; hence the adaptability is n. $\square$

4. The bubble routing scheme.

In this section we present our main result, which is a novel dynamic graph partitioning technique, used in conjunction with a new routing scheme for dynamic high-speed networks. We introduce a new type of hierarchical partitioning scheme for general graphs, with certain desirable properties. We then present a graph partitioning algorithm that constructs such a partition. Then we explain how to maintain the properties of the partition (most notably, the size bounds on connected components) upon a dynamic change. Finally, we present our routing scheme, which is based on the hierarchical partition.

The partitioning scheme uses a spanning tree of the communication graph. The partition of the spanning tree into connected components called bubbles is done in levels. The first level is a single bubble that includes $x_1 = n$ nodes. The second level partitions the single bubble into bubbles (i.e., connected components) of at least x_2 (where $x_2 < x_1/\delta$) nodes and no more than δx_2 nodes. In general, the bubble partition of level i partitions the bubble of level $i-1$ into bubbles of at least x_i nodes and no more than δx_i nodes. A failure of a link that belongs to the spanning tree may disconnect a number of bubbles on several levels. We present a scheme to maintain the bubble partition with few changes as long as the communication graph is still connected. Assume that a bubble of level i is partitioned because of a link failure. One or two of the resulting bubble parts may contain fewer than x_i nodes. In such a case each part is combined with a neighboring bubble. Each of the two new combined bubbles may include more than δx_i nodes and therefore require a split. This fact and the fact that the partition on level $i+1$ is a refinement of the partition on level i imply that (at most) three edges should be removed from level $i+1$, namely, the failed link of level i and the two edges that are removed from bubbles of level i because of the split operations. The details follow.

4.1. The bubble partition.

DEFINITION 4.1. *Given a graph G and parameters $0 < a < b$, an $[a, b]$-*bubble partition *of G is a partition of the nodes of G into disjoint* connected *components called* bubbles, *$\mathcal{P} = \{\mathcal{B}^1, \ldots, \mathcal{B}^r\}$, such that the size of each bubble $\mathcal{B}$ (i.e., the number of nodes it contains) satisfies $a \le |\mathcal{B}| \le b$.*

Let us next describe a partitioning algorithm based on a technique presented in [17]. Given a n-node graph G with maximum degree δ and a parameter $1 \le x \le n$, the algorithm produces an $[x, \delta x]$-bubble partition for G. To be more precise, as a preprocessing step we choose an arbitrary node of the graph, R, and construct a spanning tree, $\mathcal{T}_R$, rooted at R with edges directed towards the leaves. The partitioning

```
(1) While $\mathcal{T}$ contains more than $\delta x$ nodes do:
        (a) Mark every node $P$ in $\mathcal{T}$ by $M_P$, the total number of nodes in the
            subtree $\mathcal{T}_P$ rooted at $P$.
        (b) Find a node $P$ with the property that $M_P \geq x$, but all of its children
            $Q$ satisfy $M_Q < x$.
        (c) Make $\mathcal{T}_P$ into a bubble.
        (d) Remove this bubble and the edge connecting $P$ to its parent from $\mathcal{T}$.
        End_while
(2) Make $\mathcal{T}$ into a bubble and terminate.
```

FIG. 2. Algorithm PARTITION($\mathcal{T}, x$).

algorithm is then applied to this tree. The algorithm, PARTITION($\mathcal{T}, x$), is described in Figure 2.

LEMMA 4.1. *Algorithm* PARTITION($\mathcal{T}_R, x$) *produces an* $[x, \delta x]$*-bubble partition of* G.

Proof. Obviously, each bubble constructed in step (1c) is of size at least x and no larger than δx.

We now show that every execution of the loop in step (1) succeeds in finding a candidate node P as required in (1b). Since step (1) is performed only as long as $M_R > \delta x$, R itself trivially satisfies the first requirement, namely, $M_R > x$. It now suffices to show that every tree $\mathcal{T}_Q$ of size $M_Q \geq x$ has a node P as required. This is proved by simple induction on the size of the tree, beginning at size x. For the inductive step we note that if Q has no child with more than x nodes in its subtree $\mathcal{T}_Q$, then Q can be taken as the required P; otherwise, let S be the child of Q with $M_S \geq x$ and apply the inductive hypothesis to $\mathcal{T}_S$, to find a suitable node P in it.

The fact that any execution of the loop in step (1) results in a new bubble and that $\mathcal{T}_R$ is finite implies the termination of the algorithm.

Finally, we show that when the algorithm terminates, $x \leq M_R \leq \delta x$; hence the last bubble is legal as well. The termination condition of the loop implies that when it terminates, $M_R \leq \delta x$. If the number of nodes in the entire graph G is also smaller than δx, then this is the only bubble, and we are done (since $x \leq n$ by choice of x). Otherwise, there exists at least one more bubble. Examine the last removal executed at step (1d) before the termination of the algorithm. Right before this removal, M_R is at least $\delta x + 1$, and as a result of the removal, no more than $(\delta - 1)(x - 1) + 1$ nodes are removed from $\mathcal{T}_R$, leaving at least $\delta x + 1 - (\delta - 1)x + (\delta - 1) - 1 = x + \delta - 1 \geq x$ nodes that are still connected to R. ∎

Theorem 4.2 is implied by the existence of the partition algorithm.

THEOREM 4.2. *For every n-node graph G with maximum degree δ and parameter* $1 \leq x \leq n$, *G has an* $[x, \delta x]$*-bubble partition.* ∎

Next we define special hierarchies of bubble partitions. Given two partitions $\mathcal{P}$ and $\mathcal{P}'$, we say that $\mathcal{P}$ is a *refinement* of $\mathcal{P}'$ if each bubble of $\mathcal{P}$ is fully contained in some bubble of $\mathcal{P}'$ (or equivalently, each bubble of $\mathcal{P}'$ is partitioned into bubbles of $\mathcal{P}$).

DEFINITION 4.2. *Given a graph G and a list of k integer parameters* $\bar{x} = (x_1, x_2, \ldots, x_k, x_{k+1})$, *where* $x_1 = n$, $x_i > \delta x_{i+1}$, *and* $x_{k+1} = 1$, *an* $\bar{x}$-hierarchical bubbles partition *of G is a collection of k partitions $\mathcal{P}_i$, $1 \leq i \leq k$, with the following properties:*

1. *For every* $1 \leq i \leq k$, $\mathcal{P}_i$ *is an* $[x_i, \delta x_i]$*-bubble partition.*
2. *For every* $1 < i \leq k$, $\mathcal{P}_i$ *is a* refinement *of* $\mathcal{P}_{i-1}$.

```
(1) Choose an arbitrary node of the graph, $R$.
(2) Construct a spanning tree $\mathcal{T}_R$ rooted at $R$ with edges directed towards the
        leaves.
(3) Let the first level partition $\mathcal{P}_1$ consist of a single bubble covering the entire
        graph.
(4) For $i = 2$ to $k$ do:
        For every bubble $\mathcal{B}$ of partition $\mathcal{P}_{i-1}$ do:
        (a) Let $\mathcal{T}(\mathcal{B})$ be the portion of $\mathcal{T}_R$ that spans the bubble $\mathcal{B}$.
        (b) Apply Algorithm PARTITION$(\mathcal{T}(\mathcal{B}), x_i)$ to $\mathcal{T}(\mathcal{B})$.
        (c) Add the resulting bubbles to the current level partition $\mathcal{P}_i$.
```

FIG. 3. Algorithm HIERARCHY$(\bar{x})$.

In what follows, we choose the parameters $\bar{x}$ as $x_i = \lceil n^{(k-i+1)/k} \rceil$. This choice satisfies the requirements of the definition, so long as $n^{1/k} \geq \delta$. We use $\mathcal{B}_i$ to denote a bubble at level i. For every bubble $\mathcal{B}_i$ let $\mathcal{T}(\mathcal{B}_i)$ be a representation of the subtree of the spanning tree $\mathcal{T}$ that spans the bubble $\mathcal{B}_i$. (Hereafter we refer to both the description of the spanning tree and the spanning tree itself as $\mathcal{T}(\mathcal{B}_i)$.) Thus, each of the k partitions $\mathcal{P}_i$, $1 \leq i \leq k$, defines a *forest* $\mathcal{F}_i = \bigcup_{\mathcal{B} \in \mathcal{P}_i} \mathcal{T}(\mathcal{B})$. Note that $\mathcal{F}_1$ is identical to the spanning tree $\mathcal{T}$.

We note the following properties of the hierarchical partition. Since each bubble of level i is composed of the union of some bubbles of level $i+1$, the spanning tree $\mathcal{T}(\mathcal{B}_i)$ is a combination of the spanning trees of those bubbles. Hence $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ for every $1 \leq i \leq k-1$, and $\mathcal{F}_1$, corresponding to the single bubble containing the whole network, is the entire tree $\mathcal{T}$.[2] These properties will be maintained as invariants of the hierarchical partition throughout the updates performed upon topology changes and are important for the analysis of the scheme, as will be shown later on.

The hierarchical bubble partition of a graph is constructed level by level by Algorithm HIERARCHY$(\bar{x})$, described in Figure 3.

It is clear that $\mathcal{P}_1$ is an $[n, \delta n]$-bubbles partition. Since for every $i > 1$ the partition $\mathcal{P}_i$ is obtained from $\mathcal{P}_{i-1}$ by splitting each bubble $\mathcal{B}$ separately, using Algorithm PARTITION$(\mathcal{T}(\mathcal{B}), x_i)$, the requirements of Definition 4.2 are clearly met, establishing the following theorem.

THEOREM 4.3. *Every graph has an $\bar{x}$-hierarchical bubbles partition.* □

Figure 4 depicts a partition of a graph into bubbles. First, a spanning tree is constructed (the upper part of the figure). Then the spanning tree is partitioned into bubbles of levels 2 and 3 (lower parts of the figure).

4.2. Dynamic maintenance of the partition. In order to use the bubble partition over time in a dynamically changing environment, it is necessary to be able to maintain a legal partition in the presence of link and node failures and recoveries. One point that we need to address first when dealing with the dynamic case is that of failures that disconnect the network. Our policy in such a case is to treat each connected component separately but still to use the global parameters, $\bar{x}$. This has the implication that certain connected components cannot be partitioned in some of the lower levels of the hierarchy, since they are too small. In particular, if a certain connected component G' of the graph has less than x_i (but at least x_{i+1}) nodes, then on levels 1 through i there is a single bubble encompassing the entire connected

[2]We say that $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ when every tree in $\mathcal{F}_{i+1}$ is a subtree in $\mathcal{F}_i$.

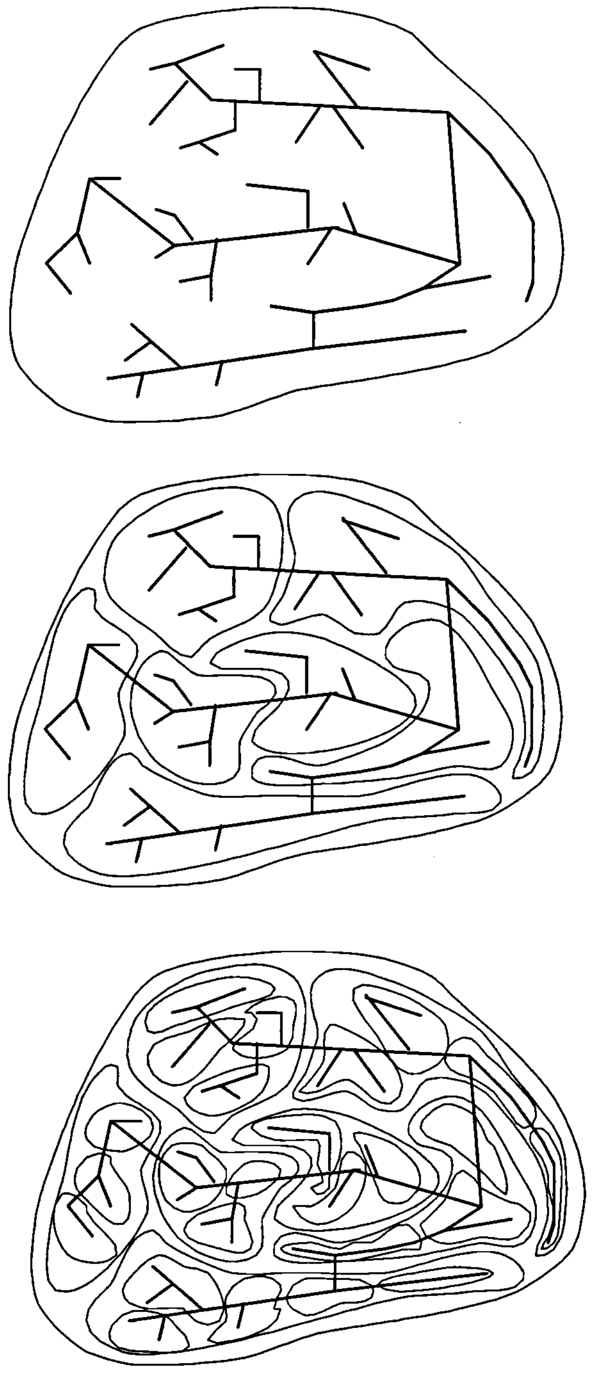


FIG. 4. *Bubble partition $k = 3$.*

```
Input:   Two tree-neighboring level $l$ bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$.
Action:
    Connect $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$ using their connecting link.
Output:   A combined bubble $\mathcal{B}_l$.
```

FIG. 5. Procedure COMBINE($\mathcal{B}'_l, \mathcal{B}''_l$).

```
Input:   A (typically oversized) bubble $\mathcal{B}_l$.
Action:
    Select one of the nodes of $\mathcal{B}_l$ to be the root $R$.
    Direct $\mathcal{T}(\mathcal{B}_l)$ from $R$ towards the leaves.
    Execute Algorithm PARTITION($\mathcal{T}(\mathcal{B}_l), x_l$).
Output:   An $[x_l, \delta x_l]$-bubbles partition of $\mathcal{B}_l$.
```

FIG. 6. Procedure SPLIT($\mathcal{B}_l$).

component. This bubble is said to be *illegal*, since it is smaller than the allowed lower bound. The implication of such a situation is that once a connecting link recovers, it is necessary to reorganize the partition in order to "legalize" the bubble structure.

We now present the strategy for handling each topology change. For now, the strategy we describe is centralized, as if an outside operator changes the distributed routing database. Later, in section 7, we discuss a distributed implementation of our maintenance strategy. We use the following definition.

DEFINITION 4.3. *Two bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$, both at level $l > 1$, are said to be* tree-neighboring *with respect to (w.r.t.) the level $l - 1$ forest $\mathcal{F}_{l-1}$ if there exists an edge of $\mathcal{F}_{l-1}$ that connects them. We refer to this tree link as the* connecting link *of $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$.*

Our maintenance activities are based on employing two basic operations, or "Procedures," COMBINE($\mathcal{B}'_l, \mathcal{B}''_l$) and SPLIT($\mathcal{B}_l$), described in Figures 5 and 6, responsible for combining and splitting bubbles, respectively. These two basic procedures are used as building blocks for our update operations.

Let us next describe another procedure, named Procedure REMOVE($e, \mathcal{F}_l, \mathcal{F}_{l-1}$), that handles the removal of a link e from the forest of spanning trees of the level $l > 1$ bubbles partition, $\mathcal{F}_l$. This removal can be the result of either the link's failure, or the restructuring of bubbles at the next lower level, $l - 1$, that caused the removal of this link from $\mathcal{F}_{l-1}$. In any case, it is assumed that e does not appear in $\mathcal{F}_{l-1}$. The computation of the new forest $\mathcal{F}'_l$, that no longer contain e, is based on the current forest for level $l-1$, $\mathcal{F}_{l-1}$, in the sense that the only edges that are added to $\mathcal{F}'_l$ during the procedure's execution are also edges of $\mathcal{F}_{l-1}$. Procedure REMOVE($e, \mathcal{F}_l, \mathcal{F}_{l-1}$), described next in Figure 7, relies on the assumption that both $\mathcal{F}_{l-1}$ and $\mathcal{F}_l$ span legal bubbles partitions (of level $l - 1$ and l, respectively). However, it is not assumed that $\mathcal{F}_l$ is a refinement of $\mathcal{F}_{l-1}$.

It is important to understand that Procedure REMOVE ends up not only removing the specified link e from $\mathcal{F}_l$ but also removing some *additional* links, as well as adding some links to $\mathcal{F}_l$, in its attempt to regain the legality of the partition. In particular,

```
Input:   Edge $e$ for removal, forests $\mathcal{F}_l$ and $\mathcal{F}_{l-1}$ spanning legal partitions.
Action:
  The removal of $e$ causes the partition of a single bubble, say $\mathcal{B}_l$, into two portions,
  denoted $\mathcal{B}_l'$ and $\mathcal{B}_l''$.
  For each of these two portions $\hat{\mathcal{B}}$ do:
      1. If the size of $\hat{\mathcal{B}}$ is still greater than $x_l$, then make it into an independent
         bubble without change.
      2. Otherwise, if it is too small, then do the following:
          (a) Find some bubble $\tilde{\mathcal{B}}$ of $\mathcal{F}_l$ that is a tree-neighbor of $\hat{\mathcal{B}}$ w.r.t. $\mathcal{F}_{l-1}$.
          (b) Merge $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$ into $\bar{\mathcal{B}}$ by invoking procedure Combine($\tilde{\mathcal{B}}, \hat{\mathcal{B}}$).
          (c) If $\bar{\mathcal{B}}$ includes more than $\delta x_l$ nodes, then split it by invoking proce-
              dure Split($\bar{\mathcal{B}}$).
Output:   Forest $\mathcal{F}_l'$.
```

Fig. 7. Procedure Remove$(e, \mathcal{F}_l, \mathcal{F}_{l-1})$.

the following scenario may take place during the operation of removing e on level l. Suppose the removal of e breaks bubble $\mathcal{B}_l$ into two portions. We then apply Combine operations to the broken portions, combining them with tree-neighboring level l bubbles. The resulting combined bubbles might be too large, in which case we apply a Split operation to each of them and thus remove additional edges from each.

In the next lemma we establish some basic properties of Procedure Remove and, in particular, prove that the above mentioned phenomenon is bounded.

Lemma 4.4. 1. *If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, \delta x_l]$-bubbles partition, then after the execution of Procedure* Remove$(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *and removing the edge e, the output forest $\mathcal{F}_l'$ represents a legal $[x_l, \delta x_l]$-bubbles partition as well.*

2. *Every edge added to $\mathcal{F}_l'$ during the execution of* Remove$(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *is an edge of $\mathcal{F}_{l-1}$.*

3. *The number of edges removed from $\mathcal{F}_l$ during the execution of* Remove$(e, \mathcal{F}_l, \mathcal{F}_{l-1})$ *is at most* 3.

Proof. It is easy to verify (by simple case by case inspection) that, assuming we start with a legal bubble partition, then each bubble generated during the update process is connected, and the different bubbles are disjoint and cover the entire graph.

For proving part 1, we need to argue that in the output partition every bubble is in the right size range. For establishing the upper size bound, $|\mathcal{B}| \le \delta x_l$, we rely on the fact that whenever the algorithm merges a bubble and it becomes too large, it is split.

For proving that $|\mathcal{B}| \ge x_l$, we rely on the fact that whenever the algorithm creates a bubble $\hat{\mathcal{B}}$ that is too small (as the result of the edge removal), it is combined with a tree-neighboring bubble $\tilde{\mathcal{B}}$, such that their combined size is at least x_l at this time.

The only point that needs to be verified in order to complete the above proof of part 1 is that whenever some portion $\hat{\mathcal{B}}$ of a broken bubble is too small and needs to be combined with a tree-neighboring bubble, the existence of a suitable tree-neighboring bubble $\tilde{\mathcal{B}}$ is guaranteed.

To see this, let $\mathcal{B}_{l-1}$ be a level $l-1$ bubble that includes at least one node of $\hat{\mathcal{B}}$, say v. Since level $l-1$ bubbles are always larger than level l bubbles, $\mathcal{B}_{l-1}$ is a connected component that includes more nodes than $\hat{\mathcal{B}}$ does; hence there must be a node w that is not within $\hat{\mathcal{B}}$ but is within $\mathcal{B}_{l-1}$. A path from v to w in $\mathcal{T}(\mathcal{B}_{l-1})$ must include a link connecting $\mathcal{B}'_l$ to a tree-neighboring level l bubble.

Property 2 follows immediately from the procedure and the definition of a tree-neighboring bubble.

It remains to prove property 3, namely, to show that the procedure removes at most two more edges in addition to the intended e itself. To see this, observe that if a bubble $\bar{\mathcal{B}}$ is created in step 2 by combining bubbles $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$, then $|\hat{\mathcal{B}}| \le x_l$ and $|\tilde{\mathcal{B}}| \le \delta x_l$; hence $|\bar{\mathcal{B}}| \le (\delta+1)x_l$. This implies that Procedure Split($\bar{\mathcal{B}}$) will erase a single edge, splitting $\bar{\mathcal{B}}$ into precisely two new bubbles. This is because the very first splitting operation in Procedure Partition($\mathcal{T}(\bar{\mathcal{B}}), x_l$) will create a bubble of size at least x_l, leaving the remaining portion of the bubble $\bar{\mathcal{B}}$ with fewer than δx_l nodes and thus terminating the procedure. Since steps 1 and 2 of Procedure Remove are executed at most twice, the claim follows. $\square$

The observation made just before Lemma 4.4, combined with claim 3 of the lemma, has the following important implication. Consider a single topology change in the form of the failure of a link e_1, requiring us to remove e_1 on some level l (and possibly also in other levels). This change might in fact cause us to remove up to two other edges, e_2 and e_3, on level $l+1$. But removing each of these three edges e_j on level $l+1$ may in turn cause the removal of two other edges, resulting in a total of nine removals on level 2. More generally, this single topology change might result in up to (but no more than) 3^{i-1} link removal operations in each level i.

Consequently, whenever we need to remove an edge from the partition of some level i, we typically need to immediately fix the partitions of all higher levels, $i+1$ through k, as well. This is done via Procedure RemoveAll. The procedure fixes the bubble partitions $\mathcal{P}_l$ at each level $i \le l \le k$, one at a time, starting from level i and moving up. At each level l, the procedure outputs a corrected forest $\mathcal{F}'_l$. The computation of the new bubble partition of level l, $\mathcal{P}'_l$, and the corresponding forest $\mathcal{F}'_l$, is based on the recently computed forest for level $l-1$, $\mathcal{F}'_{l-1}$, and the old forest $\mathcal{F}_l$ of level l before the topology change. Suppose the procedure already computed the new partition up to and including level $l-1$. As a result, there is a number of edges that appear in $\mathcal{F}_l$ and do not appear in $\mathcal{F}'_{l-1}$ (since they have already been removed on that level). The procedure removes each tree link of $\mathcal{F}_l - \mathcal{F}'_{l-1} = \{e_1, e_2, \ldots, e_m\}$ from $\mathcal{F}_l$. This is done sequentially, edge by edge, and the forest $\mathcal{F}_l$ is modified at each edge removal step to reflect the change. Hence the sequence of link removals creates a sequence of forests $\mathcal{F}_l = \mathcal{F}_l^{(0)}, \mathcal{F}_l^{(1)}, \ldots, \mathcal{F}_l^{(m)} = \mathcal{F}'_l$, where the update for removing e_j leads from $\mathcal{F}_l^{(j-1)}$ to $\mathcal{F}_l^{(j)}$.

It remains to describe the modification of the forest $\mathcal{F}_l^{(j-1)}$ due to the removal of e_j. If e_j is not in $\mathcal{F}_l^{(j-1)}$, nothing need be done. Otherwise, if the removal of e_j causes the partition of some bubble, then we invoke Procedure Remove($e_j, \mathcal{F}_l^{(j-1)}, \mathcal{F}_{l-1}$) to remove the edge and correct $\mathcal{F}_l^{(j-1)}$ into $\mathcal{F}_l^{(j)}$. Again, it is assumed that e_j does not appear in $\mathcal{F}_{l-1}$ and that all edges added to $\mathcal{F}'_l$ or higher partitions during the procedure's execution belong to $\mathcal{F}_{l-1}$. The procedure relies on the assumption that initially, $\mathcal{F}_l$ (for every $i-1 \le l \le k$) spans a legal bubbles partition. Procedure RemoveAll is described formally in Figure 8.

Lemma 4.5. 1. *If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, \delta x_l]$-bubbles partition for every $i-1 \le l \le k$, then after the execution of Procedure* RemoveAll(e, i) *and removing the edge e, the output forest $\mathcal{F}'_l$ represents a legal $[x_l, \delta x_l]$-bubbles partition,*

```
Input:  Edge $e$ for removal,
   forests $\mathcal{F}_l$ for $i-1 \le l \le k$ spanning legal partitions.

Action:
  Let $\mathcal{F}'_i \leftarrow$ REMOVE$(e, \mathcal{F}_i, \mathcal{F}_{i-1})$.
  For $l = i+1$ to $k$ do:
      1. Let $\{e_1, e_2, \dots, e_m\} = \mathcal{F}_l - \mathcal{F}'_{l-1}$.
      2. Let $\mathcal{F}_l^{(0)} \leftarrow \mathcal{F}_l$.
      3. For $j = 1$ to $m$ do:
           If $e_j$ is in $\mathcal{F}_l^{(j-1)}$, then invoke $\mathcal{F}_l^{(j)} \leftarrow$ REMOVE$(e_j, \mathcal{F}_l^{(j-1)}, \mathcal{F}_{l-1})$.
      4. Let $\mathcal{F}'_l \leftarrow \mathcal{F}_l^{(m)}$.
Output:  Forests $\mathcal{F}'_l$ for $i \le l \le k$.
```

FIG. 8. Procedure REMOVEALL(e, i).

as well, for every $i \le l \le k$.

2. *The output partition $\mathcal{P}'_l$ is a refinement of $\mathcal{P}'_{l-1}$ (i.e., $\mathcal{F}'_l \subseteq \mathcal{F}'_{l-1}$) for every $i \le l \le k$.*

3. *The number of edges removed from $\mathcal{F}_l$ during the execution of* REMOVEALL(e, i) *is at most 3^{l-i+1} for every $i \le l \le k$.*

Proof. Claim 1 follows directly from claim 1 of Lemma 4.4. Moreover, this property holds throughout the process, that is, after the procedure completes handling levels 1 through l and works on level $l+1$, in the internal stage after the procedure has completed handling the removal of the first j edges, and has reached the intermediate forest $\mathcal{F}^{(j)}$, the claim holds for this $\mathcal{F}^{(j)}$.

Claim 2 holds only for the final partition constructed for each level. That is, in intermediate points along the computation of the procedure for some level $l+1$, the intermediate level $l+1$ forest $\mathcal{F}_{l+1}^{(j)}$ might not obey this relationship w.r.t. $\mathcal{F}_l$ (i.e., it may not be that $\mathcal{F}_{l+1}^{(j)} \subseteq \mathcal{F}_l$).

For the final partitions, the claim follows from the fact that in constructing the new level l forest $\mathcal{F}'_{l+1}$, the procedure removes from the original level $l+1$ forest $\mathcal{F}_{l+1}$ any edge of $\mathcal{F}_{l+1} - \mathcal{F}'_l$.

Claim 3 follows directly from claim 3 of Lemma 4.4. □

Note that Lemma 4.5 holds for the case of legal bubbles; i.e., the connected component includes at least x_l nodes. Similar arguments hold for the case in which the connected component is smaller.

Until this stage we described the algorithm used to construct the bubble partition and basic procedures for maintaining the partition. Now we detail how those procedures are invoked in each of the four possibilities for topology changes.

Link removal. If the removal of the link e does not partition the spanning tree of any bubble, then no change of the hierarchical bubble partition is required. If the spanning tree of a bubble is disconnected by removing e, then the link removal is handled at each level starting from level 1 and moving up. At level 1, if the communication graph is still connected, then a nontree link is promoted to be a tree link in order to retain the connectivity of the entire spanning tree, $\mathcal{T}$. The removal itself is then performed by applying Procedure REMOVEALL$(e, 2)$.

Link addition. An addition of a link that does not connect two previously separated connected components does not change the routing database. An addition of a link that does connect two previously disconnected components G' and G'' is handled as follows. Let $\mathcal{T}(\mathcal{B}'_1)$ and $\mathcal{T}(\mathcal{B}''_1)$ be the spanning trees of G' and G'', respectively, before the link addition. Connect $\mathcal{T}(\mathcal{B}'_1)$ with $\mathcal{T}(\mathcal{B}''_1)$ by invoking Procedure Combine$(\mathcal{B}'_1, \mathcal{B}''_1)$, using the new link to form a spanning tree $\mathcal{T}(\mathcal{B}_1)$ of the new connected communication graph.

Continue with levels 2 up to k, one at a time. For level $2 \leq i \leq k$, if the new link connects two legal bubbles, then no update operations are triggered by level i. Otherwise, when the new link connects at least one illegal bubble, combine the illegal bubble with the new tree-neighboring bubble (using Procedure Combine), and split the combined bubble (using Procedure Split) if necessary. The split operation may result in the need to remove a link e at the next level. This is handled by invoking the link removal procedure RemoveAll(e, i) described above. Then the update for level $i + 1$ may be started.

Node addition or removal. Both the addition and the removal of a node can be described in terms of addition and removal of links. The addition is handled by first adding a link that connects two separated connected components one of which is the single node. Then adding the rest of the links one by one. Node removal is done by removing one link at a time and then removing the node.

In order to argue about the correctness of the dynamic update procedures, two main claims need to be established, namely, that at the end of each update operation, each partition $\mathcal{P}_i$ is a legal $[x_i, \delta x_i]$-bubble partition and that the entire hierarchy is a hierarchical bubble partition. These claims are naturally implied by the properties established earlier regarding Procedures Remove and RemoveAll.

Theorem 4.6. *At the end of each update operation, each partition $\mathcal{P}_i$ is a legal $[x_i, \delta x_i]$-bubble partition and the entire hierarchy is a hierarchical bubble partition.*

Proof. We prove the theorem for link removal and addition updates since node additions or removals are reduced to links removals or additions. Link removal employs Procedure RemoveAll. By Lemma 4.5 the result of the procedure is a legal bubble partition. In the case of link addition, each Combine execution is followed by Split and RemoveAll (if required) to result in a legal partition. ∎

4.3. The routing scheme. Next we describe the distributed routing database by the use of the bubble partition.

Routing distributed database. Start by constructing a hierarchical bubbles partition for the network. For every bubble $\mathcal{B}_k$ at level k, define the nodes that reside in $\mathcal{B}_k$ as its *members*, and choose one of these nodes to be the *leader* of the bubble, denoted $L(\mathcal{B}_k)$. In general, for level $i < k$, define the *members* of a bubble $\mathcal{B}_i$ to be all the leaders of level $i + 1$ bubbles that reside in the connected component of $\mathcal{B}_i$. Then choose one of these members to be the bubble's leader, $L(\mathcal{B}_i)$. (The single bubble of level 1, $\mathcal{B}_1$, will have no use for a leader.)

In Figure 9 we illustrate the way the members of the bubbles are chosen. The members of a bubble at level 3 are the nodes that reside in the bubble (omitted from the figure description). For each bubble at level 3 a single leader is chosen (e.g., A, D, or E); this leader is a member of the bubble of level 2 in which it resides. For every bubble at level 2 a leader is chosen among its members (e.g., D or G). These chosen leaders are members of level 1. The tree $\mathcal{T}(\mathcal{B}_i)$ is known to every member of $\mathcal{B}_i$. Note that in particular, every member of $\mathcal{B}_1$ maintains in its memory $\mathcal{T}(\mathcal{B}_1)$, which is a spanning tree of the entire communication graph. Note that for load balancing reasons the representation of the spanning tree can be extended to include the full

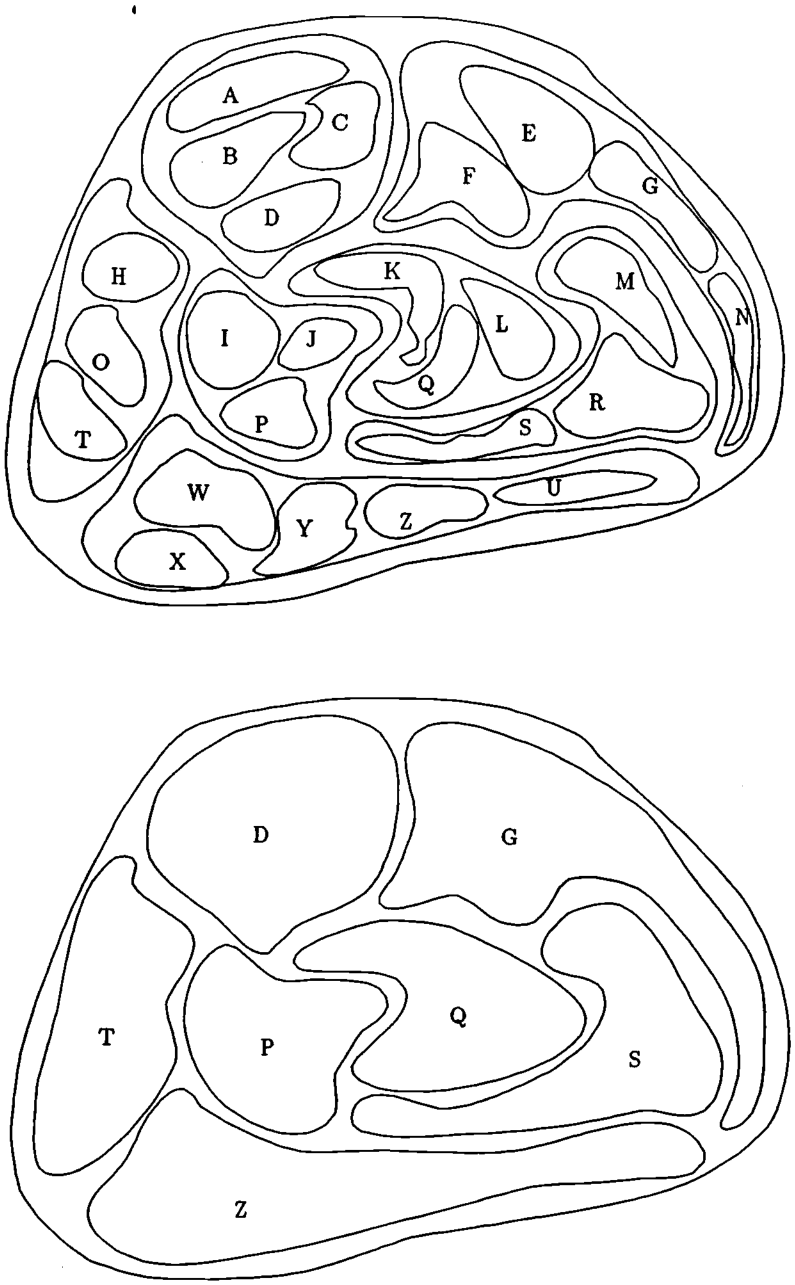


FIG. 9. *The members at level* 2 *and level* 1.

topology of the bubble, if desired, thereby increasing the memory requirement by a factor of δ. The precise method of choosing paths to avoid congestion is beyond the scope of this paper.

Routing strategy. P delivers a message m to a processor Q by the following procedure. Let $i_1 \geq k$ be the lowest level in which P is a member, and let $\mathcal{B}_{i_1}$ be its *home bubble.* Then P searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_1})$. If Q is not found in $\mathcal{T}(\mathcal{B}_{i_1})$, then P sends the packet $(path_{L_1}, Q, m)$ to $L_1 = L(\mathcal{B}_{i_1})$, the leader of $\mathcal{B}_{i_1}$. This leader, L_1, repeats the process. That is, let $i_2 < i_1$ be the lowest level in which L_1 is a member, and let $\mathcal{B}_{i_2}$ be its home bubble. Then L_1 searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_2})$, and if Q is not found, then L_1 sends the packet $(path_{L_2}, Q, m)$ to $L_2 = L(\mathcal{B}_{i_2})$.

This process must terminate (at a member of $\mathcal{B}_1$ in the worst case) since every member of $\mathcal{B}_1$ has a spanning tree description of the entire communication graph, namely, $\mathcal{T}(\mathcal{B}_1)$.

Routing update. Whenever we apply Procedure COMBINE$(\mathcal{B}'_l, \mathcal{B}''_l)$, the members of the combined bubble have to be updated regarding the new tree. Likewise, whenever we apply SPLIT$(\mathcal{B}_l)$, the members of the two split bubbles have to be updated. Finally, upon adding a new link for reconnecting the tree $\mathcal{T}$, every member of $\mathcal{B}_1$ (namely, every level 2 leader) is updated with the new tree $\mathcal{T}$, similar to any other COMBINE operation.

Analysis. We now turn to analyzing the complexity of the routing scheme. Let us first establish some basic properties of the hierarchical bubble partition. For uniformity of treatment, let us define $x_{k+1} = 1$. Recall that the size of each bubble at level i is in the range $[x_i, \delta x_i]$. Define $\pi_i = \delta x_i / x_{i+1}$. As shown in the next lemma, π_i serves as an estimate for the population of level i bubbles.

LEMMA 4.7. *The number of members in a bubble of level i is at most π_i.*

Proof. A bubble $\mathcal{B}$ of level i has at most δx_i nodes. For $i = k$, every node is a member, and we are done. Now consider $i < k$. The nodes of $\mathcal{B}$ are partitioned into bubbles of level $i+1$, each of which contains at least x_{i+1} nodes. Hence the number of level $i+1$ bubbles contained in $\mathcal{B}$ is at most $\delta x_i / x_{i+1}$. The members of $\mathcal{B}$ are precisely the leaders of these bubbles. The claim follows. □

LEMMA 4.8. *The total number of members in all level i bubbles is at most n/x_{i+1}.*

Proof. For $i = k$, all nodes are members. For $i < k$, the members of level i bubbles are precisely all leaders of level $i+1$ bubbles. Their number is at most n/x_{i+1}, since each such bubble is of size at least x_{i+1}. □

The next three lemmas state the super-hop count, the memory requirement, and the adaptability of the bubble routing scheme.

LEMMA 4.9. *The super-hop count of the bubble routing scheme is bounded by k.*

Proof. In the worst case a processor P that is only a member in level k sends a packet to its bubble leader, Q, which is a member at level $k-1$, and so on. With no more than $k-1$ packets a member of level 1 is reached; this member knows $\mathcal{T}(\mathcal{B}_1)$ and sends a direct packet to the destination of the message. □

LEMMA 4.10. *The memory requirement for the bubble routing scheme is bounded from above by $n(\log n + \log \delta) \sum_{1 \le i \le k} \pi_i$.*

Proof. Every processor maintains a spanning tree of the lowest bubble it is a member of. The members of bubbles at level i maintain a spanning tree of at most δx_i nodes, which requires $\delta x_i(\log n + \log \delta)$ bits. There are at most n/x_{i+1} members altogether at level i, so the memory requirement for the ith level is $\delta x_i(\log n + \log \delta) \cdot n/x_{i+1} = n(\log n + \log \delta)\pi_i$ bits. □

LEMMA 4.11. *The adaptability of the bubble routing scheme is bounded from above by $\delta \sum_{1 \le i \le k} 3^{i-1}\pi_i$ and by only $\sum_{1 \le i \le k} 3^{i-1}\pi_i$ in the link-failure model.*

Proof. The number of processors that change their routing database is greatest when a processor is removed. This number is bounded from above by the effect of removal of δ links one at a time.

The removal of a link that partitions a spanning tree of a bubble at every level implies the largest number of updates. On level 1, such a link removal requires at most n/x_2 updates of the members of $\mathcal{B}_1$.

We continue our analysis according to the description of the link removal procedure. Note that by claim 3 of Lemma 4.5, at most 3^{i-1} edges are removed at level i as the result of a single edge removal at level 1. Since each link belongs to at most one bubble on each level, this bounds also the number of level i bubbles whose topology has undergone changes as a result of this removal.

The routing database must be updated at every member of a bubble whose topology was changed. Hence the total number of members (on all levels) whose routing database must be updated upon a link removal is bounded by $\sum_{1\le i\le k} 3^{i-1}\pi_i$.

The total number of processors that are updated upon a node removal is no more than $\delta\sum_{1\le i\le k} 3^{i-1}\pi_i$. Both claims of the lemma follow. $\square$

We now fix $x_i = \lceil n^{(k-i+1)/k}\rceil$ for $1 \le i \le k$. Then $\sum_{1\le i\le k}\pi_i = k\delta n^{1/k}$ and $\sum_{1\le i\le k} 3^{i-1}\pi_i = \frac{1}{2}(3^k-1)\delta n^{1/k}$. The following theorem summarizes the properties of the bubbles routing scheme.

THEOREM 4.12. *The bubble routing scheme has the following properties:*

1. *super-hop count* $= k$,
2. *memory requirement* $= k\delta n^{1+1/k}(\log n + \log\delta)$,
3. *adaptability* $= 3^k\delta^2 n^{1/k}$ *in the node-failure model and* $3^k\delta n^{1/k}$ *in the link-failure model.* $\square$

5. Improved partitioning: Edge-bubble partition.

5. Improved partitioning: Edge-bubble partition. The bubble routing scheme as described above is efficient when δ is small, and in particular, for bounded degree networks there is a little significance to this factor. However, if δ is large (close to n) and we consider only link failures, the bubble scheme may not compare favorably even with the simple multiple trees scheme, because of the occurrences of δ in the expression for adaptability (and in the expression for memory). For such a case it is desirable to get rid of the dependence of the complexities on δ. Note that a δ factor for the adaptability is inherent to the problem if we consider a model that allows node failures and recoveries. This δ factor is based on the fact that a node addition can cause the connection of δ previously disconnected components.

We now present a modified partitioning scheme called the *edge-bubble partition*, which eliminates a δ factor (occurring in both the bound of adaptability and memory) at the cost of using a somewhat more complex partition and algorithm. Recall that in order to partition the communication graph into node disjoint connected components, the bubble scheme presented above allows the range of the bubble size to be $[x, \delta x]$. It turns out that it is possible to form a bubblelike partition of a tree into *edge-disjoint* trees (i.e., with bubbles possibly sharing nodes), such that the size (i.e., number of edges) of each tree is between x and $3x$.

DEFINITION 5.1. *Given a graph* $G = (V, E)$ *and parameters* $0 < a < b$, *an* $[a, b]$ edge-bubble partition *of* G *is a (possibly node-overlapping) cover of* V *by a collection of edge-disjoint* connected *trees called* bubbles, $\{\mathcal{B}_1, \ldots, \mathcal{B}_q\}$, *where* $\mathcal{B}_i = (V_i, E_i)$, *with the property* $E_i \cap E_j = \emptyset$ *for every* $1 \le i < j \le q$.

Intuitively, the modified scheme can be thought of as utilizing the idea of transforming the tree $\mathcal{T}$ at hand into a *logical* tree $\hat{\mathcal{T}}$ of maximum degree 3 and applying the previous solution to this logical tree. The transformation can be achieved by splitting each node P of degree d into d copies $P_1, \ldots, P_d$, connected by a simple chain, and hanging exactly one child Q_i of P off each copy P_i (see Figure 10).

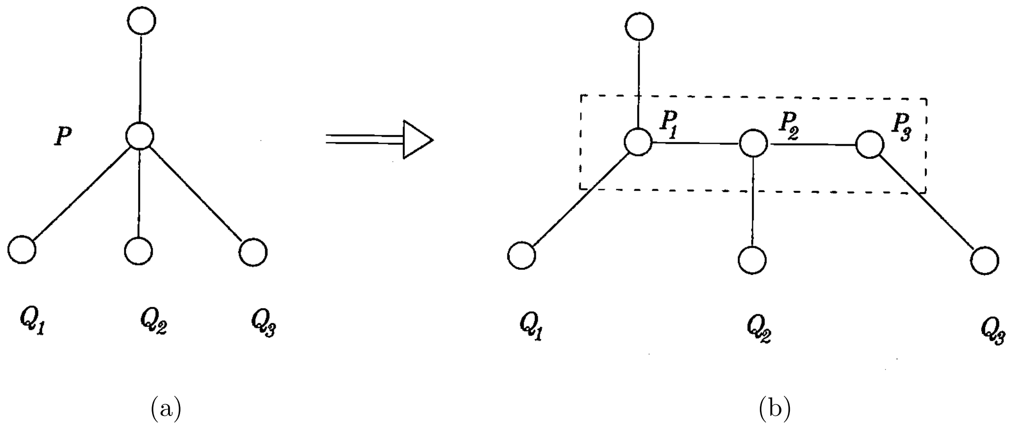


FIG. 10. (a) *A node P of degree* 3. (b) *P is split into three logical copies* P_1, P_2, P_3.

The bubbles solution on the logical tree is simulated on the actual network by requiring each node P to simulate the behavior of its d logical copies. Some care is still needed with respect to the modification procedures, and in particular, it is necessary to specify how a node P reacts to topological changes in terms of reorganizing its internal (logical) structure—for example, as a result of a combine operation that changed its root (and made its former parent into a child and one of its former children into its new parent). In what follows, the resulting algorithm is described directly (rather than via the intuitive simulation method discussed above), in order to gain better understanding concerning its behavior.

We next present an overview of a partitioning algorithm, that given an n-node graph G and a parameter $1 \leq x \leq n$, produces an $[x, 3x]$ edge-bubble partition for G. The new algorithm is similar in nature to Algorithm PARTITION (Figure 2). The essential differences between the two algorithms are that in the new algorithm (i) M_P is the total number of *edges* in the subtree $\mathcal{T}_P$, and (ii) The treatment of a node P found in step (2) of the algorithm, namely, a node P satisfying $M_P \geq x$, such that all of P's children Q satisfy $M_Q < x$, is different. For a particular edge e from P to one of its children, Q, we define a subtree, $\mathcal{T}_e$, rooted at e to be the subtree rooted at Q together with e. Recall that the previous algorithm simply makes the subtree $\mathcal{T}_P$ rooted at P into one bubble, containing, in particular, all the subtrees rooted at P's children. In contrast, the new algorithm will select *some* of the subtrees $\mathcal{T}_e$ rooted at P's outgoing edges of total size greater than x but not exceeding $2x$. The selected subtrees are now merged into an edge-connected bubble. This bubble is removed from the tree. We note that the resulting bubbles are of size $[x, 3x]$, since the partition of the last bubble into two bubbles of size in the range $[x, 2x]$ might not be possible in the case of P having three outgoing edges for which the subtree rooted at each of them is of size less than x but greater than, say, $3x/4$.

We proceed with a more formal description of the algorithm. As in the previous algorithm, a preprocessing step involves constructing a downward-directed spanning tree $\mathcal{T}_R$ for the graph, rooted at an arbitrary node R. The algorithm then partitions this tree into bubbles (see Figure 11).

```
(1) While $\mathcal{T}$ contains more than $3x$ nodes do:
        (a) Mark every node $P$ in $\mathcal{T}$ by $M_P$, the total number of edges in the
            subtree $\mathcal{T}_P$ rooted at $P$.
        (b) Find a node $P$ with the property that $M_P \geq x$, but all of its children
            $Q$ satisfy $M_Q < x$.
        (c) Let $E_P = (e_1, e_2, \ldots, e_i)$ be the edges connecting $P$ to its children.
            Let $n_i$ be the number of edges in the subtree connected to $P$ by
            $e_i$ (including $e_i$). Let $j$ be the minimal index $j$ such that $x \leq$
            $\sum_{i=1}^{j} n_i \leq 2x$.
        (d) Make the edges of the subtrees connected to $P$ by $e_1, e_2, \ldots, e_j$
            (including the edges $e_1, e_2, \ldots, e_j$ themselves) into a bubble.
        (e) Remove this bubble from $\mathcal{T}$.
        End_while
(2) Make $\mathcal{T}$ into a bubble and terminate.
```

FIG. 11. Algorithm EDGEPARTITION(x).

LEMMA 5.1. *Algorithm* EDGEPARTITION(x) *produces an* $[x, 3x]$ *edge-bubble partition of* G.

Proof. The proof is a variant of that of Lemma 4.1. Step (1c) guarantees, by the choice of j, that the selected collection is of total size no greater than $2x$.

Showing that every execution of step (1) succeeds in finding a candidate node P as required in (1b) follows just as in the proof of Lemma 4.1. Hence termination of the algorithm follows in the same way too.

Next, we show that when the algorithm terminates, $x \leq M_R \leq 3x$; hence the last bubble is within the correct size range as well. The termination condition of step (1) implies that when the algorithm terminates, $M_R \leq 3x$. If the number of nodes in the entire graph G is also smaller than $3x$, then this is the only bubble and we are done. Otherwise, there exists at least one more bubble. Examine the last removal executed at step (1c) before the termination of the algorithm. Right before this disconnection M_R is greater than $3x$, and during the disconnection no more than $2x$ edges are removed from $\mathcal{T}_R$, leaving at least (a set of) x edges that are still connected to R (i.e., a path from R to each of the edges in this set consists of only edges of the set). □

Theorem 5.2 is implied by the existence of the partition algorithm.

THEOREM 5.2. *For every* n*-node graph* G *and parameter* $1 \leq x \leq n$, G *has an* $[x, 3x]$ *edge-bubble partition.*

Figure 12 depicts a step in the partition of a graph into edge-bubbles. The tree of the (remaining) graph is depicted in the top portion of the figure. It is assumed that $x = 6$ and hence the partition takes place only in the node of level 2 for which every subtree has less than 6 edges. The partition gathered the first two subtrees to be a bubble of size greater than 6 and not exceeding 12.

Again, for our routing schemes we will use hierarchies of edge-bubble partitions, defined as follows.

DEFINITION 5.2. *Given a graph* G *and a list of* k *integer parameters* $\bar{x} = (x_1, x_2, \ldots, x_k, x_{k+1})$, *where* $x_1 = n$, $x_{k+1} = 1$, *and* $x_i > 3x_{i+1}$, *an* $\bar{x}$-hierarchical edge-bubbles partition *of* G *is a collection of* k *edge-bubbles partitions* $\mathcal{P}_i$, $1 \leq i \leq k$, *with the following properties.*

1. *For every* $1 \leq i \leq k$, $\mathcal{P}_i$ *is an* $[x_i, 3x_i]$ *edge-bubble partition.*
2. *For every* $1 < i \leq k$, $\mathcal{P}_i$ *is a refinement of* $\mathcal{P}_{i-1}$.

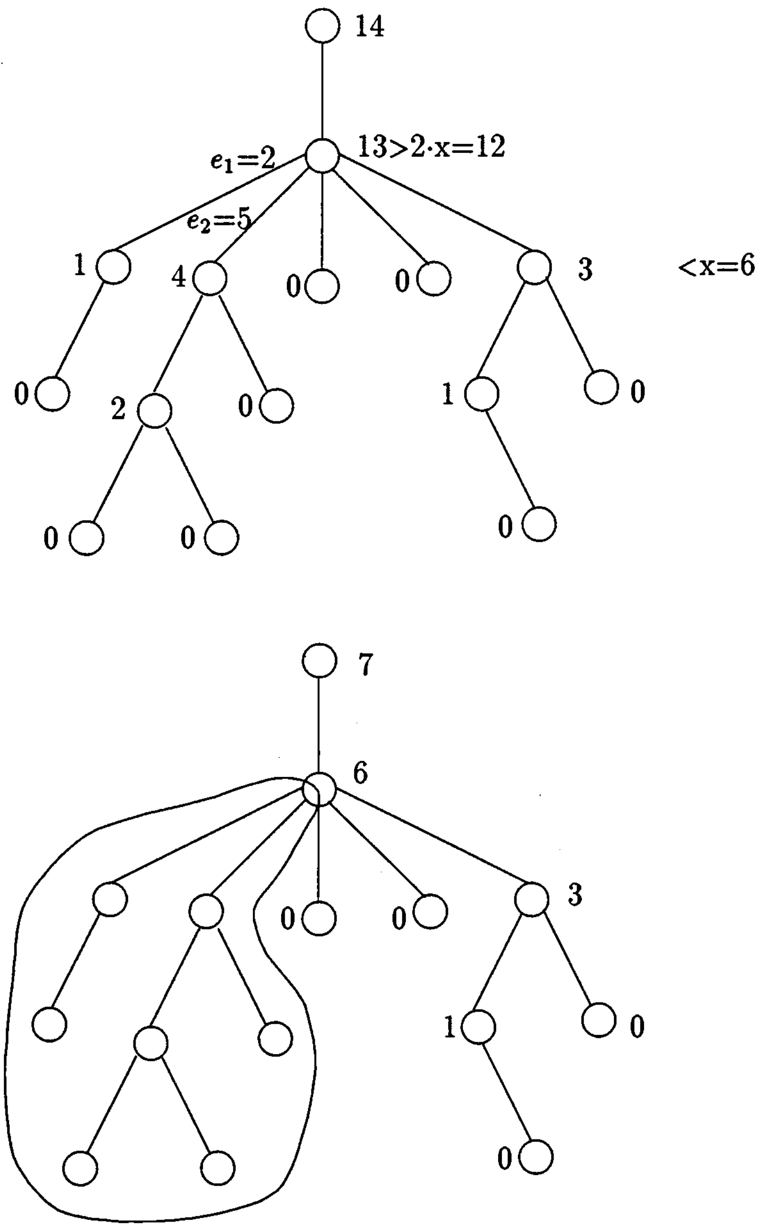


FIG. 12. *A step of the edge-bubbles partition.*

Similar to the previous section, we choose the parameters $\bar{x}$ as $x_i = \lceil n^{(k-i+1)/k} \rceil$.

The hierarchical edge-bubble partition of a graph is constructed by Algorithm EDGEHIER($\bar{x}$) in a manner similar to Algorithm HIERARCHY($\bar{x}$) of section 4, except that the partitioning algorithm invoked is EDGEPARTITION rather than PARTITION. The requirements of Definition 5.2 are clearly met, giving the following theorem.

Input: Two edge-tree-neighboring level l bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$.

Action: Connect $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$ using their connecting node.

Output: A combined bubble $\mathcal{B}_l$.

FIG. 13. Procedure EDGECOMBINE($\mathcal{B}'_l, \mathcal{B}''_l$).

Input: A (typically oversized) bubble $\mathcal{B}_l$ and $\mathcal{F}_{l+1}$.

Action: Select one of the nodes of $\mathcal{B}_l$ to be the root R.
Direct $\mathcal{T}(\mathcal{B}_l)$ from R towards the leaves.
Execute steps (1) to (2) of the Algorithm EDGEPARTITION(x_l) (ordered w.r.t. $\mathcal{F}_{l+1}$).

Output: An $[x_l, 3x_l]$ edge-bubbles partition of $\mathcal{B}_l$.

FIG. 14. Procedure EDGESPLIT($\mathcal{B}_l, \mathcal{F}_{l+1}$).

THEOREM 5.3. *Every graph has a hierarchical edge-bubble partition.*

5.1. Dynamic maintenance. Dynamic maintenance of the bubbles is carried out in much the same way as with the basic algorithm of section 4.

The combine and split procedures for the edge partition bubbles appear in Figures 13 and 14. We use the following definition for *edge-tree-neighboring.*

DEFINITION 5.3. *Two bubbles $\mathcal{B}'_l$ and $\mathcal{B}''_l$, both at level l, are said to be* edge-tree-neighboring *w.r.t. the level $l-1$ forest $\mathcal{F}_{l-1}$ if there exist two edges $e' \in \mathcal{B}'_l$ and $e'' \in \mathcal{B}''_l$ adjacent to the same node such that both e' and e'' are in the same bubble of $\mathcal{F}_{l-1}$. We refer to the shared adjacent node of e' and e'' as the* connecting node *of $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$.*

It turns out that the order of the edges $e_1, e_2, \ldots, e_i$ of step (1c) of Algorithm EDGEPARTITION is crucial in the context of dynamic maintenance of the hierarchical partition. When a bubble of level l is split, it might split bubbles of level $l+1$. We would like to minimize the number of bubbles at level $l+1$ that are split because a split of level l took place. Toward this end, we specify the order of the edges $e_1, e_2, \ldots, e_i$ in a way that ensures that at most one bubble of level $l+1$ is split because of a split operation of a bubble in level l. The next definition uses e_0 to denote the edge connecting P with its parent in $\mathcal{T}$.

DEFINITION 5.4. *The execution of Algorithm* EDGEPARTITION *is said to be* ordered w.r.t. $\mathcal{F}_{l+1}$ *if in step* (1c) *of Algorithm* EDGEPARTITION*, the edges $E_P = (e_1, e_2, \ldots, e_i)$ are ordered according to $\mathcal{F}_{l+1}$ in such a way that, for every bubble $\mathcal{B}_{l+1}$, the edges of E_P that belong to $\mathcal{B}_{l+1}$ appear in one contiguous subsequence of E_P. In addition, if e_0 belongs to $\mathcal{B}'_{l+1}$, then the contiguous subsequence of edges in the ordered E_P that belong to $\mathcal{B}'_{l+1}$ (if any) is ordered last among the other contiguous subsequences.*

Procedures EDGEREMOVE and EDGEREMOVEALL appear in Figures 15 and 16. The next lemma states some basic properties of procedure REMOVE.

LEMMA 5.4. 1. *If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, 3x_l]$ edge-bubble partition, then after the execution of Procedure* EDGEREMOVE($e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1}$) *and removing edge e, the output forest $\mathcal{F}'_l$ represents a legal $[x_l, 3x_l]$ edge-bubble partition as well.*

Input: Edge e for removal, forest $\mathcal{F}_l$, $\mathcal{F}_{l-1}$ and $\mathcal{F}_{l+1}$ spanning a legal partition.

Action:

```
The removal of $e_i$ causes the partition of a single bubble, say $\mathcal{B}_l$, into two portions,
denoted $\mathcal{B}'_l$ and $\mathcal{B}''_l$. Let $\hat{\mathcal{B}}_l$ be an edge-tree-neighboring of $\mathcal{B}_l$ and w.l.o.g. of $\mathcal{B}'_l$.
For $\hat{\mathcal{B}} = \mathcal{B}'_l$ and then $\hat{\mathcal{B}} = \mathcal{B}''_l$ do:
    1. If the size of $\hat{\mathcal{B}}$ is still greater than $x_l$, then make it into an independent
       bubble without change.
    2. Otherwise, if it is too small, then do the following:
        (a) If $\hat{\mathcal{B}} = \mathcal{B}'_l$ then let $\tilde{\mathcal{B}}$ be $\hat{\mathcal{B}}_l$.
            Else find some bubble $\tilde{\mathcal{B}}$ of $\mathcal{F}_l$ that is an edge-tree-neighbor of $\hat{\mathcal{B}}$
            w.r.t. $\mathcal{F}_{l-1}$.
        (b) Merge $\tilde{\mathcal{B}}$ and $\hat{\mathcal{B}}$ into $\bar{\mathcal{B}}$ by invoking procedure EdgeCombine($\tilde{\mathcal{B}}, \hat{\mathcal{B}}$).
        (c) If $\bar{\mathcal{B}}$ includes more than $3x_l$ nodes, then split it by invoking proce-
            dure EdgeSplit (ordered w.r.t. $\mathcal{F}_{l+1}$).
```

Output: Forest $\mathcal{F}'_l$.

FIG. 15. Procedure EDGEREMOVE$(e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1})$.

2. *Every two bubbles combined into a single bubble of $\mathcal{F}'_l$ during the execution of* EDGEREMOVE$(e, \mathcal{F}_l, \mathcal{F}_{l-1}, \mathcal{F}_{l+1})$ *resides in a single bubble of $\mathcal{F}_{l-1}$.*

Proof. It is easy to verify (by simple case-by-case inspection) that, assuming we start with a legal bubbles partition, each bubble generated during the update process is connected and that the different bubbles are disjoint and cover the entire graph.

For proving part 1, we need to argue that in the output partition, every bubble is in the right size range. For establishing the upper size bound, $|\mathcal{B}| \leq 3x_l$, we rely on the fact that whenever the algorithm merges a bubble and it becomes too large, it is split.

For proving that $|\mathcal{B}| \geq x_l$, we rely on the fact that whenever the algorithm creates a bubble $\hat{\mathcal{B}}_j$ that is too small (as the result of an edge removal operation), it is combined with a tree-neighboring bubble $\tilde{\mathcal{B}}$ that is of size at least x_l.

The only point that needs to be verified in order to complete the above proof, is that whenever some portion $\hat{\mathcal{B}}$ of a broken bubble is too small and needs to be combined with a tree-neighboring bubble, the existence of a suitable tree-neighboring bubble $\tilde{\mathcal{B}}$ is guaranteed.

To see this, let $\mathcal{B}_{l-1}$ be a level $l-1$ bubble that includes at least one node of $\hat{\mathcal{B}}$, say v. Since level $l-1$ bubbles are always larger than level l bubbles, $\mathcal{B}_{l-1}$ is a connected component that includes more edges than $\hat{\mathcal{B}}$ does, hence there must be an edge w that is not within $\hat{\mathcal{B}}$ but is within $\mathcal{B}_{l-1}$. A path in $\mathcal{T}(\mathcal{B}_{l-1})$ from v to one of the nodes attached to w must include a node connecting $\hat{\mathcal{B}}$ to an edge-tree-neighboring level l bubble.

Property 2 follows immediately from the procedure and the definition of an edge-tree-neighboring bubble. ◻

The following lemma uses Lemma 5.4 to prove some properties of Procedure REMOVEALL.

LEMMA 5.5. 1. *If $\mathcal{F}_l$ is the spanning forest of a legal $[x_l, 3x_l]$ edge-bubble partition for every $i \leq l \leq k$, then after the execution of Procedure* REMOVEALL(e, i) *and*

```
Input:  Edge $e$ for removal, forests $\mathcal{F}_l$ for $i-1 \le l \le k$ spanning legal partitions.

Action:
  Let $\mathcal{F}'_i \leftarrow$ EdgeRemove($e, \mathcal{F}_i, \mathcal{F}_{i-1}$).
  For $l = i+1$ to $k$ do:
      1. Let $\{\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_m\}$ be all the bubbles such that, $\mathcal{B}_j$, $1 \le j \le m$, resides
         in more than a single bubble of $\mathcal{F}_{l-1}$.
      2. Let $\mathcal{F}_l^{(0)} \leftarrow \mathcal{F}_l$.
      3. For $j = 1$ to $m$ do:
         (a) Partition $\mathcal{B}_j$ into $\{\mathcal{B}_{j_1}, \mathcal{B}_{j_2}, \dots, \mathcal{B}_{j_x}\}$ such that each $\mathcal{B}_{j_y}$, $1 \le y \le x$,
             is the maximal portion of $\mathcal{B}_j$ that resides in a single bubble of $\mathcal{F}_{l-1}$.
         (b) For $y = 1$ to $x$ do:
              i. Make $\mathcal{B}_{j_y}$ a bubble.
             ii. If the size of $\mathcal{B}_{j_y}$ is too small then find an edge-tree-neighboring
                 bubble of $\mathcal{B}_{j_y}$, $\mathcal{B}'_j$, EdgeCombine($\mathcal{B}_{j_y}, \mathcal{B}'_j$). If the resulting
                 bubble, $\mathcal{B}''_j$, is a too big bubble then EdgeSplit($\mathcal{B}''_j, \mathcal{F}_{l+1}$)
                 (ordered w.r.t. $\mathcal{F}_{l+1}$).
      4. Let $\mathcal{F}'_l \leftarrow \mathcal{F}_l^{(m)}$.

Output:  Forests $\mathcal{F}'_l$ for $i \le l \le k$.
```

FIG. 16. Procedure EdgeRemoveAll(e, i).

removal of the edge e, the output forest $\mathcal{F}'_l$ *represents a legal* $[x_l, 3x_l]$ *edge-bubble partition for every* $i \le l \le k$.

2. *The output partition* $\mathcal{P}'_l$ *is a refinement of* $\mathcal{P}'_{l-1}$ *(i.e.,* $\mathcal{F}'_l \subseteq \mathcal{F}'_{l-1}$*) for every* $i \le l \le k$.

Proof. Claim 1 follows directly from claim 1 of Lemma 5.4 and the EdgeCombine and EdgeSplit procedures applied to the portions of the bubbles. Note that in the execution of EdgeCombine($\mathcal{B}_{j_y}, \mathcal{B}'_j$), $\mathcal{B}'_j$ is not a portion of $\mathcal{B}_j$ and thus is a legal-sized bubble. Moreover, as in Lemma 4.5, this property holds throughout the process, namely, for each intermediate forest $\mathcal{F}^{(j)}$.

Claim 2 holds (again as in Lemma 4.5) only for the final partition constructed for each level.

For the final partitions, the claim follows from the fact that in constructing the new level l forest $\mathcal{F}'_{l+1}$, the procedure handles every bubble portion. □

The update operations are handled in a similar fashion to the first bubble partition operations, yielding the next theorem.

THEOREM 5.6. *At the end of each update operation, each partition* $\mathcal{P}_i$ *is a legal* $[x_i, 3x_i]$ *edge-bubble partition, and the entire hierarchy is a hierarchical edge-bubble partition.*

5.2. The routing scheme. The routing proceeds as in section 4. The only points to note are the following.

A node with adjacent edges $e_1, e_2, \dots, e_j$ is a member of every bubble at level k to which one of its edges belongs. For each bubble of level k, a single *leader-edge* is elected from among the edges of the bubble—then one of the adjacent node of the leader-edge is elected to be the leader (node) of the bubble. Note that a single node may function as a leader of several (up to δ) bubbles.

The elected leader-edge and the elected leader (node) are members of the bubble of level $k-1$ at which the elected leader-edge resides. For each bubble of level $k-1$, $\mathcal{B}_{k-1}$, a single leader-edge is elected from among its edge members. The elected edge and the leader of level $k-1$ attached to it are members of the bubble of level $k-2$ at which the elected edge resides. This membership definition is naturally extended to the next lower levels. Similar to the node partition bubbles, a member of a bubble maintains the spanning tree portion of the bubble.

Analysis. The next three lemmas state the super-hop count, the memory requirement, and the adaptability of the edge-bubble routing scheme. The end result is very similar to the analysis of section 4, except that π_i is redefined as $\pi_i = 3x_i/x_{i+1}$. The proof of the next lemma is identical to the proof of Lemma 4.9.

LEMMA 5.7. *The super-hop count of the edge-bubble routing scheme is bounded by k.*

LEMMA 5.8. *The memory requirement for the edge-bubble routing scheme is bounded from above by $n(\log n + \log \delta)\sum_{1\le i\le k}\pi_i$.*

Proof. For every edge e such that the lowest level bubble that e is a member of is $\mathcal{B}_i$ there is an adjacent processor that maintains a description of $\mathcal{T}(\mathcal{B}_i)$. A bubble of level i includes at most $3x_i$ edges. The total number of bits required for such a description of a tree is $3x_i(\log n + \log \delta)$. Since there are at most n/x_{i+1} edge members altogether at level i, the memory requirements for the ith level is $3x_i(\log n + \log \delta)\cdot n/x_{i+1} = n(\log n + \log \delta)\pi_i$ bits. □

LEMMA 5.9. *The adaptability of the edge-bubble routing scheme is bounded from above by $\delta\sum_{1\le i\le k}3^{i-1}\pi_i$ and by only $\sum_{1\le i\le k}3^{i-1}\pi_i$ in the link-failure model.*

Proof. The number of processors that change their routing database is greatest when a processor is removed. This number is bounded from above by the effect of removal of δ links, one at a time.

The removal of a link that partitions a spanning tree of a bubble at every level implies the largest number of updates. On level 1, such a link removal requires at most n/x_2 updates of the members of $\mathcal{B}_1$.

We continue our analysis according to the description of the link removal procedure. Since we use ordered partition whenever EDGEREMOVE is invoked, a partition of a bubble at level l results with partition of up to three bubbles of level $l+1$. The bubble in which the partition of level l occurred and two other neighboring bubbles.

Thus, at most 3^{i-1} bubbles are partitioned at level i as the result of a single edge removal at level 1. Hence the total number of members (on all levels) whose routing database must be updated upon a link removal is bounded by $\sum_{1\le i\le k}3^{i-1}\pi_i$.

The total number of processors that are updated upon a node removal is no more than $\delta\sum_{1\le i\le k}3^{i-1}\pi_i$. Both claims of the lemma follow. □

We now fix $x_i = \lceil n^{(k-i+1)/k}\rceil$ for $1 \le i \le k$. Then $\sum_{1\le i\le k}\pi_i = k\delta n^{1/k}$ and $\sum_{1\le i\le k}3^{i-1}\pi_i = \frac{1}{2}(3^k-1)\delta n^{1/k}$. The following theorem summarizes the properties of the edge-bubble routing scheme.

THEOREM 5.10. *The edge-bubble routing scheme has the following properties:*

1. *super-hop count $= k$,*
2. *memory requirement $= kn^{1+1/k}(\log n + \log \delta)$,*
3. *adaptability $= 3^k\delta n^{1/k}$ in the node-failure model and $3^k n^{1/k}$ in the link-failure model.* □

6. Lower bound. In this section we prove a lower bound on the adaptability for graphs with bounded degree δ and super-hop count k. For the purpose of establishing the lower bound, we consider a single destination for all messages and then derive a tree of superhops, rather than of links.

Given any source-oblivious routing scheme with super-hop count k, we build a spanning tree T_k as follows. The root of the spanning tree is a processor P. Assume that every processor is to send a message m to P, and connect each processor Q with the destination of its first packet when Q sends m to P. The *depth* of a tree is the maximal number of edges from the root to a leaf.

CLAIM 6.1. *T_k is of depth no more than k.*

Proof. Otherwise the super-hop count is greater than k. □

CLAIM 6.2. *There exists at least one node with at least $(n/2)^{1/k}$ children in T_k.*

Proof. There are n processors in T_k and the depth of T_k is k. Letting Δ denote the maximum out-degree of T_k, the number of nodes in the tree satisfies $n \leq \sum_{0 \leq i \leq k} \Delta^i \leq 2\Delta^k$ (as $\Delta \geq 2$, implied by $n > k$). Hence $\Delta \geq (n/2)^{1/k}$. □

THEOREM 6.3. *Any source-oblivious routing scheme with super-hop count k has adaptability $\Omega(n^{1/k})$ in a communication graph of constant degree.*

Proof. Let Q be a processor that has at least $(n/2)^{1/k}$ children in T_k. The existence of Q is implied by Claim 6.2. The node Q is connected to the rest of the processors in G by at most δ links. Thus, by the pigeon-hole principle there must be a link, (Q, R), used by a set C of at least $(n/2)^{1/k}/\delta$ processors in sending their first packet to carry the message to P. We now show a sequence of at most three topology changes, involving *edge* removals and additions only, that implies $(n/2)^{1/k}/(3\delta)$ adaptability.

If a disconnection of (Q, R) does not partition the communication graph, then obviously this topology change alone forces all $(n/2)^{1/k}/\delta$ processors of the set C to change their routing database.

Now we analyze the case in which the removal of the link (Q, R) does partition the graph. We need to show that there exists a sequence of changes that preserve the bound on the degree δ and forces an omission of the link (Q, R) from the routing tables. We claim there are two different cases.

Case 1. Before the disconnection of (Q, R), the graph G is a tree, and the number of processors is greater than two. In this case, following the disconnection of (Q, R) there must be at least one leaf $X \notin \{Q, R\}$, say, w.l.o.g. in the connected component of Q. Then one more topology change, involving the reconnection of X to R, is possible without exceeding δ. As a result of this change, the graph becomes connected again, and the nodes of the set C are forced to change their routing database to use the resulting tree for reaching P.

Case 2. Before the disconnection of (Q, R), the graph G contains at least one cycle. In this case, following the disconnection one of the connected components must contain a cycle (otherwise no partition would take place). Let (X, Y) be a link in this cycle and w.l.o.g. assume that (X, Y) is in Q's connected component and that $X \neq Q$. Disconnect (X, Y) and connect X to R. Again the connection of X to R is possible without exceeding δ.

Thus at most three topology changes cause $(n/2)^{1/k}/\delta$ processors to change their routing database. □

7. Distributed bubble update.

7. Distributed bubble update. So far we have not been concerned with the issue of how to distribute the bubble update algorithm. Still the upper bound described for the adaptability of the bubble routing scheme is useful for the case of manually adding and removing links and nodes. This is the case for telephone networks, where new users may be connected and existing users may be disconnected. However, our results are made stronger by handling topology changes automatically while keeping the low adaptability. That is, the number of node control units that participate in the distributed automatic update together with the number of node control units that must change their portion of the routing distributed database is low. For the sake of keeping a low adaptability, we introduce a distributed bubble update algorithm that can cope with transient failures and recoveries as well as permanent disconnection and connections.

The main idea for the distributed bubble update is to have the processors neighboring a topology change to be the *monitors* of the change. The task of a monitor is (1) to collect information on the tree structure it belongs to and on the existing routing database, (2) to try to merge with other neighboring trees, and (3) to modify the existing routing database to fit the current topology. The modification should be performed in a way that involves the least number of node control units.

Toward this end it is assumed that each switching subsystem has a *sack variable* which may be modified by the node control unit. This sack variable contains the processor's portion of the distributed routing database. The sack variable can be collected by a special arriving message. We also assume that an arriving message may be concatenated with the label of the link through which it arrived. Note that no direct modification of the set of labels or the sack variable by an arriving message is allowed.

The links of the spanning tree $\mathcal{T}(\mathcal{B}_1)$ are distributively marked on the edges of the system. The two endpoints of a tree link are marked by a label t. In the following, we use the term *marked tree* to be the graph obtained by the set of marked links and the nodes they connect. Upon failure of a tree link the marked tree of a connected component may essentially be a marked forest that has to be fused to a new marked tree. Each tree in the forest is an autonomous entity with a monitor. The monitors of trees in the forest negotiate in order to promote a nontree link into a tree link. Upon such a promotion the number of trees (and monitors) in the forest is reduced by one. More details follow.

A monitor is associated with the time of the topology change that created it. We assume the existence of a synchronized clock at every processor. Each monitor uses the marked tree it belongs to for *tree broadcast with feedback* (in short, TBF). The TBF collects the information in the sacks of every processor that belongs to the tree and delivers the information to the monitor. The monitor initiates the TBF by sending a message with its identifier, the timestamp of its creation, and a TB (tree-broadcast) label on every marked tree link. Every switch that receives a message m of type TB concatenates the unique label of the link through which m arrived to the end of the message and forwards a copy of the message to each link that is labeled t except the link through which m arrived. If no other tree link exists (the switch is a leaf in the marked tree), then the switch modifies the message as follows: TB is replaced by TF (tree feedback), the labels at the end become the address of the message (that transfer it back to the TBF initiator), and the content of the sack is concatenated to the end of the message. When a switch receives a TF message it removes the first label of the message that arrived and concatenates the value of its sack to the tail of the message.

A processor P starts to act as a monitor when either of the tree links attached to it fails or an attached link recovers. A monitor marks the label that leads from its switching subsystem to itself by the label m. This enables the switching subsystem to deliver a copy of every BF message to its node control unit when the node control unit is a monitor. Then the monitor collects information on its tree and the distributed routing database of this tree. This is done by the tree broadcast with feedback mechanism. Whenever a tree broadcast with feedback of another monitor arrives at (the node control unit of) P and the timestamp of the arriving TBF is greater (breaking ties by the monitors identifiers), P stops being a monitor.

Upon receiving the information from the tree broadcast with feedback, P checks whether there are neighboring processors that do not belong to its tree. If such neighboring processor exists, then P sends a *promote* message to the processor Q attached to the link with the highest lexicographic order that leads to a neighboring tree. (For purposes of this lexicographic ordering, the name of a link is defined as the pair of identifiers of the two processors that are attached to it, where the first identifier in the pair is always greater than the second.) Then Q becomes the monitor and queries the other side about the possibilities of merging. Upon receiving an accept message, the link changes its status to a tree link, and the monitor with the smaller identifier sends the information collected in its TBF to its neighbor. At the same time it stops being a monitor. When there is no further possibility of merging, the corrections of the routing database are sent by the monitor to the appropriate node control units, and the processor that acted as a monitor resumes operation as a regular processor.

LEMMA 7.1. *In every instance of time and for every connected component, if the marked tree does not span the entire connected component, then for each tree in the forest there exists at least one monitor.*

Proof. This is true in the first instance of the system. Further topology changes keep this invariant. A monitor stops existing only when another monitor exists in the same tree or when the marked tree spans the entire connected component. □

LEMMA 7.2. *If no monitor exists in a connected component, then the distributed routing database of this connected component is correct.*

Proof. For every connected component of more than a single processor, there has been an instance in which one processor of this connected component has been a monitor, i.e., the instance that follows a connection of any two neighboring processors by a link. The last processor that stopped being a monitor in this connected component must have finished the corrections of the distributed routing database. □

LEMMA 7.3. *At some time following the last topology change, no monitor exists.*

Proof. The tree broadcast with feedback terminates. Then the monitor may wait for a promotion of a link into a tree link. Assume toward contradiction that the monitor waits for some link promotion forever. This in turn implies that the monitor of the other tree portion is waiting on another edge that has a greater identifier. Since links that are to be promoted into tree links are chosen according to lexicographic order, this chain of waiting monitors is finite and must end with two monitors waiting on the same edge. □

THEOREM 7.4. *The distributed adaptability is of the adaptability order.*

Proof. Only monitors and the node control units that have to change their distributed routing database participate in the distributed bubbles update. The number of monitors is no more than 2δ for a topology change: at most δ are directly influenced and another δ are the result of monitor migration.

Note that, during the correction process of the distributed routing database by one monitor, a topology change might take place, stopping the update before it is completed. Nevertheless, the partial correction process took place in a limited number of bubbles (as explained for the centralized case), leaving most of the bubbles unaffected.

Thus, the number of node control units that have to change their distributed routing database following c topology changes is $O(ck3^{k-1}\delta^2 n^{1/k})$.

The theorem is proved since the additional $O(\delta)$ monitors do not change the $O(k3^{k-1}\delta^2 n^{1/k})$ adaptability. □

8. Concluding remarks.

In this paper we defined new measures for the efficiency of routing schemes for high-speed dynamic networks. We presented the concept of the bubble partition of a graph to support routing in high-speed dynamic networks. We believe that the resulting bubble partitioning technique may be of independent interest in other contexts as well. Our routing scheme is efficient in terms of the number of routing steps (super-hop count), memory, and adaptability. The routing scheme also supports load balancing of traffic by allowing each processor to forward a message along a randomly chosen path within its bubble (the path does not necessarily belong to the bubble tree). The scheme has been proven optimal in its adaptability for a network of bounded degree and for a small constant k by a matching lower bound.

A number of variants of our scheme are possible in different contexts. We briefly discuss some of them next.

8.1. The programmable model.

The basic model as described assumes that the labels associated with each link are fixed and permanent. A natural variant of this model is one in which it is assumed that the switching subsystem permits the node control unit to configure the link labels dynamically, or in other words, that the link labels are programmable. This assumption is commonly made in the high-speed model, including PARIS (cf. [5]), and in fact, it is often assumed that each link may have a *set* of usable labels. We shall refer to this variant of the model as the *programmable model*.

A possible implementation enabling this assumption is as follows. Each port in the processor has a permanent (hard-wired) *physical id*. In addition, the link has a logical label, which can be changed from time to time. The switching subsystem maintains a *conversion mechanism*, enabling it to translate any given logical label into the appropriate physical port id. Hence upon the arrival of a message, the switching subsystem removes the first label l, consults the mechanism for the appropriate physical port id, and forwards the message to that port for transmission.

For the programmable model, a simple variant of the single-leader routing scheme can be implemented with $O(n \log n)$ memory requirements (as opposed to $O(n^2 \log n)$ of the original single-leader scheme). The necessary modifications are as follows. Given the spanning tree, each processor labels its link upward in the tree (toward the root L) by 0. (The rest of the links may be labeled arbitrarily with a nonzero value.) A message with zero in its header that arrives at a nonroot is forwarded through the link labeled zero without shortening the message. For the first part of each message transmission, namely, the path from v to L, the processor v attaches the label zero to the message header. The root maintains the entire tree topology and forwards arriving messages to their destinations, using nonzero labels.

8.2. Constrained bubble partitions.

The second variant is related to special restrictions on the bubble partition. For some cases the bubble partition might be restricted due to other constraints, e.g., geographic constraints. For instance, one would not like to have two bubbles (say, at level 2) cover the network in the United States, such that one of the bubbles includes Japan and the other includes the United Kingdom; instead a single bubble for the United States is preferred. Our bubble partition can take into account such considerations by ignoring some of the communication links during the bubble partition in some levels.

REFERENCES

[1] Y. Afek, E. Gafni, and M. Ricklin, *Upper and lower bounds for routing schemes in dynamic networks*, in Proceedings of the 30th Symposium on Foundations of Computer Science, 1989, pp. 370–375.

[2] B. Awerbuch, *Complexity of network synchronization*, J. ACM, 32 (1985), pp. 804–823.

[3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg, *Compact distributed data structure for adaptive routing*, in Proceedings of the 21th Symposium on Theory of Computing, 1989, pp. 479–489.

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg, *Improved routing strategies with succinct tables*, J. Algorithms, 11 (1990), pp. 307–341.

[5] B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan, and S. Kutten, *Distributed control for PARIS*, in Proceedings of the 9th PODC, Quebec, 1990, pp. 145–159.

[6] B. Awerbuch and D. Peleg, *Routing with polynomial communication-space trade-off*, SIAM J. Discrete Math., 5 (1992), pp. 151–162.

[7] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks, *Adapting to asynchronous dynamic networks*, in Proceedings of the 24th Symposium on Theory of Computing, 1992, pp. 557–570.

[8] I. Cidon and I. Gopal, *PARIS: An approach to private integrated networks*, Journal of Analog and Digital Cabled Systems, 1 (1988), pp. 77–86.

[9] I. Cidon, I. Gopal, and S. Kutten, *New models and algorithms for future networks*, IEEE Trans. Inform. Theory, 41 (1995), pp. 769–780.

[10] I. Cidon, O. Gerstel, and S. Zaks, *A scalable approach to routing in ATM networks*, in Proceedings of the 8th International Workshop on Distributed Algorithms, 1994.

[11] S. Dolev and T. Herman, *Superstabilizing protocols for dynamic distributed systems*, Chicago J. Theoret. Comput. Sci., (1997), special issue on self-stabilization, article 4.

[12] O. Gerstel and S. Zaks, *The virtual path layout problem in fast networks*, in Proceedings of the 13th ACM Symposium on Principles of Distributed Computing, 1994.

[13] L. Kleinrock and F. Kamoun, *Hierarchical routing for large networks*, Computer Networks, 1 (1977), pp. 155–174.

[14] L. Kleinrock and F. Kamoun, *Optimal clustering structures for hierarchical topological design of large computer networks*, Networks, 10 (1980), pp. 221–248.

[15] D. Peleg and E. Upfal, *A tradeoff between size and efficiency for routing tables*, J. ACM, 36 (1989), pp. 510–530.

[16] M. Santoro and R. Khatib, *Labeling and implicit routing in networks*, The Computer Journal, 28 (1985), pp. 5–8.

[17] L. G. Valiant, *Universality consideration in VLSI circuits*, IEEE Trans. Comput., 30 (1981), pp. 135–140.

[18] J. van Leeuwen and R.B. Tan, *Routing with compact routing tables*, in The Book of L, G. Rozenberg and A. Salomaa, eds., Springer-Verlag, New York, 1986, pp. 259–273.

[19] J. van Leeuwen and R.B. Tan, *Interval routing*, The Computer Journal, 30 (1987), pp. 298–307.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 834–853

RANDOMLY SAMPLING MOLECULES*

LESLIE ANN GOLDBERG† AND MARK JERRUM‡

Abstract. We give a polynomial-time algorithm for the following problem: Given a degree sequence in which each degree is bounded from above by a constant, select, uniformly at random, an unlabelled connected multigraph with the given degree sequence. We also give a polynomial-time algorithm for the following related problem: Given a molecular formula, select, uniformly at random, a structural isomer having the given formula.

Key words. Pólya theory, random graphs, structural isomers

AMS subject classifications. 05C80, 20B25, 20B40, 60C05, 68Q20, 92E10

PII. S0097539797318864

1. Introduction.

In this paper, we give a polynomial-time algorithm for the following problem: Given a degree sequence in which each degree is bounded from above by a constant, select, uniformly at random (u.a.r.), an unlabelled connected multigraph with the given degree sequence. We also give a polynomial-time algorithm for the following related problem: Given a molecular formula, select, u.a.r, a *structural isomer* having the given formula. A molecular formula [18] simply gives the number of atoms of each kind that occur in a molecule. A *structural formula* [17] is a method of representing the way in which the atoms in a molecule are linked together. A *structural isomer* is a structural formula, viewed as an unlabelled multigraph in which the vertices are of several different kinds.

Some of the structural isomers corresponding to a given molecular formula are chemically irrelevant due to geometric (and other) constraints. Nevertheless, counting all of the structural isomers corresponding to a given formula is a long-standing open problem for which no practical general solution has been found [18]. Solutions do exist for certain restricted cases of chemical compounds [16, 17, 18]. Benecke et al. [2], Faulon [8], and Wieland, Kerber, and Laue [21] have developed (and coded) algorithms for *listing* all of the structural isomers corresponding to a given molecular formula. These programs typically allow the user to prescribe and forbid substructures and some of the programs deal with geometric constraints. These programs are useful if the number of structural isomers corresponding to the relevant formula is sufficiently small, so the isomers can all be listed. Faulon has argued [10] that randomly sampling structural isomers is useful for structural elucidation and molecular design in cases in which the number of isomers is too large to list them all. He [9] has developed a program for randomly sampling structural isomers and has used it for chemical applications such as a statistical study of the potential energy distribution of the isomers of C_8H_{10} and the structural elucidation of several compounds.

*Received by the editors March 26, 1997; accepted for publication (in revised form) November 5, 1998; published electronically December 15, 1999. This work was partially supported by ESPRIT Project 21726–RAND-II. A preliminary version of this paper appeared in *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, LA, January 1997.

http://www.siam.org/journals/sicomp/29-3/31886.html

†Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK (leslie@dcs.warwick.ac.uk.). The first author was partially supported by ESPRIT LTR Project 20244–ALCOM-IT, and by EPSRC grant GR/L60982.

‡Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK (mrj@dcs.ed.ac.uk).

Faulon's program applies to a realistic chemical problem including 3-D simulation of molecules and chemical analysis. However, his methods are heuristic. By contrast, we study an idealization of the problem (randomly sampling structural isomers without regard to geometric and other chemical constraints) but we achieve rigorous performance guarantees—polynomial-time computation and exactly uniform generation. Thus, we describe the first polynomial-time algorithm that uniformly samples structural isomers given a molecular formula. Our isomer-sampling algorithm is based on our algorithm for uniformly sampling unlabelled connected multigraphs with a given degree sequence.

1.1. Previous work. Uniformly sampling *labelled* multigraphs with a given bounded-degree sequence can be done in polynomial time by dynamic programming. More sophisticated techniques exist for a wider class of degree sequences—see, for example, Jerrum and Sinclair [13] and McKay and Wormald [14]—but it is not known how to apply these techniques to the problem of sampling *unlabelled* multigraphs. Nijenhuis and Wilf [15] showed how to uniformly sample unlabelled rooted trees with a specified number of vertices. This approach was extended by Wilf [22], who showed how to uniformly sample free (unrooted) trees. Their algorithms are based on an inductive definition (i.e., a generating function) for the trees. This approach will be systematized by Flajolet, Zimmerman, and Van Cutsem in a forthcoming paper [12].

More complicated techniques are required when the graphs to be sampled are not trees. Dixon and Wilf [7] were the first to give an algorithm for uniformly sampling unlabelled graphs with a specified number, n, of vertices. Their algorithm is based on Burnside's lemma. First, a permutation of the n vertices is chosen with the appropriate probability and then a graph is chosen u.a.r. from those graphs which are fixed by the chosen permutation. The choice of the permutation requires a calculation of the number of unlabelled graphs with n vertices. Wormald's algorithm [25] avoids doing this expensive calculation. Instead, it achieves a uniform distribution by restarting itself when appropriate. Wormald's method can also be used to sample r-regular graphs u.a.r. for any fixed degree $r \geq 3$. The method relies on the fact that most unlabelled r-regular graphs are rigid (without nontrivial symmetries) when $r \geq 3$. This is not true for $r = 1$ or $r = 2$.

1.2. Outline of our algorithm. Our algorithm for sampling unlabelled connected multigraphs with a given degree sequence combines the above ideas with other ideas from the field of random graphs. A natural approach to the problem is the approach of Wormald—first generate a permutation of the vertices, then generate a random connected multigraph fixed by the permutation, and finally use rejection/restarting to obtain the correct distribution. However, this approach relies heavily on the fact that many of the desired structures are rigid (so the algorithm will be likely to choose the identity permutation, which leads to a quick result without restarting). This is not the case for the set of unlabelled connected multigraphs with a given degree sequence, because the degree sequence may have many vertices of degree 1 and 2. Thus, we first reduce our problem to that of sampling unlabelled connected multigraphs with degree sequences that do not have any vertices of degree 1 or 2. Every multigraph G is associated with a unique "core" which has no vertices of degree 1 or 2. To generate G, we will generate the core of G and we will then extend the core by adding trees and chains of trees to obtain G.

For the generation of the core, we work in the configuration model of Bender and Canfield [1], Bollobás [4], and Wormald [23]. The correctness of our algorithm follows from a careful analysis of unlabelled configurations in which all block sizes are at least 3. This analysis extends Bollobás's analysis of unlabelled regular graphs [3]. Our algorithm rejects the generated core if it is not connected. The fact that this does not happen too often follows from a result of Wormald [24]. After generating the core of our random multigraph, we extend the core by adding trees and chains of trees. This part of our algorithm is based on the generating function approach mentioned earlier. An alternative approach, also based on generating functions, is to use Pólya's theorem. This approach was used to enumerate molecules with certain specified "frames" (a frame is somewhat similar to a core) by Pólya and Read [16] and others.

1.3. Outline of this paper. Section 2 sets up the machinery that we will use to reduce the general multigraph problem to the problem in which the degree sequence has no vertices of degree 1 or 2. Section 3 solves the problem when there are no vertices of degree 1 or 2. Section 4 describes the tools that we will use to lift the solution from section 3 to a solution for general degree sequences. Section 5 gives our sampling algorithm and proves that it is correct. Section 6 extends our result to the chemical problem—given a molecular formula, select, u.a.r., a structural isomer having the given formula.

2. Cores and colored configurations.

A d-rooted *multigraph* is a tuple $G = (V, E, r_0, \dots, r_{d-1})$, in which V is the *vertex set* of G, E is the *edge multiset* of G, and $r_0, \dots, r_{d-1}$ are distinct roots in V. Each element of E is an unordered pair of vertices. The expression $E(v, w)$ denotes the multiplicity of (v, w) in E. A *cycle* of G is a (closed, simple) path from a vertex v to itself that uses each edge (x, y) at most $E(x, y)$ times. (If any edge is used twice then the path is in fact of length 2.) We use the term *rooted multigraph* to refer to any d-rooted multigraph (for any d, including $d = 0$), and we use the term *multigraph* to refer to any 0-rooted multigraph. A rooted *tree* is a connected rooted multigraph (in fact a graph) with no cycles. The definitions imply that a connected unicyclic multigraph is either a connected unicyclic graph or a multigraph obtained from a tree by doubling one of its edges.

The *degree* of vertex v in a rooted multigraph $G = (V_n, E)$ is

$$d(v) = 2\,E(v, v) + \sum_{w \in V, w \neq v} E(v, w).$$

Let Δ be any fixed constant. In this paper we will be concerned with rooted multigraphs whose vertices have degree at most Δ. The *degree sequence* of such a rooted multigraph G is the sequence $\mathbf{n} = n_0, \dots, n_\Delta$, where n_i denotes the number of vertices of G with degree i. The integers $n_0, \dots, n_\Delta$ are represented in *unary,* so the input size of the degree sequence $\mathbf{n} = n_0, \dots, n_\Delta$ is $n = n_0 + \cdots + n_\Delta$. (Similarly, the input size of degree sequence $\mathbf{n}' = n'_0, \dots, n'_\Delta$ will be denoted $n' = n'_0 + \cdots + n'_\Delta$.) Let V_n be the set $\{v_1, \dots, v_n\}$, and $\mathcal{G}_\mathbf{n}$ the set containing

- every connected multigraph with degree sequence $\mathbf{n}$ and vertex set V_n that has at least two cycles, and
- every 1-rooted connected tree with degree sequence $\mathbf{n}$ and vertex set V_n, and
- every 1-rooted connected unicyclic multigraph with degree sequence $\mathbf{n}$ and vertex set V_n, in which the root is part of the cycle.

Two d-rooted multigraphs $G = (V, E, r_0, \dots, r_{d-1})$ and $G' = (V', E', r'_0, \dots, r'_{d-1})$ are said to be *isomorphic* (written $G \cong G'$) if there is a bijection π from V to V' such that, for all unordered pairs (v, w) of vertices in V, $E(v, w) = E'(\pi(v), \pi(w))$, and for all roots r_j of G, $\pi(r_j) = r'_j$. Note that if $V = V'$ then π can be viewed as a permutation of the vertices in V. Isomorphism induces an equivalence relation on $\mathcal{G}_{\mathbf{n}}$ and the equivalence classes are called isomorphism classes. We use the notation $\widetilde{\mathcal{G}}_{\mathbf{n}}$ to denote the set of isomorphism classes of $\mathcal{G}_{\mathbf{n}}$. If a rooted multigraph G is isomorphic to some $G' \in \mathcal{G}_{\mathbf{n}}$ then we use the notation $\Psi(G)$ to denote the isomorphism class of G'. For any isomorphism class $U \in \widetilde{\mathcal{G}}_{\mathbf{n}}$ we use the notation $\Psi^{-1}(U)$ to denote the lexicographically least member of U.

We consider two nondeterministic transformations which may be applied to a rooted multigraph G with vertex set V and edge multiset E. Similar transformations were used by Zhan in [26].

T_1: Choose a degree-1 vertex v other than the root of G. Remove v from V and the edge containing v from E.

T_2: If T_1 cannot be applied to G, choose a degree-2 vertex v other than the root of G such that for vertices $w \neq v$ and $x \neq v$, (v, w) and (v, x) are in E. (We allow $w = x$, but naturally insist that (v, w) and (v, x) are taken to be distinct elements from the edge multiset.) Remove v from V. Remove (v, w) and (v, x) from E and add (w, x) to E.

Note that the transformations T_1 and T_2 do not change the labels of vertices. A rooted multigraph $G \in \mathcal{G}_{\mathbf{n}}$ is *irreducible* if neither transformation T_1 nor T_2 can be applied to it.

OBSERVATION 2.1. *If $G \in \mathcal{G}_{\mathbf{n}}$ and G can be transformed into G' by T_1 or T_2 then, for some $\mathbf{n}'$ with $n' < n$, $\Psi(G') \in \widetilde{\mathcal{G}}_{\mathbf{n}'}$.*

Informally, Observation 2.1 says that the transformations T_1 and T_2 preserve the properties of being connected and of having at least two cycles.

OBSERVATION 2.2. *If $G \in \mathcal{G}_{\mathbf{n}}$ and some sequence of T_1 and T_2 transform G into G', then the sequence is of length less than n.*

We say that a degree sequence $\mathbf{n}$ is *irreducible* if any of the following applies, and that it is *degenerate* if one of the first two possibilities applies.

1. $\mathbf{n}$ describes the single-vertex multigraph. That is, $n_0 = 1$ and $n_i = 0$ for $i \neq 0$.
2. $\mathbf{n}$ describes the single-self-loop multigraph. That is, $n_2 = 1$ and $n_i = 0$ for $i \neq 2$.
3. $\mathbf{n}$ describes multigraphs without low-degree vertices. That is, $n_0 = n_1 = n_2 = 0$ and $n_i > 0$ for some $i \in [3, \ldots, \Delta]$.

We say that a rooted multigraph is *degenerate* if its degree sequence is degenerate.

OBSERVATION 2.3. *$G \in \mathcal{G}_{\mathbf{n}}$ is irreducible iff its degree sequence is irreducible.*

LEMMA 2.4. *If $G \in \mathcal{G}_{\mathbf{n}}$ and G can be transformed into irreducible rooted multigraphs G_1 and G_2 using a sequence of transformations T_1 and T_2, then $G_1 = G_2$.*

Proof. Suppose G has vertex set V and edge multiset E. We will show that if G can be transformed into distinct rooted multigraphs G_1 and G_2 by a single transformation, then there is a rooted multigraph G_3 such that a (possibly empty) sequence of transformations transforms G_1 into G_3 and another (possibly empty) sequence of transformations transforms G_2 into G_3. Thus, the transformation process is *locally confluent* [20]. As the process terminates in finite time (see Observation 2.2), it is *confluent,* which implies the result [20].

Suppose that T_1 with choice v transforms G into G_1 and T_1 with choice $w \neq v$ transforms G into G_2. Note that $(v, w) \notin E$ (otherwise, either v or w would be the root of G). Let G_3 be the result of applying T_1 to G_1 with choice w. Then T_1 with choice v transforms G_2 into G_3.

Suppose that T_2 with choice v transforms G into G_1 and T_2 with choice $w \neq v$

transforms G into G_2. Note that (v, w) does not appear twice in E (otherwise, either v or w would be the root of G). Let G_3 be the result of applying T_2 to G_1 with choice w. Then T_2 with choice v transforms G_2 into G_3. $\square$

Note that the proof of Lemma 2.4 would have failed if we had included unrooted trees and unicyclic graphs in $\mathcal{G}_{\mathbf{n}}$. It is for this reason that the definition of $\mathcal{G}_{\mathbf{n}}$ is slightly more complicated that might be expected. As we observed in Observation 2.2, a rooted multigraph G can only be transformed a finite number of times before an irreducible rooted multigraph G' is reached. G' is called the *core* of G and is denoted by $\mathrm{Core}\,(G)$. $\mathrm{Core}\,(G)$ is uniquely defined, by Lemma 2.4.

LEMMA 2.5. *If G_1 and G_2 are in $\mathcal{G}_{\mathbf{n}}$ and $\pi(G_1) = G_2$, then $\pi(\mathrm{Core}\,(G_1)) = \mathrm{Core}\,(G_2)$.*

Proof. Consider a sequence of transformations that transforms G_1 into $\mathrm{Core}\,(G_1)$. Now apply this sequence of transformations to G_2, but choose $\pi(v)$ instead of v for each vertex v that is chosen. Clearly, the result is $\mathrm{Core}\,(G_2)$. Thus, $v \notin \mathrm{Core}\,(G_1)$ exactly when $\pi(v) \notin \mathrm{Core}\,(G_2)$. $\square$

Lemma 2.5 implies that if $G_1 \cong G_2$, then $\mathrm{Core}\,(G_1) \cong \mathrm{Core}\,(G_2)$. (This property was also used by Zhan [26].) We use the notation $\widetilde{\mathcal{G}}_{\mathbf{n},\mathbf{n}'}$ to denote the set $\{U \in \widetilde{\mathcal{G}}_{\mathbf{n}} \mid \Psi(\mathrm{Core}\,(\Psi^{-1}(U))) \in \widetilde{\mathcal{G}}_{\mathbf{n}'}\}$; loosely, $\widetilde{\mathcal{G}}_{\mathbf{n},\mathbf{n}'}$ is the set of unlabelled connected multigraphs with degree sequence $\mathbf{n}$ whose cores have degree sequence $\mathbf{n}'$. We will need the following definitions. Let $\mathcal{B}$ be an (infinite) set containing one representative from each isomorphism class of the set of 1-rooted trees. A *tree-chain* with two roots is constructed from any sequence $T_1, \dots, T_k$ of 1-rooted trees as follows: If the sequence is empty, then the tree-chain consists of r_0 and r_1 and an edge between them. Otherwise, the tree-chain graph is constructed as follows: Choose distinct labels for the vertices of $T_1, \dots, T_k$. Let $r'_1, \dots, r'_k$ be the roots of $T_1, \dots, T_k$. For $i \in [1, \dots, k-1]$, add edge (r'_i, r'_{i+1}). Add the new roots r_0 and r_1 and edges (r_0, r'_1) and (r'_k, r_1). For every tree-chain G, we use the notation $R(G)$ to denote the tree-chain constructed from G by swapping r_0 and r_1. Let $\mathcal{P}$ be a set containing one representative from each isomorphism class of tree-chains. (Note that the two roots of a tree-chain are distinguishable, and any isomorphism of tree-chains must respect this distinction.)

In the following definition of "coloring," colors will encode information that is lost while forming the core. We will use colorings to recover an original rooted multigraph from its core. A *coloring* of a rooted multigraph G is a function λ that maps each vertex in the vertex set of G to an element of $\mathcal{B}$ and each edge in the edge multiset of G to an element of $\mathcal{P}$.

We will describe a function Γ that maps each colored rooted multigraph (G, λ) to an isomorphism class. $\Gamma(G, \lambda)$ is constructed as follows, where V denotes the vertex set of G and E denotes the edge multiset of G: Start with the collection of rooted trees $\{\lambda(v) \mid v \in V\} \cup \{\lambda(e) \mid e \in E\}$. Let the roots of the resulting forest be the roots of those trees that correspond to the roots of G. For each edge $(u, w) \in E$ with $u \le w$, identify root r_0 of $\lambda(u, w)$ with the root of the tree $\lambda(u)$ and root r_1 of $\lambda(u, w)$ with the root of the tree $\lambda(w)$. Relabel to avoid name clashes. Let $\Gamma(G, \lambda)$ be the isomorphism class of the resulting rooted multigraph.

Given a degree sequence $\mathbf{n}$, let $m = \frac{1}{2}\sum_i i\, n_i$ and let $B_{\mathbf{n}}$ be the lexicographically least partition of the point set $R_{\mathbf{n}} = \{1, \dots, 2m\}$ into *blocks* (subsets) such that, for each i, there are n_i blocks of size i. A d-rooted *configuration* C with degree sequence $\mathbf{n}$ [1, 4] is a tuple $(R_{\mathbf{n}}, B_{\mathbf{n}}, P, r_0, \dots, r_{d-1})$ where P is a partition of the points in $R_{\mathbf{n}}$ into *pairings,* which are unordered pairs of points and $r_0, \dots, r_{d-1}$ are distinct blocks (roots). We use the phrase *configuration* to mean a 0-rooted configuration and the phrase *rooted configuration* to mean a d-rooted configuration for any d (including $d = 0$). We let Multigraph (C) denote the rooted multigraph obtained from C by identifying the points in each block. We say that C is *connected* if Multigraph (C) is connected. If $\mathbf{n}$ is degenerate, let $\mathcal{C}_{\mathbf{n}}$ be the set containing the 1-rooted configuration with degree sequence $\mathbf{n}$. For all other irreducible degree sequences $\mathbf{n}$, let $\mathcal{C}_{\mathbf{n}}$ be the set containing all connected unrooted configurations with degree sequence $\mathbf{n}$.

A *coloring* of a rooted configuration $C = (R, B, P)$ is a function λ that maps each block $b \in B$ to an element of $\mathcal{B}$ and each pairing $p \in P$ to an element of $\mathcal{P}$. The function Γ is defined in terms of the corresponding function for rooted multigraphs. In particular, $\Gamma(C, \lambda)$ is defined to be equal to $\Gamma(\text{Multigraph}\,(C), \lambda)$. We use the notation $\mathcal{C}_{\mathbf{n},\mathbf{n}'}$ to denote the set $\{(C, \lambda) \mid C \in \mathcal{C}_{\mathbf{n}'} \text{ and } \Gamma(C, \lambda) \in \widetilde{\mathcal{G}}_{\mathbf{n},\mathbf{n}'}\}$.

For degree sequence $\mathbf{n}'$ let $K_{\mathbf{n}'}$ denote the *Kranz* group [6] operating on the points in $R_{\mathbf{n}'}$. Each permutation π in $K_{\mathbf{n}'}$ is associated with a tuple $(\pi_0, \ldots, \pi_{|B_{\mathbf{n}'}|})$ where π_0 is a permutation of blocks and π_i for $i > 0$ is a permutation of the points within block i. To apply π to $R_{\mathbf{n}'}$, one first permutes the blocks using π_0, and then permutes the points within block i (for each i) using π_i. A rooted configuration $C_1 = (R_{\mathbf{n}'}, B_{\mathbf{n}'}, P_1, r_{0,1}, \ldots, r_{d-1,1})$ is said to be *isomorphic* to a rooted configuration $C_2 = (R_{\mathbf{n}'}, B_{\mathbf{n}'}, P_2, r_{0,2}, \ldots, r_{d-1,2})$ if there is a permutation $\pi = (\pi_0, \ldots, \pi_{|B_{\mathbf{n}'}|}) \in K_{\mathbf{n}'}$ such that for all pairings $(u, v) \in P_1$ we have $(\pi(u), \pi(v))$ in P_2 and for all $j \in [0, d-1]$ we have $\pi_0(r_{j,1}) = r_{j,2}$. The colored rooted configuration $C_1' = (C_1, \lambda_1)$ is said to be *isomorphic* to the colored rooted configuration $C_2' = (C_2, \lambda_2)$ if there is an isomorphism $\pi = (\pi_0, \ldots, \pi_{|B_{\mathbf{n}'}|})$ between C_1 and C_2 such that for all blocks $b \in B_{\mathbf{n}'}$, $\lambda_1(b) = \lambda_2(\pi_0(b))$ and for all pairings $(u, v) \in P_1$, $\lambda_1(u, v) = \lambda_2(\pi(u), \pi(v))$. Note that if $(C_1, \lambda_1) \cong (C_2, \lambda_2)$ and $(C_1, \lambda_1) \in \mathcal{C}_{\mathbf{n},\mathbf{n}'}$, then $(C_2, \lambda_2) \in \mathcal{C}_{\mathbf{n},\mathbf{n}'}$. We use the notation $\widetilde{\mathcal{C}}_{\mathbf{n},\mathbf{n}'}$ to denote the set of isomorphism classes in $\mathcal{C}_{\mathbf{n},\mathbf{n}'}$. The *automorphism group* of rooted configuration C (denoted $\text{Aut}(C)$) is the group of isomorphisms between C and itself. The *colored automorphism group* of rooted colored configuration (C, λ) (denoted $\text{Aut}(C, \lambda)$) is the group of isomorphisms between (C, λ) and itself.

LEMMA 2.6. *If $G \in \mathcal{G}_{\mathbf{n}}$ and C is a rooted configuration such that* Multigraph $(C) \cong$ Core (G)*, then there is a coloring λ such that $\Psi(G) = \Gamma(C, \lambda)$.*

Proof. The process of forming core Core (G) with vertex set V and edge multiset E can be viewed as deleting a tree $h(v)$ for each node $v \in V$ and a tree-chain $h(u, v)$ for each edge $(u, v) \in E$. Suppose that $\pi(\text{Multigraph}\,(C)) = \text{Core}\,(G)$. Let λ be a coloring of Multigraph (C) defined by $\lambda(v) = h(\pi(v))$ and

$$\lambda(u, v) = \begin{cases} h(\pi(u), \pi(v)) & \text{if } \pi(u) < \pi(v), \\ R(h(\pi(u), \pi(v))) & \text{otherwise}, \end{cases}$$

where we assume the endpoints of the edge (u, v) are normalized so that $u < v$. Then $G \in \Gamma(\text{Multigraph}\,(C), \lambda)$ so $G \in \Gamma(C, \lambda)$. ☐

LEMMA 2.7. *Suppose that C_1 and C_2 are rooted configurations with irreducible degree sequence $\mathbf{n}'$. If $\Gamma(C_1, \lambda_1) = \Gamma(C_2, \lambda_2)$, then $(C_1, \lambda_1) \cong (C_2, \lambda_2)$.*

Proof. Let G_1 be the multigraph obtained in the construction of $\Gamma(C_1, \lambda_1)$. Make sure that the relabelling that occurs in the construction of G_1 does not change the labels of the vertices of Multigraph (C_1). Similarly, let G_2 be the multigraph obtained in the construction of $\Gamma(C_2, \lambda_2)$ in which the labels of the vertices of Multigraph (C_2) are unchanged. Now, by the definition of Γ, Core (G_1) = Multigraph (C_1) and Core (G_2) = Multigraph (C_2). Suppose that $\pi(G_1) = G_2$. Then $\pi(\text{Multigraph}\,(C_1)) =$ Multigraph (C_2), by Lemma 2.5. Thus, $\lambda_1(v) = \lambda_2(\pi(v))$ for any vertex v in the vertex set of Multigraph (C_1). Furthermore, for any unordered pair (u, v) of vertices, and any color ℓ, the number of copies of (u, v) in the edge multiset of Multigraph(C_1) that are colored ℓ by λ_1 is equal to the number of copies of $(\pi(u), \pi(v))$ in the edge multiset of Multigraph(C_2) that are colored ℓ by λ_2. Hence, π can be extended to an isomorphism mapping (C_1, λ_1) to (C_2, λ_2). □

COROLLARY 2.8. *There is a bijection between* $\widetilde{\mathcal{G}}_{\mathbf{n},\mathbf{n}'}$ *and* $\widetilde{\mathcal{C}}_{\mathbf{n},\mathbf{n}'}$.

Proof. The corollary follows from Lemmas 2.6 and 2.7. □

LEMMA 2.9. *Each isomorphism class in* $\widetilde{\mathcal{C}}_{\mathbf{n},\mathbf{n}'}$ *comes up* $|K_{\mathbf{n}'}|$ *times in*

$$\{(C, \lambda, \pi) \mid (C, \lambda) \in \mathcal{C}_{\mathbf{n},\mathbf{n}'} \quad \textit{and} \quad \pi \in \mathrm{Aut}(C, \lambda)\}.$$

Proof. This is a straightforward application of Burnside's Lemma [6]. □

3. Sampling irreducible multigraphs. The goal of this section is a polynomial-time algorithm that takes as input an irreducible degree sequence, $\mathbf{n}$, and samples, u.a.r., a pair (C, π), where $C \in \mathcal{C}_{\mathbf{n}}$ is a rooted connected configuration with degree sequence $\mathbf{n}$, and $\pi \in K_{\mathbf{n}}$ is an automorphism of C. This is straightforward if $\mathbf{n}$ is degenerate, so we focus on the nondegenerate case in which $n_0 = n_1 = n_2 = 0$ and $n_i > 0$ for some $i \in [3, \ldots, \Delta]$. In this case, $\mathcal{C}_{\mathbf{n}}$ is the set of connected unrooted configurations with degree sequence $\mathbf{n}$. (The configurations are unrooted because every connected multigraph in which each vertex has degree at least 3 has more than one cycle.) Thus, our goal is equivalent to generating, u.a.r., an unlabelled connected multigraph (possibly with self-loops) with degree sequence $\mathbf{n}$. So we obtain a solution to our basic problem in the special case in which all vertex degrees are at least 3. The techniques described in section 2 provide a reduction from the case of general degrees sequences to the restricted ones considered here, as we shall see in section 5.

Our approach borrows freely from Bollobás's treatment of unlabelled regular graphs [3], though we find it more convenient to work throughout with configurations in place of (multi)graphs. Recall that $2m = \sum_i in_i$. We say that a triple (s, s_2, s_3) of nonnegative integers is *legal* if $2s_2 + 3s_3 \le s \le 2m$. For every legal triple (s, s_2, s_3), let $K_{\mathbf{n}}(s, s_2, s_3)$ denote the set of permutations in $K_{\mathbf{n}}$ that contain exactly s_2 transpositions, s_3 3-cycles, and move exactly s points in all. For convenience, we introduce $s_4 = (s - 2s_2 - 3s_3)/4$; note that s_4 is not necessarily an integer. Of course, only three of the four parameters need to be specified in any situation, but the freedom to move between different triples according to context is convenient.

To generate the pair (C, π) we first select a legal triple (s, s_2, s_3), then a permutation $\pi = K_{\mathbf{n}}(s, s_2, s_3)$, and finally a configuration $C \in \mathrm{Fix}\,\pi$, where $\mathrm{Fix}\,\pi$ denotes the set of configurations with degree sequence $\mathbf{n}$ that are fixed by π. In the unlikely event that C is not connected, we return $\perp$ (see Figure 1 and Theorem 3.3). (Informally, we say that the algorithm "rejects" if $\perp$ is returned, and that it "accepts" otherwise.) For every legal triple (s, s_2, s_3), define

$$(1) \qquad F_{\mathbf{n}}(s, s_2, s_3) = \left\lceil 4 \times \frac{(2m)!}{m!\,2^m} \times \left(\frac{6s_2}{m^2}\right)^{s_2/2} \left(\frac{3s_3}{m^3}\right)^{s_3/2} \left(\frac{21s_4}{m^4}\right)^{s_4/2} \right\rceil.$$

The significance of $F_{\mathbf{n}}(s, s_2, s_3)$, as we shall see presently, is that it is a uniform upper bound on $|\,\mathrm{Fix}\,\pi|$ over all $\pi \in K_{\mathbf{n}}(s, s_2, s_3)$. Note that in (1), and throughout the proof of Lemma 3.1 below, we shall encounter expressions such as

$$\left(\frac{6s_2}{m^2}\right)^{s_2/2}$$

that are formally undefined when s_2 (or s_3 or s_4 or s) is equal to 0. The intended meaning is the limit as the variable in question (here s_2) tends to 0 from above. In all cases the upshot is that the factor concerned is 1 when the variable is 0. Note that $F_{\mathbf{n}}(s, s_2, s_3)$ is the square-root of a rational number, rounded up, and hence can be computed exactly in polynomial time. Define

$$W_{\mathbf{n}}(s, s_2, s_3) = |K_{\mathbf{n}}(s, s_2, s_3)| \times F_{\mathbf{n}}(s, s_2, s_3), \tag{2}$$

and let

$$W_{\mathbf{n}} = \sum_{s, s_2, s_3} W_{\mathbf{n}}(s, s_2, s_3), \tag{3}$$

where the sum is over all legal triples (s, s_2, s_3). Observe that $W_{\mathbf{n}}$ is a bound on the size of the set of pairs (C, π) we wish to sample from.

The proposed sampling procedure is conceptually very simple and is presented in Figure 1 towards the end of this section. Its analysis rests on the following technical lemma.

LEMMA 3.1. *With $F_{\mathbf{n}}(s, s_2, s_3)$, $W_{\mathbf{n}}(s, s_2, s_3)$, and $W_{\mathbf{n}}$ defined as above,*

1. $|\operatorname{Fix}()| = \frac{1}{4} F_{\mathbf{n}}(0, 0, 0)$*, where* () *denotes the identity permutation in* $K_{\mathbf{n}}$*;*
2. $|\operatorname{Fix} \pi| \leq F_{\mathbf{n}}(s, s_2, s_3)$*, for all* $\pi \in K_{\mathbf{n}}(s, s_2, s_3)$*;*
3. $W_{\mathbf{n}} \leq A\, W_{\mathbf{n}}(0, 0, 0)$*, where A depends only on* Δ*.*

Proof. The total number of configurations with degree sequence $\mathbf{n}$ is equal to the number of ways of choosing m pairings in a set of size $2m$. All configurations are fixed by the identity permutation, so we have

$$|\operatorname{Fix}()| = (2m-1)(2m-3)\cdots 3 \cdot 1 = \frac{(2m)!}{m!\, 2^m}.$$

Comparing the above expression with the definition of $F_{\mathbf{n}}(s, s_2, s_3)$ already gives us part 1 of the lemma.

An asymptotic expression for the number of configurations can be obtained using the usual Stirling's approximation. For our purposes, it is convenient to have absolute upper and lower bounds, which can be obtained using a more refined version of Stirling's approximation due to Robbins [19] (or see [5, p. 4]):

$$\left(\frac{2m}{\mathrm{e}}\right)^m \leq \frac{(2m)!}{m!\, 2^m} \leq \sqrt{2} \left(\frac{2m}{\mathrm{e}}\right)^m. \tag{4}$$

While we are on the subject of Stirling's formula, let us note for future reference the following slight strengthening of a familiar bound on binomial coefficients:

$$\sum_{i=0}^{t} \binom{n}{i} \leq \left(\frac{\mathrm{e}n}{t}\right)^t. \tag{5}$$

To verify this inequality, first observe that the right-hand side is monotonically increasing (viewed as a real function) for $t \in (0, 1)$ and is greater than 2^n for $t \geq n/3$. In the case $t < n/3$, the ratio between successive terms on the left-hand side exceeds 2, so the sum is bounded by the sum of a geometric series with common ratio 1/2. Thus

$$\sum_{i=0}^{t} \binom{n}{i} \leq 2\binom{n}{t} \leq \frac{2n^t}{t!} \leq n^t \left(\frac{\mathrm{e}}{t}\right)^t,$$

again using a sufficiently strong form of Stirling's approximation.

Consider $C \in \operatorname{Fix}\pi$, with $\pi \in K_{\mathbf{n}}(s, s_2, s_3)$. Each point in a 3-cycle of π must be paired with a point in a *different* 3-cycle, and the other two pairings of C incident at the first cycle are then forced. Thus $|\operatorname{Fix}\pi| = 0$ unless s_3 is even, in which case C induces a set of "higher level pairings" on the 3-cycles of π. Given these higher level pairings, there are $3^{s_3/2}$ ways to choose the pairings themselves. In all there are

$$\frac{s_3!\,3^{s_3/2}}{(s_3/2)!\,2^{s_3/2}} \le \sqrt{2}\left(\frac{3s_3}{\mathrm{e}}\right)^{s_3/2}$$

ways to choose the restriction of C to the 3-cycles of π. For transpositions, the calculation is similar, except we must now allow for the pairing to join the two points in a single transposition. But this new freedom can only blow up the number of choices by (crudely) a factor 2^{s_2}, so that there are at most

$$\sqrt{2}\left(\frac{8s_2}{\mathrm{e}}\right)^{s_2/2}$$

ways to choose the restriction of C to the transpositions of π. An optimization over the distribution of cycle lengths greater than 3 confirms that the number of ways of choosing the restriction of C to those cycles is at most

$$\sqrt{2}\left(\frac{16s_4}{\mathrm{e}}\right)^{s_4/2},$$

the bound we would obtain by assuming all the remaining cycles have length exactly 4. The number of ways of extending C to the fixed points of π is clearly bounded by

$$\sqrt{2}\left(\frac{2m}{\mathrm{e}}\right)^{(2m-s)/2} \le \sqrt{2} \times \frac{(2m)!}{m!\,2^m} \times \left(\frac{2m}{\mathrm{e}}\right)^{-s/2},$$

where we have used the other part of inequality (4). Multiplying these four bounds together, recalling $s = 2s_2 + 3s_3 + 4s_4$, yields the following upper bound on $|\operatorname{Fix}\pi|$:

$$|\operatorname{Fix}\pi| \le 4 \times \frac{(2m)!}{m!\,2^m} \times \left(\frac{2\mathrm{e}s_2}{m^2}\right)^{s_2/2} \left(\frac{3\mathrm{e}^2 s_3}{8m^3}\right)^{s_3/2} \left(\frac{\mathrm{e}^3 s_4}{m^4}\right)^{s_4/2};$$

comparing this expression with (1) defining $F_{\mathbf{n}}(s, s_2, s_3)$ gives us the second part of Lemma 3.1.

For the third part, we introduce a more refined partitioning of the group $K_{\mathbf{n}}$ according to cycle structure. For each cycle of a permutation $\pi \in K_{\mathbf{n}}$, we distinguish whether the cycle touches more than one block of $R_{\mathbf{n}}$ (type 1), or whether its action is entirely confined to a single block (type 2). We write, for example, $s_2 = s_2' + s_2''$, where s_2' is the number of type 1 transpositions and s_2'' the number of type 2 transpositions. The prime and double prime convention is applied consistently, so that we write $s = s' + s''$, where s' is the total number of points contained in all type 1 cycles and s'' the number in all type 2 cycles. Naturally, s_4' and s_4'' are defined by $s' = 2s_2' + 3s_3' + 4s_4'$ and $s'' = 2s_2'' + 3s_3'' + 4s_4''$. Denote by

$$K_{\mathbf{n}}(s, s_2', s_3'; s'', s_2'', s_3'') \subseteq K_{\mathbf{n}}(s' + s'', s_2' + s_2'', s_3' + s_3'')$$

the set of permutations with s_2' type 1 transpositions, s_2'' type 2 transpositions, and so on.

The strategy for establishing the final part of the lemma is to (i) compute an upper bound on $|K_{\mathbf{n}}(s', s_2', s_3'; s'', s_2'', s_3'')|$, (ii) optimize over the feasible region to obtain an upper bound on $|K_{\mathbf{n}}(s, s_2, s_3)|$ and hence on $W_{\mathbf{n}}(s, s_2, s_3)$, and (iii) sum over feasible s, s_2, s_3 to obtain an upper bound on $W_{\mathbf{n}}$. Our upper bound for (i) will be of the form $\kappa'(s_2', s_3', s_4') \times \kappa''(s_2'', s_3'', s_4'')$, where κ' and κ'' are bounds on the number of ways of choosing the type 1 cycles and type 2 cycles, respectively. The latter is more tractable, so we deal with it first.

Let $\pi \in K_{\mathbf{n}}(s', s_2', s_3'; s'', s_2'', s_3'')$. The number of ways of choosing the $i \le s_2''$ blocks containing the s_2'' type 2 transpositions in π is at most

$$\sum_{i=0}^{s_2''} \binom{n}{i} \le \left(\frac{\mathrm{e}n}{s_2''}\right)^{s_2''},$$

using inequality (5), and so the total number of ways of choosing the transpositions themselves is at most

$$\left(\frac{\mathrm{e}cn}{s_2''}\right)^{s_2''},$$

where $c = \Delta!$. Similar bounds hold for the longer cycles, yielding an overall bound of

$$\kappa''(s_2'', s_3'', s_4'') = \left(\frac{\mathrm{e}cn}{s_2''}\right)^{s_2''} \left(\frac{\mathrm{e}cn}{s_3''}\right)^{s_3''} \left(\frac{\mathrm{e}cn}{s_4''}\right)^{s_4''} \tag{6}$$

on the number of ways of choosing all the type 2 cycles.

We now consider the type 1 cycles of π. Denote by $B = B(s', s_2', s_3')$ the set of integer triples (b, b_2, b_3) satisfying

$$b_2, b_3 \ge 0, \quad 2b_2 + 3b_3 \le b, \quad 3b + [2s_2' - 6b_2] + [3s_3' - 9b_3] \le s', \tag{7}$$

where $[x] = \max\{x, 0\}$. The intended interpretation of (b, b_2, b_3) is as follows: b is the total number of blocks moved by π, b_2 is the number of transpositions of blocks induced by π, and b_3 is the number of 3-cycles on blocks induced by π. The significance of B is that it contains, as we shall demonstrate, all feasible choices for (b, b_2, b_3) consistent with (s', s_2', s_3').

Only the final inequality of (7) requires explanation. The weaker inequality $3b \le s'$ is easy enough to justify, as each block contains at least 3 points, so we just have to account for the other two terms. If a block contains $p \ge 4$ points, regard $p - 3$ of the points as constituting an "excess." All s_2' type 1 transpositions in π must be contained within the $2b_2$ blocks that are transposed by π. If $2s_2' > 6b_2$, then $2s_2' - 6b_2$ points in type 1 cycles must be in the excess. Similarly, if $3s_3' > 9b_3$, then $3s_3' - 9b_3$ further points in type 1 cycles must be in the excess. This justifies the final inequality in (7).

Applying a crude bound on the number of ways of choosing the type 1 cycles, given (b, b_2, b_3), we have

$$\sum_{(b,b_2,b_3)\in B} \frac{(cn)^b}{b_2!\, b_3!} \le (s'+1)^3 \max\left\{\frac{(cn)^b}{b_2!\, b_3!} : (b, b_2, b_3) \in B\right\} \tag{8}$$

as a bound on the number of ways of choosing the type 1 cycles. The right-hand side of (8) presents a small optimization problem. We claim that at the maximum, $b_2 \geq \lfloor s_2'/3 \rfloor$ (otherwise $b_2 \leftarrow b_2 + 1$ and $b \leftarrow b + 2$ leads to an improvement), and $b_3 \geq \lfloor s_3'/3 \rfloor$ (otherwise $b_3 \leftarrow b_3 + 1$ and $b \leftarrow b + 3$ does), and in any case $b \leq s'/3$. So, from (8), and using the lower bound $b! \geq (b/e)^b$, the number of ways of choosing the type 1 cycles is at most $\kappa'(s_2', s_3', s_4')$, where

$$\kappa'(s_2', s_3', s_4') = (s'+1)^5 (cn)^{s'/3} \left(\frac{3\mathrm{e}}{s_2'}\right)^{s_2'/3} \left(\frac{3\mathrm{e}}{s_3'}\right)^{s_3'/3} \tag{9}$$

$$= (s'+1)^5 \left(\frac{3\mathrm{e}c^2n^2}{s_2'}\right)^{s_2'/3} \left(\frac{3\mathrm{e}c^3n^3}{s_3'}\right)^{s_3'/3} (cn)^{4s_4'/3}. \tag{10}$$

The extra factor $(s'+1)^2$ in (9) takes account of the floor functions. Our upper bound for $|K_{\mathbf{n}}(s', s_2', s_3'; s'', s_2'', s_3'')|$ is thus

$$|K_{\mathbf{n}}(s', s_2', s_3'; s'', s_2'', s_3'')| \leq \kappa'(s_2', s_3', s_4') \times \kappa''(s_2'', s_3'', s_4''),$$

where $\kappa'(s_2', s_3', s_4')$ and $\kappa''(s_2'', s_3'', s_4'')$ are as defined in (6) and (10).

The next stage is to bound $|K_{\mathbf{n}}(s, s_2, s_3)|$. Clearly we have

$$|K_{\mathbf{n}}(s, s_2, s_3)| \leq \sum_S \kappa'(s_2', s_3', s_4') \times \kappa''(s_2'', s_3'', s_4'')$$

$$\leq (s+1)^3 \max_S \Big\{\kappa'(s_2', s_3', s_4') \times \kappa''(s_2'', s_3'', s_4'')\Big\}, \tag{11}$$

where S is the region

$$S = \Big\{(s_2', s_3', s_4', s_2'', s_3'', s_4'') \in (\mathbb{R}^+)^6 : \\ s_2' + s_2'' = s_2,\ s_3' + s_3'' = s_3, \text{ and } s_4' + s_4'' = s_4\Big\}.$$

If we bound the $(s'+1)^5$ factor in κ' simply by $(s+1)^5$, then the factors in s_2', s_2'', in s_3', s_3'', and in s_4', s_4'' appearing in the objective function of (11) separate out, and we can optimize over each pair separately.

- The s_4 factor is

$$(cn)^{4s_4'/3} \left(\frac{\mathrm{c}\mathrm{e}n}{s_4''}\right)^{s_4''} \leq (cn)^{4s_4/3}, \tag{12}$$

  since the maximum is achieved at $s_4' = s_4$ and $s_4'' = 0$.
- The s_3 factor is

$$\left(\frac{3\mathrm{e}c^3n^3}{s_3'}\right)^{s_3'/3} \left(\frac{\mathrm{e}^3c^3n^3}{(s_3'')^3}\right)^{s_3''/3} \leq 2\left(\frac{\mathrm{e}^3c^3n^3}{s_3'}\right)^{s_3'/3} \left(\frac{\mathrm{e}^3c^3n^3}{s_3''}\right)^{s_3''/3} \tag{13}$$

$$\leq 2\left(\frac{2\mathrm{e}^3c^3n^3}{s_3}\right)^{s_3/3}. \tag{14}$$

  Inequality (13) uses the fact that $x^{-x} \leq 2x^{-x/3}$ for all positive x, and inequality (14) follows from symmetry and unimodality.

- The s_2 factor is

$$\left(\frac{3\mathrm{e}c^2n^2}{s_2'}\right)^{s_2'/3}\left(\frac{\mathrm{e}^3c^3n^3}{(s_2'')^3}\right)^{s_2''/3} \le 2\left(\frac{\mathrm{e}^3c^3n^3}{(s_2')^2}\right)^{s_2'/3}\left(\frac{\mathrm{e}^3c^3n^3}{(s_2'')^2}\right)^{s_3''/3}$$
$$\le 2\left(\frac{4\mathrm{e}^3c^3n^3}{s_2^2}\right)^{s_2/3}, \tag{15}$$

by similar considerations to the previous case.

Plugging (12), (14), and (15) into (11) gives

$$|K_{\mathbf{n}}(s,s_2,s_3)| \le 4(s+1)^8\left(\frac{4\mathrm{e}^3c^3n^3}{s_2^2}\right)^{s_2/3}\left(\frac{2\mathrm{e}^3c^3n^3}{s_3}\right)^{s_3/3}(cn)^{4s_4/3},$$

which, on recalling the definitions (1) and (2) of $F_{\mathbf{n}}(s,s_2,s_3)$ and $W_{\mathbf{n}}(s,s_2,s_3)$, leads to

$$W_{\mathbf{n}}(s,s_2,s_3) \le 32(s+1)^8 \times \frac{(2m)!}{m!\,2^m} \times \left(\frac{c_2}{s_2}\right)^{s_2/6}\left(\frac{c_3s_3}{n^3}\right)^{s_3/6}\left(\frac{c_4s_4^3}{n^4}\right)^{s_4/6}, \tag{16}$$

where c_2, c_3, and c_4 are constants depending only on c and hence only on Δ. (The multiplicative factor has been boosted from 16 to 32 to allow for the ceiling function in the definition of $F_{\mathbf{n}}(s,s_2,s_3)$.)

To finish off the proof of the final part of Lemma 3.1, we merely need to sum (16) over all legal triples (s,s_2,s_3):

$$\begin{aligned}
W_{\mathbf{n}} &= \sum_{s,s_2,s_3} W_{\mathbf{n}}(s,s_2,s_3)\\
&\le 32 \times \frac{(2m)!}{m!\,2^m} \times \sum_{s_2}\left(\frac{c_2}{s_2}\right)^{s_2/6}\sum_{s_3,s_4}(2s_2+3s_3+4s_4+1)^8\left(\frac{c_3\Delta}{3n^2}\right)^{s_3/6}\left(\frac{c_4\Delta^3}{64n}\right)^{s_4/6}\\
&\approx 32 \times \frac{(2m)!}{m!\,2^m} \times \sum_{s_2}\left(\frac{c_2}{s_2}\right)^{s_2/6}(2s_2+1)^8\\
&\approx 4c' \times \frac{(2m)!}{m!\,2^m} = c'\,W_{\mathbf{n}}(0,0,0),
\end{aligned}$$

where c' depends only on c_2, hence only on Δ. ∎

LEMMA 3.2. *There is a polynomial-time algorithm for computing $|K_{\mathbf{n}}(s,s_2,s_3)|$, and hence for computing $W_{\mathbf{n}}(s,s_2,s_3)$ and $W_{\mathbf{n}}$. There is also a polynomial-time algorithm for sampling, u.a.r., a permutation from $K_{\mathbf{n}}(s,s_2,s_3)$.*

Proof. By partitioning $|K_{\mathbf{n}}(s,s_2,s_3)|$, first according to the length of the first induced cycle on blocks, and then on the exact pattern of cycles within those blocks (at most $\Delta!$ possibilities), we obtain an inductive formula for $|K_{\mathbf{n}}(s,s_2,s_3)|$. Only polynomially many distinct assignments to the parameters $\mathbf{n}$, s, s_2, and s_3 arise during the induction, so $|K_{\mathbf{n}}(s,s_2,s_3)|$ can be computed in time polynomial in n by dynamic programming. ∎

THEOREM 3.3. *The procedure* CONFIGSAMPLE *presented in Figure* 1 *is correct:* (a) *the probability that the algorithm returns a value other than $\perp$ is bounded away from* 0*;* (b) *for any configuration $C \in \mathcal{C}_{\mathbf{n}}$ with degree sequence $\mathbf{n}$, and any automorphism $\pi \in \mathrm{Aut}(C)$ of C, the probability that the pair (C,π) is returned by* CONFIG-SAMPLE *is a constant, namely $W_{\mathbf{n}}^{-1}$, independent of C and π; and* (c) *the procedure*

Step 1. If $\mathbf{n}$ is degenerate, let C be the sole member of $\mathcal{C}_{\mathbf{n}}$, choose $\pi \in \mathrm{Aut}(C)$ u.a.r., and output (C, π). Otherwise, perform Steps 2–6.
Step 2. Choose the triple (s, s_2, s_3) with probability $W_{\mathbf{n}}(s, s_2, s_3)/W_{\mathbf{n}}$.
Step 3. Choose $\pi \in K_{\mathbf{n}}(s, s_2, s_3)$, u.a.r.
Step 4. Choose $C \in \mathrm{Fix}\,\pi$, u.a.r.
Step 5. If C is not connected, output $\perp$ and halt.
Step 6. With probability $|\mathrm{Fix}\,\pi|/F_{\mathbf{n}}(s, s_2, s_3)$ output (C, π); otherwise output $\perp$.

FIG. 1. *Procedure* CONFIGSAMPLE *for sampling a pair* (C, π).

CONFIGSAMPLE *runs in time polynomial in n, provided the maximum degree Δ is bounded. Indeed, we have the following strengthening of the first part:* (d) *the probability that a pair with $\pi = ()$ is returned is bounded away from* 0.

Proof. By the second part of Lemma 3.1, the acceptance probability in Step 6 is well defined. By the third part of Lemma 3.1, the particular triple $(0, 0, 0)$ is selected in Step 2 with probability at least A^{-1}, which is bounded away from 0. This forces the identity permutation to be selected in Step 3. In this case, the probability of acceptance in Step 5 is bounded away from 0. By Bollobás (see [5, page 48] and Bender and Canfield [1], the probability that a random configuration with degree sequence $\mathbf{n}$ corresponds to a simple graph is bounded away from 0. Each simple graph corresponds to an equal number of configurations, and by Wormald [24], the probability that a simple graph with degree sequence $\mathbf{n}$ is connected is bounded away from 0.) By the first part of Lemma 3.1, we know that the pair $(C, ())$ survives Step 6 with probability $\frac{1}{4}$. This deals with (a) and its strengthening (d).

Now consider an arbitrary pair (C, π) satisfying $C \in \mathrm{Fix}\,\pi$, and suppose $\pi \in K_{\mathbf{n}}(s, s_2, s_3)$. For (C, π) to be generated, a certain well defined event must occur at each step of the algorithm. The probability that (C, π) is generated is simply the product of these four probabilities:

$$\frac{W_{\mathbf{n}}(s, s_2, s_3)}{W_{\mathbf{n}}} \times \frac{1}{|K_{\mathbf{n}}(s, s_2, s_3)|} \times \frac{1}{\mathrm{Fix}\,\pi} \times \frac{\mathrm{Fix}\,\pi}{F_{\mathbf{n}}(s, s_2, s_3)} = \frac{1}{W_{\mathbf{n}}},$$

which is clearly independent of C and π, as asserted in (b).

According to Lemma 3.2 the procedure can be implemented to run in polynomial time. ∎

4. Sampling unlabelled trees. The previous section showed how to sample, u.a.r., an unlabelled connected multigraph with a specified irreducible nondegenerate degree sequence. In section 5 we will show how to sample, u.a.r., an unlabelled connected multigraph with *any* specified degree sequence $\mathbf{n}$. Our basic strategy will be to select an irreducible degree sequence $\mathbf{n}'$ such that $n' \leq n$ with the "appropriate" probability, sample u.a.r. an unlabelled connected multigraph G' with degree sequence $\mathbf{n}'$, and finally color G' to obtain a multigraph G with degree sequence $\mathbf{n}$. (G' will be the core of G as defined in section 2.) Instead of constructing G' directly, we will select a pair (C', π) as described in section 3 such that $C' \in \mathcal{C}_{\mathbf{n}'}$ and $\pi \in \mathrm{Aut}(C')$. Then we will choose a coloring λ such that $\Gamma(C', \lambda) \in \widetilde{\mathcal{G}}_{\mathbf{n}}$. The process of choosing $\mathbf{n}'$ and λ involves counting and sampling unlabelled rooted trees. We provide the relevant tree results in this section.

The basic framework in which we will work is as follows: We will consider "structures" (trees with one or more root), each of which has a "weight" (a $(\Delta+2)$-tuple of integers). The weight of a d-rooted n-vertex tree G (which is denoted $\mu(G)$) is the tuple $(n-d, i_0, \ldots, i_\Delta)$, where i_r is the number of vertices of degree r, *excluding* the roots. We let $\mathbb{T}_1$ denote the 1-rooted tree consisting of a single vertex and $\mathbb{T}_2$ denote the 2-rooted tree consisting of a single edge. We define the following operations on trees. If G is a 1-rooted tree, we let $[d]G$ denote the 1-rooted tree obtained by taking d copies of G, identifying the roots of the d copies, and then relabelling the remaining vertices of trees $2, \ldots, d$ to avoid name clashes. If G and G' are 1-rooted trees, we let $G \times G'$ denote the tree obtained by identifying their roots and relabelling the remaining vertices of G' to avoid name clashes. If G is a tree-chain and G' is a 1-rooted tree then we let $G * G'$ denote the tree-chain constructed from G and G' as follows. Root r_1 of G is disconnected from its neighbor, v, in G, and is connected to the root of G'. The root of G' is connected to v. The vertices in G' are then relabelled to avoid name clashes. If G is a d-rooted tree and G' is a d'-rooted tree, we let $G+G'$ be the $(d+d')$-rooted tree obtained by relabelling the vertices and roots of G' to avoid name clashes.

Following Flajolet, Zimmerman, and Van Cutsem [11], we form sets of structures inductively from $\{\mathbb{T}_1\}$ and $\{\mathbb{T}_2\}$ using the following constructors.

- $S + S'$: The disjoint union of S and S'.
- For $d > 1$, $[d]S$: $\{[d]G \mid G \in S\}$.
- $S \times S'$: $\{G \times G' \mid G \in S, G' \in S'\}$.
- If all structures in S' have a single root of a given degree, $S * S'$: $\{G * G' \mid G \in S, G' \in S'\}$.
- $S \cdot S'$: $\{G + G' \mid G \in S, G' \in S'\}$.

The constructors $+$ and $\cdot$ are from [11], which also considers other constructors. We use the notation $m \cdot S$ as an abbreviation for the disjoint union of m copies of S, $S + \cdots + S$, and we use the notation S^m as an abbreviation for the Cartesian product of m copies of S, $S \cdots S$.

A *specification* of sets $S_0, \ldots, S_r$ (which is sometimes referred to as a specification of S_r) is defined to be a sequence of equations

$$m_0 \cdot S_0 = \Psi_0()\,,\ m_1 \cdot S_1 = \Psi_1(S_0)\,,\ \ldots\,,\ m_r \cdot S_r = \Psi_r(S_0, \ldots, S_{r-1})\,,$$

where $m_0, \ldots, m_r$ are positive integers and, for $i \in [0, \ldots, r]$, Ψ_i is a term built from $\{\mathbb{T}_1\}$, $\{\mathbb{T}_2\}$, and $S_0, \ldots, S_{i-1}$ using the constructors. An ℓ-specification is a specification using ℓ constructors.[1] For every set S of structures, we use the notation $S(i_0, \ldots, i_j)$ to denote the set

$$\{s \in S \mid \text{for some } i_{j+1}, \ldots, i_{\Delta+1},\ \mu(s) = (i_0, \ldots, i_{\Delta+1})\}.$$

The generating function for the set S is a function $s(x_0, \ldots, x_{\Delta+1})$ such that the coefficient of $x_0^{i_0} \cdots x_{\Delta+1}^{i_{\Delta+1}}$ in $s(x_0, \ldots, x_{\Delta+1})$, which is denoted

$$[x_0^{i_0} \cdots x_{\Delta+1}^{i_{\Delta+1}}]\, s(x_0, \ldots, x_{\Delta+1}),$$

is equal to $|S(i_0, \ldots, i_{\Delta+1})|$. The following is a straightforward extension of the results of Flajolet, Zimmerman, and Van Cutsem [11].

[1]For this, we count the constructor $[d]$ in $[d]S$ as d constructors.

THEOREM 4.1. *Given an ℓ-specification for sets $S_0,\dots,S_r$, a set of equations for the corresponding generating functions is obtained automatically by the following translation rules:*

$$\begin{array}{ll}
m\cdot S = S'+S'' & \Rightarrow s(x_0,\dots,x_{\Delta+1}) = (1/m)(s'(x_0,\dots,x_{\Delta+1})+s''(x_0,\dots,x_{\Delta+1}));\\
m\cdot S = [d]S' & \Rightarrow s(x_0,\dots,x_{\Delta+1}) = (1/m)s'(x_0^d,\dots,x_{\Delta+1}^d);\\
m\cdot S = S'\times S'' & \Rightarrow s(x_0,\dots,x_{\Delta+1}) = (1/m)(s'(x_0,\dots,x_{\Delta+1})\cdot s''(x_0,\dots,x_{\Delta+1}));\\
m\cdot S = S' * S'' & \Rightarrow s(x_0,\dots,x_{\Delta+1}) =\\
 & \qquad (x_{r+2}/m)(s'(x_0,\dots,x_{\Delta+1})\times s''(x_0,\dots,x_{\Delta+1})),\\
 & \qquad \textit{where } r \textit{ is the degree of the roots of the structures in } S'';\\
m\cdot S = S'\cdot S'' & \Rightarrow s(x_0,\dots,x_{\Delta+1}) = (1/m)(s'(x_0,\dots,x_{\Delta+1})\times s''(x_0,\dots,x_{\Delta+1})).
\end{array}$$

Furthermore, there is a polynomial p such that all coefficients

$$[x_0^{i'_0}\dots x_{\Delta+1}^{i'_{\Delta+1}}]S_j(x_0,\dots,x_{\Delta+1})$$

for which $j\in[0,\dots,r]$ and every i'_γ is at most i_γ can be computed in at most $p(i_0,\dots,i_{\Delta+1},r,\ell)$ steps and a member of $S_j(i_0,\dots,i_{\Delta+1})$ can be sampled u.a.r. in $p(i_0,\dots,i_{\Delta+1},r,\ell)$ steps.

Proof. The coefficients of $S_0,\dots,S_r$ can be computed in order and stored in a table. Sampling u.a.r. from $S_j(i_0,\dots,i_{\Delta+1})$ is accomplished as follows. If $m\cdot S_j=\{\mathbb{T}_1\}$ or $m\cdot S_j=\{\mathbb{T}_2\}$, this is straightforward. If $m\cdot S_j=S_a+S_b$, sample u.a.r. from $S_a(i_0,\dots,i_{\Delta+1})$ with probability

$$|S_a(i_0,\dots,i_{\Delta+1})|/|S_j(i_0,\dots,i_{\Delta+1})|,$$

and from $S_b(i_0,\dots,i_{\Delta+1})$ with the remaining probability. If $m\cdot S_j=[d]S_a$, then recursively sample from $S_a(i_0/d,\dots,i_{\Delta+1}/d)$ and make d copies of the resulting structure. If $m\cdot S_j=S_c\times S_d$, evaluate

$$N_{\mathbf{i}'}=|S_c(i'_0,\dots,i'_{\Delta+1})|\times|S_d(i_0-i'_0,\dots,i_{\Delta+1}-i'_{\Delta+1})|$$

for all tuples $\mathbf{i}'=(i'_0,\dots,i'_{\Delta+1})$ (other than $\mathbf{i}'=(0,\dots,0)$ and $\mathbf{i}'=(i_0,\dots,i_{\Delta+1})$) in which every i'_γ satisfies $0\le i'_\gamma\le i_\gamma$. Choose $\mathbf{i}'$ with probability $N_{\mathbf{i}'}/\sum_{\mathbf{j}'}N_{\mathbf{j}'}$. Recursively sample structures from $S_c(i'_0,\dots,i'_{\Delta+1})$ and $S_d(i_0-i'_0,\dots,i_{\Delta+1}-i'_{\Delta+1})$ and combine the structures. The cases in which $m\cdot S_j=S_c * S_d$ and $m\cdot S_j=S_c\cdot S_d$ are handled similarly, except that in the case $m\cdot S_j=S_c*S_d$ we replace i_{r+2} with $i_{r+2}-1$. $\square$

We will now use the above framework to show how to count and sample unlabelled rooted trees. Let S_r be the set containing one representative from each isomorphism class in the set of 1-rooted trees with degree-r roots. We will first give a sequence of equations to define the sets $S_r(n)$ in terms of the constructors and we will then argue that the definition is a specification (that is, the equations can be ordered in such a way that each equation depends only on sets previously defined). The set of equations is adapted from Nijenhuis and Wilf [15]. First, note that $S_0(0)=\{\mathbb{T}_1\}$ and $S_0(n)=\emptyset$ for $n\ne 0$. Furthermore, $S_r(0)=\emptyset$ for $r\ne 0$. For $n>0$, we have

$$S_1(n)=\sum_{\substack{0\le r\le\Delta-1\\ i_0+\cdots+i_\Delta=n-1}} S_r(n-1,i_0,\dots,i_r,i_{r+1}-1,i_{r+2},\dots,i_\Delta),\ \text{and}$$

$$n\cdot S_r(n)=\sum_{\substack{1\le s\le r\\ 1\le sd\le n}} d\cdot([s]S_1(d+1)\times S_{r-s}(n-sd))\ \text{ for } r>1.$$

The first equation expresses the correspondence between $n+1$-vertex trees rooted at a vertex of degree 1 and n-vertex trees with unrestricted root degree. The second equation expresses a construction for trees which has the property that each unlabelled $n+1$-vertex tree is represented n times: choose numbers s, d satisfying $1 \leq s \leq r$ and $1 \leq sd \leq n$; choose a tree τ' with $n+1-sd$ vertices, rooted at a vertex of degree $r-s$, and a tree τ'' with $d+1$ vertices rooted at a vertex of degree 1; take s copies of τ' and one copy of τ'' and identify all the roots (the identified vertices constitute the new root). Make d copies of the resulting rooted tree. Nijenhuis and Wilf [15, p. 274] give a combinatorial proof of the equation by establishing an explicit bijection between the figures enumerated by the left and right sides.

To see that the sequence of equations given is a specification, consider a 2-dimensional table. The first $r+1$ entries of column n correspond to sets $S_0(n), \ldots, S_r(n)$ (in the given order). The remaining entries correspond to the sets $[s]S_1(n/s+1)$, where $s > 1$ divides n. Note that the equation corresponding to each table entry only uses sets corresponding to smaller rows or columns of the table. Thus, we have a specification for the sets $S_r(n)$.

As we described in the beginning of this section, our algorithm in section 5 will sample u.a.r. a coloring λ of a configuration $C = (R_{\mathbf{n}'}, B_{\mathbf{n}'}, P)$ such that $\Gamma(C, \lambda) \in \widetilde{\mathcal{G}}_{\mathbf{n}}$. Recall that a coloring λ of C is a mapping from $B_{\mathbf{n}'}$ to $\mathcal{B}$ (the set of block-colors), and from P to $\mathcal{P}$ (the set of pairing-colors). The blocks in $B_{\mathbf{n}'}$ are ordered and the pairings in P can be ordered according to the ordering of the blocks, so a coloring may be specified as a sequence of n' block-colors followed by a sequence of $m' = \frac{1}{2}\sum_i in'_i$ pairing colors. Thus, the set of available colorings depends only on $\mathbf{n}$ and $\mathbf{n}'$. Let $\Lambda_{\mathbf{n},\mathbf{n}'}$ denote the set of available colorings. Given the specification for the set $S_r(n)$, we can derive specifications for $\mathcal{B}$, $\mathcal{P}$, and therefore, $\Lambda_{\mathbf{n},\mathbf{n}'}$. We start by observing that $\mathcal{B} = S_0 + \cdots + S_\Delta$. Let $\mathcal{P}_\ell$ denote the set containing one representative from each isomorphism class of length-ℓ tree-chains (thus, $\mathcal{P} = \mathcal{P}_0 + \mathcal{P}_1 + \mathcal{P}_2 + \cdots$). A specification for $\mathcal{P}$ is as follows:

$$\begin{aligned}
&\mathcal{P}_0 = \{\mathbb{T}_2\},\\
&\mathcal{P}_\ell = (\mathcal{P}_{\ell-1} * S_0) + \cdots + (\mathcal{P}_{\ell-1} * S_{\Delta-2}).
\end{aligned}$$

Let $L_{\mathbf{n}', r_0, \ldots, r_{n'}}$ denote the set of colorings of a configuration with degree sequence $\mathbf{n}'$ in which the block-coloring for the ith block is a tree whose root has degree r_i. Then the set $L_{\mathbf{n}', r_0, \ldots, r_{n'}}$ can be specified using the equation

$$L_{\mathbf{n}', r_0, \ldots, r_{n'}} = \mathcal{P}^{m'} \cdot S_{r_0} \cdots S_{r_{n'}}.$$

Finally, note that $\Lambda_{\mathbf{n},\mathbf{n}'}$ is the disjoint union, over all (polynomially many) choices of $r_0, \ldots, r_{n'}$ of $L_{\mathbf{n}', r_0, \ldots, r_{n'}}(n - n'', n_0 - n''_0, \ldots, n_\Delta - n''_\Delta)$, where

$$n''_i = \left|\{j : 1 \leq j \leq n' \text{ and } v_j + r_j = i\}\right|,$$

where v_i denotes the size of the ith block in $B_{\mathbf{n}'}$.

Thus, we have a specification for $\Lambda_{\mathbf{n},\mathbf{n}'}$. While some of the sets used in the specification such as $\mathcal{P}$ and $S_0, \ldots, S_\Delta$ are infinite, these sets are made up by taking the disjoint union of finite subsets. Accordingly, there is a polynomial-sized specification for $\Lambda_{\mathbf{n},\mathbf{n}'}$ and the following is a corollary of Theorem 4.1.

COROLLARY 4.2. *There is a polynomial p such that computing $|\Lambda_{\mathbf{n},\mathbf{n}'}|$ and sampling u.a.r. from $\Lambda_{\mathbf{n},\mathbf{n}'}$ take at most $p(n)$ steps.*

Step 1. Select a degree sequence $\mathbf{n}'$ such that $n' \leq n$ according to the probability distribution $p_{\mathbf{n}}$.

Step 2. Select a pair (C, π) using the procedure ConfigSample developed in section 3 (see Figure 1), with parameter $\mathbf{n}'$. If that procedure returns $\perp$, then output $\perp$ and halt; otherwise the result is a pair selected u.a.r. from the set of pairs (C, π), with $C \in \mathcal{C}_{\mathbf{n}'}$ and $\pi \in \mathrm{Aut}(C)$.

Step 3. Select a coloring λ u.a.r. from $\Lambda_{\mathbf{n},\mathbf{n}'}$.

Step 4. If $\pi \in \mathrm{Aut}(C, \lambda)$ then let G be any rooted multigraph in $\Gamma(C, \lambda)$; otherwise output $\perp$ and halt.

Step 5. If G has at least two cycles then output $\Psi(G)$. Otherwise, let k be the number of nonisomorphic 1-rooted multigraphs with the same vertex and edge set as G. (The choice of root is arbitrary in the case of trees, but must be on the cycle in the case of unicyclic multigraphs.) With probability k^{-1} output $\Psi(G)$; otherwise output $\perp$.

Fig. 2. *Procedure* MultiSample *for sampling u.a.r. from* $\widetilde{\mathcal{H}}_{\mathbf{n}}$.

5. Sampling unlabelled multigraphs. Let $\mathcal{H}_{\mathbf{n}}$ be the set of connected multigraphs with degree sequence $\mathbf{n}$ and vertex set V_n and let $\widetilde{\mathcal{H}}_{\mathbf{n}}$ be the set of isomorphism classes of $\mathcal{H}_{\mathbf{n}}$. In this section, we will describe a procedure MultiSample that samples u.a.r. from $\widetilde{\mathcal{H}}_{\mathbf{n}}$. The procedure will first (see Steps 1–4 of Figure 2) sample u.a.r. from $\widetilde{\mathcal{G}}_{\mathbf{n}}$ and will then use rejection to obtain a uniform distribution on $\widetilde{\mathcal{H}}_{\mathbf{n}}$.

All the components of MultiSample are now ready: section 2 introduced the machinery that we will use to reduce the general problem to the special case in which $\mathbf{n}$ is irreducible, section 3 solved the irreducible case, and section 4 described the tools that we will use to lift the solution for the irreducible case up to a solution for general degree sequences. It remains only to assemble the pieces.

Given a degree sequence $\mathbf{n}$, let the probability distribution $p_{\mathbf{n}}$ assign probability

$$p_{\mathbf{n}}(\mathbf{n}') = \frac{W_{\mathbf{n}'}\,|\Lambda_{\mathbf{n},\mathbf{n}'}|}{M\,|K_{\mathbf{n}'}|} \tag{17}$$

to irreducible degree sequences satisfying $n' \leq n$, and zero probability to the others. Here,

$$M = \sum_{\mathbf{n}'} \frac{W_{\mathbf{n}'}\,|\Lambda_{\mathbf{n},\mathbf{n}'}|}{|K_{\mathbf{n}'}|}$$

is the normalizing factor required to form a probability distribution. (The sum is over irreducible degree sequences $\mathbf{n}'$ such that $n' \leq n$. The fact that this is the right summation follows from Observation 2.3.) The significance of $p_{\mathbf{n}}$ is that it is the "correct" distribution from which to sample the degree sequence of the core. This is the final ingredient in the sampling procedure MultiSample, which is presented in Figure 2.

Lemma 5.1. *The procedure* MultiSample *presented in Figure* 2 *is correct:* (a) *the probability that the algorithm produces an output other than* $\perp$ *is* $\Omega(n^{-1})$*;* (b) *for each isomorphism class* $U \in \widetilde{\mathcal{H}}_{\mathbf{n}}$*, the probability that* U *is returned by* MultiSample *is a constant, namely* M^{-1}*, independent of* U*; and* (c) *the procedure* MultiSample *runs in time polynomial in* n*, assuming the maximum degree* Δ *is bounded.*

Proof. The procedure successfully completes Step 2 precisely if some value other than $\perp$ is returned by procedure ConfigSample; the probability of this event is bounded away from 0, by part (a) of Theorem 3.3. Indeed, part (d) of that theorem tells us more: namely that the automorphism $\pi \in \mathrm{Aut}(C)$ returned by ConfigSample is the identity with probability bounded away from 0. But if $\pi = ()$, Step 4 is guaranteed to be successful. The probability that Step 5 is successful is at least $1/n$. This completes the proof of (a).

We now proceed to compute the probability that a certain isomorphism class $U \in \widetilde{\mathcal{H}}_{\mathbf{n}}$ appears as output. We start by showing that, after Step 4, the probability that G is in any given class in $\widetilde{\mathcal{G}}_{\mathbf{n}}$ is M^{-1}. Let U be a class in $\widetilde{\mathcal{G}}_{\mathbf{n}}$. By Lemma 2.4, U has a uniquely defined core with degree sequence $\mathbf{n}'$, say. By Lemma 2.7, a condition for U to be returned in Step 4 is that the degree sequence $\mathbf{n}'$ is selected in Step 1, an event which occurs with the probability $p_{\mathbf{n}}(\mathbf{n}')$, given in (17). Now fix attention on a particular triple (C, π, λ), satisfying $C \in \mathcal{C}_{\mathbf{n}'}$ and $\pi \in \mathrm{Aut}(C, \lambda)$. By Theorem 3.3, the probability that (C, π, λ) is selected in Steps 2 and 3, conditioned on the particular choice of degree sequence $\mathbf{n}'$, is $(W_{\mathbf{n}'}\,|\Lambda_{\mathbf{n},\mathbf{n}'}|)^{-1}$. By Corollary 2.8 and Lemma 2.9, exactly $|K_{\mathbf{n}'}|$ of these triples correspond to the desired output U. Thus, again conditioned on the choice of $\mathbf{n}'$, the probability that U is returned is

$$\frac{|K_{\mathbf{n}'}|}{W_{\mathbf{n}'}\,|\Lambda_{\mathbf{n},\mathbf{n}'}|}.$$

Multiplying this expression by the probability (17) that degree sequence $\mathbf{n}'$ is selected in Step 1, we see that the overall probability that U is returned at the end of Step 4 is a constant, in fact M^{-1}. If $U \in \widetilde{\mathcal{H}}_{\mathbf{n}}$ has at least 2 cycles, it comes up once in $\widetilde{\mathcal{G}}_{\mathbf{n}}$. Otherwise, it appears k times in $\widetilde{\mathcal{G}}_{\mathbf{n}}$, where k is as in Figure 2. By accepting U only with probability k^{-1}, the output distribution after Step 5 is uniform on $\widetilde{\mathcal{H}}_{\mathbf{n}}$.

Step 1 is polynomial time by Lemma 3.2 and Corollary 4.2; Step 2 is polynomial time by Theorem 3.3; and Step 3 by Corollary 4.2. Step 4 is clearly polynomial time. Step 5 is reducible to isomorphism of 1-rooted trees, which can conveniently be decided by a recursive canonical labelling scheme: if the root is the only vertex, assign it label `()`; otherwise let $l_1, l_2, \ldots, l_t$ be the labels of the t subtrees of the root, ordered lexicographically, and assign label $(l_1 l_2 \ldots l_t)$ to the root. By induction, two 1-rooted trees are isomorphic iff their root labels are equal. Thus, we have established (c). $\square$

6. Sampling molecules. In this section we extend our results to the chemical problem—given a molecular formula, select, u.a.r., a structural isomer having the given formula. We start by extending the algorithm in section 5 so that it can be used to uniformly sample unlabelled connected *self-loop-less* multigraphs with a given degree sequence. For this we use procedure MultiSample, except that if the output has a self-loop, it is rejected. If the degree sequence of the core is nondegenerate, then the core will be a simple graph with probability bounded away from 0 (see section 3) so the probability of rejection is not too high. If the degree sequence of the core is degenerate, then the rejection probability will also be low, provided that n is sufficiently large.

The modified version of procedure MultiSample, which uniformly samples unlabelled connected self-loop-less multigraphs with a given degree sequence, solves the following problem: Given a molecular formula in which each atom has a distinct valence, select, u.a.r., a structural isomer having the given formula.[2] We can further

[2]For some chemical applications, such as applications in which valences are variable, it may be appropriate to modify the rejection phase so that some self-loops are allowed in the final output.

modify procedure MultiSample so that it can be used to uniformly sample structural isomers even when the molecular formula has different atoms with the same valence. Formally, we fix t *types* of vertices and we interpret a typed degree sequence

$$n_{0,1},\ldots,n_{0,t},\ldots,n_{\Delta,1},\ldots,n_{\Delta,t}$$

as a requirement that a multigraph have $n_{i,j}$ degree-i vertices of type j. An isomorphism between typed multigraphs must map each vertex to a vertex of the same type. Procedure MultiSample can be extended in a straightforward way to give a polynomial-time algorithm that takes as input a typed degree sequence and selects, u.a.r., an unlabelled connected multigraph with the given degree sequence. The generation of the core is as before, except that the definition of the group $K_{\mathbf{n}}$ changes since blocks can only be mapped to other blocks of the same type. The inductive specifications in section 4 must be modified slightly to account for the types, so the choice of $\mathbf{n}'$ is modified accordingly. The choice of the coloring λ is also modified slightly. The coloring of each block must have a root that has the same type as the block and a coloring of a pairing between blocks of types i and j must have roots of types i and j, respectively. Everything else is as before.

Acknowledgments. We thank Jean-Loup Faulon for proposing the problem and explaining the chemical applications. We also thank Alan Frieze for suggesting the use of the core.

REFERENCES

[1] E. A. Bender and E. R. Canfield, *The asymptotic number of labelled graphs with given degree sequences*, J. Combin. Theory Ser. A, 24 (1978), pp. 296–307.
[2] C. Benecke, R. Grund, R. Hohberger, A. Kerber, R. Laue, and T. Wieland, *MOLGEN+, a generator of connectivity isomers and stereoisomers for molecule structure elucidation*, Anal. Chem. Acta., 314 (1995), pp. 141–147.
[3] B. Bollobás, *The asymptotic number of unlabelled regular graphs*, J. London Math. Soc., 26 (1982), pp. 201–206.
[4] B. Bollobás, *Almost all regular graphs are Hamiltonian*, European J. Combin., 4 (1983), pp. 97–106.
[5] B. Bollobás, *Random Graphs*, Academic Press, New York, London, 1985.
[6] N. G. de Bruijn, *Pólya's theory of counting*, in Applied Combinatorial Mathematics, E. F. Beckenbach, ed., John Wiley & Sons, Inc., New York, 1964; see especially section 5.13.
[7] J. D. Dixon and H. S. Wilf, *The random selection of unlabeled graphs*, J. Algorithms, 4 (1983), pp. 205–213.
[8] J. L. Faulon, *On using graph-equivalent classes for the structure elucidation of large molecules*, J. Chem. Inf. Comput. Sci., 32 (1992), pp. 337–348.
[9] J. L. Faulon, *Stochastic generator of chemical structure:* 1. *Application to the structure elucidation of large molecules*, J. Chem. Inf. Comput. Sci., 34 (1994), pp. 1204–1218.
[10] J. L. Faulon, *private communication,* Sandia National Laboratories, Albuquerque, NM, 1995.
[11] P. Flajolet, P. Zimmerman, and B. Van Cutsem, *A calculus for the random generation of labelled combinatorial structures*, Theoret. Comput. Science, 132 (1994), pp. 1–35.
[12] P. Flajolet, P. Zimmerman, and B. Van Cutsem, *A calculus for the random generation of unlabelled combinatorial structures*, in preparation.
[13] M. Jerrum and A. Sinclair, *Fast uniform generation of regular graphs*, Theoret. Comput. Sci., 73 (1990), pp. 91–100.
[14] B. D. McKay and N. C. Wormald, *Uniform generation of random regular graphs of moderate degree*, J. Algorithms, 11 (1990), pp. 52–67.
[15] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, 2nd ed., Academic Press, New York, London, 1978.
[16] G. Pólya and R. C. Read, *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds*, Springer-Verlag, Berlin, New York, 1987.
[17] R. C. Read, *Some recent results in chemical enumeration*, in Graph Theory and its Applications, Springer-Verlag, Berlin, New York, 1972.

[18] R. C. Read, *The enumeration of acyclic chemical compounds*, in Chemical Applications of Graph Theory, A. T. Balaban, ed., Academic Press, New York, London, 1976.

[19] H. Robbins, *A remark on Stirling's formula*, Amer. Math. Monthly, 62 (1955), pp. 26–29.

[20] V. Sperschneider and G. Antoniou, *Logic: A Foundation for Computer Science,* Addison-Wesley, Reading, MA, 1991, chapter 13.

[21] T. Wieland, A. Kerber, and R. Laue, *Principles of the generation of constitutional and configurational isomers*, J. Chem. Inf. Comput. Sci., 36 (1996), pp. 431–439.

[22] H. S. Wilf, *The uniform selection of free trees*, J. Algorithms, 2 (1981), pp. 204–207.

[23] N. C. Wormald, *Some Problems in the Enumeration of Labelled Graphs*, Ph.D. thesis, Department of Mathematics, University of Newcastle, New South Wales, Australia, 1978.

[24] N. C. Wormald, *The asymptotic connectivity of labelled regular graphs*, J. Combin. Theory Ser. B, 31 (1981), pp. 156–167.

[25] N. C. Wormald, *Generating random unlabelled graphs*, SIAM J. Comput., 16 (1987), pp. 717–727.

[26] S. Zhan, *On Hamiltonian line graphs and connectivity*, Discrete Math., 89 (1991), pp. 89–95.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 854–879

COMPUTATIONAL SAMPLE COMPLEXITY[*]

SCOTT E. DECATUR[†], ODED GOLDREICH[‡], AND DANA RON[§]

Abstract. In a variety of PAC learning models, a trade-off between time and information seems to exist: with unlimited time, a small amount of information suffices, but with time restrictions, more information sometimes seems to be required. In addition, it has long been known that there are concept classes that can be learned in the absence of computational restrictions, but (under standard cryptographic assumptions) cannot be learned in polynomial time (regardless of sample size). Yet, these results do not answer the question of whether there are classes for which learning from a small set of examples is computationally infeasible, but becomes feasible when the learner has access to (polynomially) more examples.

To address this question, we introduce a new measure of learning complexity called *computational sample complexity* that represents the number of examples sufficient for *polynomial time* learning with respect to a fixed distribution. We then show concept classes that (under similar cryptographic assumptions) possess arbitrarily sized gaps between their standard (information-theoretic) sample complexity and their computational sample complexity. We also demonstrate such gaps for learning from membership queries and learning from noisy examples.

Key words. computational learning theory, information vs. efficient computation, pseudorandom functions, error correcting codes, wire-tap channel

AMS subject classification. 68Q15

PII. S0097539797325648

1. Introduction.

In this work, we examine the effects of computational restrictions on the number of examples needed for learning from random examples or membership queries. It has long been known that there are concept classes, containing only concepts that are implementable by "small" Boolean circuits, which can be learned in the absence of computational restrictions, yet cannot be learned (using any hypothesis class) in polynomial time (under standard cryptographic assumptions) [Val84, KV94, AK91, Kha93]. Yet, these results do not answer the question of whether there are classes for which learning from a small set of examples is computationally infeasible, but becomes feasible when the learner has access to (polynomially) more examples. Such a phenomenon seems to be present in various learning problems (described below) and we focus on this trade-off between information and computation.

The most common method of learning from examples in the PAC setting is through the use of Occam algorithms [BEHW87, BEHW89]. These are algorithms that take as input a set of labeled examples and output a concept from the target

[*]Received by the editors August 7, 1997; accepted for publication (in revised form) February 13, 1999; published electronically December 15, 1999. An extended abstract of this work appeared in *Proceedings of the 10th Workshop on Computational Learning Theory*, Nashville, TN, 1997, ACM Press, New York, NY.

http://www.siam.org/journals/sicomp/29-3/32564.html

[†]DIMACS Center, Rutgers University, Piscataway, NJ 08855. Part of the work of this author was done while at the Laboratory for Computer Science, MIT, and was supported by a grant from the Reed Foundation.

[‡]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel (oded@wisdom.weizmann.ac.il). The work of this author was done while on sabbatical leave at the Laboratory for Computer Science, MIT.

[§]Department of Electrical Engineering–Systems, Tel Aviv University, Ramat Aviv, Israel (danar@eng.tau.ac.il). The work of this author was done as a postdoctoral fellow at the Laboratory for Computer Science, MIT.

class that is consistent with the given set of examples. Blumer et al. give an upper bound on the number of examples sufficient for an Occam algorithm to provide a good hypothesis. This bound depends on the PAC accuracy and confidence parameters and the Vapnik–Chervonenkis dimension (VC-Dimension) [VC71] of the target class. The general lower bound on the number of examples required for learning [EHKV89] nearly matches (within a logarithmic factor) the upper bound for Occam algorithms. Thus, the sample complexity for learning is essentially tight when we have an algorithm that finds a consistent concept from the target class.

While Occam algorithms exist for all classes,[1] not all such algorithms are computationally efficient. Yet, for some of these classes learning is still feasible, although the *known* computationally efficient algorithms use more examples than does the Occam algorithm for the class. In these situations, computational restrictions appear to *impair* learning by requiring more data, but do not completely *preclude* learning. For example, it is NP-hard to find a k-term-DNF formula[2] consistent with a set of data labeled by a k-term-DNF formula [PV88]. The computationally efficient algorithm most commonly used for learning k-term-DNF works by finding a consistent hypothesis from a hypothesis class (kCNF) that strictly contains the target class. In using this larger class, the Occam algorithm requires a sample size dependent on n^k (the VC-Dimension of kCNF) as opposed to $k \cdot n$ (the VC-Dimension of k-term-DNF) as would be possible if the hypothesis class were k-term-DNF itself. Thus, although k-term-DNF learning is feasible, there is a gap between the sample size sufficient for learning k-term-DNF in the absence of computational restrictions and the sample size of known algorithms for computationally efficient learning.[3]

When the learner is allowed to make queries, we again see the phenomenon in which efficient learning seems to require more information than learning without such restrictions. One such example is the learning of deterministic finite automata (DFAs). Angluin's algorithm for this class [Ang87] can be viewed as drawing the standard Occam-sized sample and outputting a DFA consistent with it. But in order to efficiently find this consistent DFA, the algorithm makes many additional membership queries.

We also find computational restrictions to effect sample size when learning from examples corrupted with noise. In the absence of computational restrictions, any PAC-learnable class can also be learned in the presence of classification noise rate $\eta = 1/2 - \gamma < 1/2$ using a factor of $\Theta(1/\gamma^2)$ more examples than the noise-free case [Lai88, Tal94]. This increase is information theoretically required [Sim93, AD96]. Yet for many classes, when computation is restricted in the presence of noisy data, the sample complexity of known algorithms is increased by more than $\Theta(1/\gamma^2)$. Furthermore, this larger increase occurs even for classes that have computationally efficient noise-free Occam algorithms with optimal sample complexity, i.e., classes with no gap in their noise-free sample complexities. One very simple class exhibiting these properties is the class of monotone Boolean conjunctions.

Thus, it appears that in a variety of learning models (PAC, PAC with queries, and PAC with noise) there may exist a trade-off between time and information—with unlimited time, a small amount of information will suffice, but with time restrictions,

[1]In this work, we restrict our attention to classes of functions that can be represented by polynomially sized circuits.

[2]A k-term-DNF formula is a disjunctive normal form Boolean formula with at most constant k terms.

[3]Note that the possibility remains of there being an algorithm that learns using the "Occam number of examples" but does not learn by outputting a consistent k-term-DNF formula.

more information is required. None of the cases described above provably requires the additional examples, yet researchers have been unable to close these gaps. In this work, we describe classes of functions for which we prove (based on cryptographic assumptions) a quantitative gap between the size of the sample required for learning a class and the size of the sample required for learning it efficiently.

Our main results. We focus our attention on learning under the uniform distribution. As discussed later in this section, this seems to be an appropriate and natural choice for demonstrating sample complexity gaps. Let $\mathcal{C} = \bigcup_n C_n$ be a concept class, where each C_n consists of Boolean functions over $\{0,1\}^n$. The (*information theoretic*) *sample complexity* of $\mathcal{C}$, denoted $\mathsf{itsc}(\mathcal{C}; n, \epsilon)$, is the sample size (as a function of n and ϵ) needed for learning the class (without computational limitations) under the uniform distribution with approximation parameter ϵ and confidence 9/10. The *computational sample complexity* of $\mathcal{C}$, denoted $\mathsf{csc}(\mathcal{C}; n, \epsilon)$ is the sample size needed for learning the class in polynomial time under the uniform distribution with approximation parameter ϵ and confidence 9/10. In both cases, when the class is clear, we may omit it from the notation.

Definition 1.1 (admissible gap functions). *A function $g : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ is called* admissible *if*

1. $g(\cdot,\cdot)$ *is polynomial-time computable;*
2. (bounded growth in n) *for every $\epsilon > 0$, $1 \leq g(n,\epsilon) \leq \mathrm{poly}(n)$;*
3. (monotonicity in ϵ) *for every $\epsilon, \epsilon' > 0$ such that $\epsilon < \epsilon'$, $g(n,\epsilon) \geq g(n,\epsilon')$;*
4. (smoothness in ϵ) *there exists a constant $a \geq 1$ so that for every $\epsilon > 0$, $g(n,\epsilon) \leq a \cdot g(n, 2\epsilon)$.*

For example, the function $g(n,\epsilon) \overset{\text{def}}{=} \frac{n^d}{\epsilon^{d'}}$ is admissible for every $d, d' \geq 0$. Our main result is that any admissible function can serve as a gap between the sample complexity of some concept class and the computational sample complexity of the same class. This gives the following theorem.

Theorem 1.2 (basic model). *Let $g : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ be an admissible function, and $k : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ be of the form $k(n,\epsilon) = \frac{n^d}{\epsilon}$, where $d \geq 2$. Suppose that one-way functions exist* (cf. [Gol95]).[4] *Then there exists a concept class $\mathcal{C}$ that has sample complexity*

$$\mathsf{itsc}(n,\epsilon) = \Theta(k(n,\epsilon))$$

and computational sample complexity

$$\mathsf{csc}(n,\epsilon) = \Theta(g(n,\epsilon) \cdot k(n,\epsilon)).$$

Furthermore, $\log_2 |C_n| = O(n^{d+1})$ and each function in C_n has a $\mathrm{poly}(n)$*-size circuit.*

In the above, and in all subsequent theorems, one-way functions are merely used to construct pseudorandom functions [HILL, GGM86]. Assuming either that RSA is a one-way function or that the Diffie–Hellman key exchange is secure, one can construct pseudorandom functions in $\mathcal{NC}$ (cf. [NR95]), and so all of our "gap theorems" will follow with concept classes having $\mathcal{NC}$ circuits.

[4]Here and in all our other conditional results, the computational sample complexity lower bounds hold for $\epsilon = 1/\mathrm{poly}(n)$ under standard complexity assumptions (i.e., the existence of one-way functions.) For smaller values of ϵ these bounds hold assuming slightly nonstandard yet reasonable complexity assumptions.

	NOISE FREE		NOISE RATE $\eta = 0.25$	
	ITSC	CSC	ITSC	CSC
Item 1	$k(n, \epsilon)$	$g_1 \cdot g_2 \cdot k(n, \epsilon)$	$k(n, \epsilon)$	$g_1 \cdot g_2^2 \cdot k(n, \epsilon)$
Item 2	$k(n, \epsilon)$	$k(n, \epsilon)$	$k(n, \epsilon)$	$\frac{1}{n^2} \cdot (k(n, \epsilon))^2$

FIG. 1.1. *The results of Theorem* 1.4 *(with* Θ*-notation omitted). For general noise rate* $\eta = 0.5 - \gamma \geq 0.25$, *both ITSC and CSC increase linearly in* $1/\gamma^2$.

We next consider classification noise at rate $\eta < \frac{1}{2}$. That is, the label of each example is flipped with probability η, independently of all other examples. In this case we add $\gamma \overset{\text{def}}{=} \frac{1}{2} - \eta > 0$ as a parameter to the sample complexity functions (e.g., $\mathsf{itsc}(\mathcal{C}; n, \epsilon, \gamma)$). We obtain the next theorem.

THEOREM 1.3 (noisy model). *Let* $g : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be an admissible function and* $k : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be of the form* $k(n, \epsilon) = \frac{n^d}{\epsilon}$, *where* $d \geq 2$. *Suppose that one-way functions exist. Then there exists a concept class* $\mathcal{C}$ *which, in the presence of noise at rate* $\eta = \frac{1}{2} - \gamma$, *has sample complexity*

$$\mathsf{itsc}(n, \epsilon, \gamma) = \Theta\left(k(n, \epsilon)/\gamma^2\right)$$

and computational sample complexity

$$\mathsf{csc}(n, \epsilon, \gamma) = \Theta\left(g(n, \epsilon) \cdot k(n, \epsilon)/\gamma^2\right).$$

Furthermore, each function in C_n *has a* poly(n)*-size circuit.*

We stress that the above holds for every noise rate and in particular for the noise-free case (where $\eta = 0$ and $\gamma = 1/2$). Thus, we have for every $\gamma > 0$

$$\frac{\mathsf{csc}(n, \epsilon, \gamma)}{\mathsf{itsc}(n, \epsilon, \gamma)} \quad = \quad \Theta\left(\frac{\mathsf{csc}(n, \epsilon)}{\mathsf{itsc}(n, \epsilon)}\right) \quad = \quad \Theta\left(g(n, \epsilon)\right).$$

In particular, the computational sample complexity for moderate noise is of the same order of magnitude as in the noise-free case (i.e., $\mathsf{csc}(n, \epsilon, \frac{1}{4}) = \Theta(\mathsf{csc}(n, \epsilon))$). This stands in contrast to the following theorem in which the ratio between the two (i.e., $\frac{\mathsf{csc}(n, \epsilon, \frac{1}{4})}{\mathsf{csc}(n, \epsilon)}$) may be arbitrarily large, while $\mathsf{itsc}(n, \epsilon, \frac{1}{4}) = \Theta(\mathsf{itsc}(n, \epsilon))$ still holds (as it always does). See Figure 1.1.

THEOREM 1.4 (noise, revisited). *Let* $g_1, g_2 : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be admissible functions and* $k : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be of the form* $k(n, \epsilon) = \frac{n^d}{\epsilon}$, *where* $d \geq 2$. *Suppose that one-way functions exist. Then*

1. *there exists a concept class* $\mathcal{C}$ *which,*
   - *in the presence of noise at rate* $\eta = \frac{1}{2} - \gamma \geq \frac{1}{4}$, *has sample complexity*

     $$\mathsf{itsc}(n, \epsilon, \gamma) = \Theta(k(n, \epsilon)/\gamma^2)$$

     and computational sample complexity

     $$\mathsf{csc}(n, \epsilon, \gamma) = \Theta(g_1(n, \epsilon) \cdot (g_2(n, \epsilon))^2 \cdot k(n, \epsilon)/\gamma^2);$$

   - *whereas the noise-free complexities are*

     $$\mathsf{itsc}(n, \epsilon) = \Theta(k(n, \epsilon)) \quad \textit{and} \quad \mathsf{csc}(n, \epsilon) = \Theta(g_1(n, \epsilon) \cdot g_2(n, \epsilon) \cdot k(n, \epsilon)),$$

     respectively;

Information Theoretic	Computational Measures	
ITSC = ITQC	CQC	CSC
$k(n,\epsilon)$	$g_1 \cdot k(n,\epsilon)$	$g_1 \cdot g_2 \cdot k(n,\epsilon)$

Fig. 1.2. *The results of Theorem* 1.5 *(with* Θ*-notation omitted).*

2. *there exists a concept class* $\mathcal{C}$ *which,*
   - *in the presence of noise at rate* $\eta = \frac{1}{2} - \gamma \geq \frac{1}{4}$, *has sample complexity*

$$\mathsf{itsc}(n,\epsilon,\gamma) = \Theta(k(n,\epsilon)/\gamma^2)$$

and computational sample complexity

$$\mathsf{csc}(n,\epsilon,\gamma) = \Theta\left(\frac{(k(n,\epsilon))^2}{n^2 \cdot \gamma^2}\right);$$

   - *whereas the noise-free complexities are*

$$\mathsf{csc}(n,\epsilon) = \Theta(\mathsf{itsc}(n,\epsilon)) = \Theta(k(n,\epsilon)).$$

Furthermore, each function in C_n *has a* poly(n)*-size circuit.*

Theorem 1.3 follows as a special case of item 1 by setting $g_2 \equiv 1$. Using item 2 we get that for every $\alpha > 0$ and for every $\gamma \leq 1/4$, $\mathsf{csc}(n,\epsilon,\gamma) = \Omega(\mathsf{csc}(n,\epsilon)^{2-\alpha}/\gamma^2)$.

We now turn to learning with membership queries. The (*information theoretic*) *query complexity* of $\mathcal{C}$, denoted $\mathsf{itqc}(\mathcal{C}; n, \epsilon)$, is the number of membership queries (as a function of n and ϵ) needed for learning the class (without computational limitations) under the uniform distribution with approximation parameter ϵ and confidence 9/10. The *computational query complexity* of $\mathcal{C}$, denoted $\mathsf{cqc}(\mathcal{C}; n, \epsilon)$, is the number of queries needed for learning the class in polynomial time under the uniform distribution with approximation parameter ϵ and confidence 9/10. We obtain (see also Figure 1.2) the following.

Theorem 1.5 (query model). *Let* $g_1, g_2 : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be two admissible functions and* $k : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$ *be of the form* $k(n,\epsilon) = \frac{n^d}{\epsilon}$, *where* $d \geq 2$. *Suppose that one-way functions exist. Then there exists a concept class* $\mathcal{C}$ *which has query complexity*

$$\mathsf{itqc}(n,\epsilon) = \Theta(k(n,\epsilon)) = \Theta(\mathsf{itsc}(n,\epsilon)),$$

computational query complexity

$$\mathsf{cqc}(n,\epsilon) = \Theta(g_1(n,\epsilon) \cdot k(n,\epsilon)),$$

and computational sample complexity

$$\mathsf{csc}(n,\epsilon) = \Theta(g_1(n,\epsilon) \cdot g_2(n,\epsilon) \cdot k(n,\epsilon)).$$

Furthermore, each function in C_n *has a* poly(n)*-size circuit.*

Note that we may set $g_2 \equiv 1$ and obtain

$$\mathsf{csc}(n,\epsilon) = \Theta(\mathsf{cqc}(n,\epsilon)) = \Theta(g_1(n,\epsilon) \cdot \mathsf{itqc}(n,\epsilon))$$

(and $\mathsf{itsc}(n,\epsilon) = \Theta(\mathsf{itqc}(n,\epsilon))$).

Uniform vs. distribution-free learning. Above, we show that in a variety of settings there exists a concept class exhibiting a sample complexity gap when learning occurs with respect to the uniform distribution. We note that in all our theorems the information theoretic upper bounds hold, within a factor of n, with respect to distribution-free learning.[5] Thus, there exist concept classes for which efficient learning *under the uniform distribution* (is possible but) requires vastly larger sample sizes than the *distribution-free* information theoretic upper bound.

One may wonder whether there exists a concept class exhibiting similar sample complexity gaps with respect to every distribution. Clearly, degenerate distributions preclude such results. Alternatively, one may wonder whether, for every distribution, there exists a class that exhibits sample complexity gaps. Again, such results are precluded by degenerate distributions. Thus, we believe that an appropriate goal is to demonstrate gaps on fixed distributions, the uniform distribution being the most natural and well studied.[6]

Although the above notion of a gap cannot exist for a single concept class across all distributions, a different notion of distribution-free gap can exist. Specifically, we may consider a gap between: (1) an upper bound on the information-theoretic distribution-free sample complexity and (2) a lower bound on the distribution-free sample complexity of an efficient learner that is *tight* (i.e., has a matching upper bound). More precisely, let the (*distribution-free information theoretic*) *sample complexity* of $\mathcal{C}$, denoted $\mathcal{ITSC}(\mathcal{C}; n, \epsilon)$, be the sample size (as a function of n and ϵ) needed for learning the class (without computational limitations); and let the *distribution-free computational sample complexity* of $\mathcal{C}$, denoted $\mathcal{CSC}(\mathcal{C}; n, \epsilon)$, be the sample size needed for learning the class in polynomial time. We stress that an upper bound for any of these measures refers to all possible distributions, whereas a lower bound merely refers to one (possibly "pathological") distribution. In fact, such pathological distributions are used in the result below.

THEOREM 1.6 (distribution-free). *Let p be a polynomial so that $p(n) \geq n$, and suppose that one-way functions exist. Then there exists a concept class $\mathcal{C}$ so that $\mathcal{ITSC}(n, \epsilon) = O(n/\epsilon)$, whereas $\mathcal{CSC}(n, \epsilon) = \Theta(p(n)/\epsilon)$. Furthermore, each function in C_n has a* poly(n)*-size circuit.*

Note that the gap shown in the theorem is polynomially in n (independent of ϵ). Thus, we do not get arbitrary admissible gaps as in Theorem 1.2. We note that the computational sample complexity under the uniform distribution for this class is $\Theta(p(n) \cdot \min\{\log(1/\epsilon), \log p(n)\})$.

Techniques. The basic idea is to consider concepts that consist of two parts: The first part of the concept is determined by a pseudorandom function (cf. [GGM86]), while the second part encodes the seed of such a function. Since it is infeasible to infer a pseudorandom function, the computational-bounded learner is forced to retrieve the seed of the function which is sparsely encoded in the second part. This sparse encoding makes retrieval very costly in terms of sample complexity; yet, the computationally-*un*bounded learner is not affected by it.

The basic idea described above suffices for establishing a gap between the computational sample complexity and the information-theoretic sample complexity for a fixed ϵ. Additional ideas are required in order to have a construction that works for any ϵ, and for which one may provide tight (up to a constant factor) bounds on each

[5]In Theorem 1.5 the upper bounds hold in the distribution-free case, without any extra factor.

[6]We note that our techniques may be used to show similar sample complexity gaps on distributions other than the uniform distribution.

of the two complexities. One of these ideas is the construction of concept classes, called *equalizers*, for which the computational sample complexity upper bound is of the same order as the information-theoretic lower bound. A result of this form follows.

THEOREM 1.7 (equalizers). *Let $p(\cdot)$ be any polynomial.*

1. (noisy-sample equalizer) *There exists a concept class $\mathcal{S} = \cup_n S_n$, with concepts realizable by polynomial-size circuits, such that*

$$\mathsf{itsc}(\mathcal{S}; n, \epsilon, \gamma) \;=\; \Theta(\mathsf{csc}(\mathcal{S}; n, \epsilon, \gamma)) \;=\; \Theta(p(n)/\epsilon\gamma^2).$$

2. (query equalizer) *There exists a concept class $\mathcal{S} = \cup_n S_n$, with concepts realizable by polynomial-size circuits, such that*

$$\mathsf{itqc}(\mathcal{S}; n, \epsilon) \;=\; \Theta(\mathsf{csc}(\mathcal{S}; n, \epsilon)) \;=\; \Theta(p(n)/\epsilon).$$

Another idea used in our proofs is the introduction and utilization of a novel (probabilistic) coding scheme. In addition to the standard coding theoretic requirements, this scheme has the property that any constant fraction of the bits in the (randomized) code word yields no information about the message being encoded. We also use this coding scheme to obtain efficient constructions for the wire-tap channel problem (cf. [Wyn75])—see Proposition 2.3. We believe that this probabilistic coding scheme is of independent interest.

Organization. After introducing the cryptographic tools we shall need, we establish the separation of computational sample complexity from (information theoretic) sample complexity in the basic model. This result (i.e., Theorem 1.2) may be derived as a special case of the other results, but we chose to present a self-contained and simpler proof of the separation in the basic model: All that is needed are our new coding scheme (presented in section 2) and the basic construction (presented in section 3).

To establish separation in the noise and query models, we use a more general construction. This construction utilizes the equalizers of Theorem 1.7 (presented in section 4). The general construction itself is presented in section 5 and is used to derive Theorems 1.2 through 1.5.

Theorem 1.6 is proven in section 6. We note that this proof is much simpler than any other proof in the paper and that it can be read without reading any of the other sections.

2. Cryptographic tools.

In subsection 2.1 we review known definitions and results regarding pseudorandom functions. In subsection 2.2 we present a computationally efficient (randomized) coding scheme which on top of the standard error-correction features has a secrecy feature. Specifically, a small fraction of (the uncorrupted) bits of the code word yield no information about the message being encoded.

2.1. Pseudorandom functions.

Loosely speaking, pseudorandom functions are easy to evaluate yet look like random functions to any computationally restricted observer who may obtain their value at inputs of its choice.

DEFINITION 2.1 (pseudorandom functions [GGM86]). *Let $\ell : \mathcal{N} \mapsto \mathcal{N}$ be a polynomially-bounded length function and $\mathcal{F} = \{F_n : n \in \mathcal{N}\}$ where $F_n = \{f_\alpha : \alpha \in \{0,1\}^n\}$ is a (multi)set of 2^n Boolean functions over the domain $\{0,1\}^{\ell(n)}$. The family $\mathcal{F}$ is called a* pseudorandom family *if*

- (easy to evaluate) *there exists a polynomial-time algorithm A so that $A(\alpha, x) = f_\alpha(x)$ for every $\alpha \in \{0,1\}^*$ and $x \in \{0,1\}^{\ell(|\alpha|)}$; the string α is called the* seed *of f_α;*

- (pseudorandomness) *for every probabilistic polynomial-time oracle machine M, every positive polynomial p, and all sufficiently large n's,*

$$\left|\Pr_{f\in F_n}(M^f(1^n)=1) - \Pr_{g\in R_n}(M^g(1^n)=1)\right| < \frac{1}{p(n)},$$

where R_n denotes the set of all $(2^{2^{\ell(n)}})$ Boolean functions over the domain $\{0,1\}^{\ell(n)}$.

Pseudorandom functions exist if and only if there exist one-way functions (cf. [GGM86] and [HILL]). Pseudorandom functions that can be evaluated by $\mathcal{NC}$ circuits (one circuit per each function) exist[7], assuming either that RSA is a one-way function or that the Diffie–Hellman key exchange is secure (cf. [NR95]).

2.2. A probabilistic coding scheme. We present an efficient probabilistic encoding scheme having constant *rate* (information/code word ratio), constant (efficient) *error-correction* capability for which a (small) constant fraction of the code word bits yield no information about the plain message. Note that a scheme as described above cannot be deterministic (as each bit in a deterministic coding scheme carries information).

THEOREM 2.2. *There exist constants $c_{\rm rate}, c_{\rm err}, c_{\rm sec} < 1$ and a pair of probabilistic polynomial-time algorithms, (E, D), so that*

1. (constant rate) $|E(x)| = |x|/c_{\rm rate}$ *for all* $x \in \{0,1\}^*$;
2. (linear error correction) *for every $x \in \{0,1\}^*$ and every $e \in \{0,1\}^{|E(x)|}$ that has at most $c_{\rm err} \cdot |E(x)|$ ones,*

$$\Pr(D(E(x) \oplus e) = x) \;=\; 1,$$

where $\alpha \oplus \beta$ denotes the bit-by-bit exclusive-or of the strings α and β; Algorithm D is in fact deterministic;

3. (partial secrecy) *loosely speaking, a substring containing $c_{\rm sec} \cdot |E(x)|$ bits of $E(x)$ does not yield information on x. Namely, let I be a subset of $\{1, \ldots, |\alpha|\}$, and let α_I denote the substring of α corresponding to the bits at locations $i \in I$. Then for every $n \in \mathcal{N}$, $m = n/c_{\rm rate}$, $x, y \in \{0,1\}^n$, $I \subset \{1, \ldots, m\}$, $|I| \le c_{\rm sec} \cdot m$, and $\alpha \in \{0,1\}^{|I|}$,*

$$\Pr(E(x)_I = \alpha) \;=\; \Pr(E(y)_I = \alpha).$$

Furthermore, $E(x)_I$ is uniformly distributed over $\{0,1\}^{|I|}$.

In addition, on input x, algorithm E uses $O(|x|)$ coin tosses.

Items 1 and 2 are standard requirements of coding theory, first met by Justesen [Jus72]. What is nonstandard in the above is item 3. Indeed, item 3 is impossible if one insists that the encoding algorithm (i.e., E) be deterministic.

Proof. Using a "nice" error correcting code, the key idea is to encode the information by first augmenting it by a sufficiently long random padding. To demonstrate this idea, consider an $2n$-by-m matrix M defining a constant-rate/linear-error-correction (linear) code. That is, the string $z \in \{0,1\}^{2n}$ is encoded by $z \cdot M$. Further suppose that the submatrix defined by the last n rows of M and any $c_{\rm sec} \cdot m$ of its columns is of full rank (i.e., rank $c_{\rm sec} \cdot m$). Then, we define the following probabilistic coding, E, of strings of length n. To encode $x \in \{0,1\}^n$, we first uniformly select $y \in \{0,1\}^n$, let $z = xy$ and output $E(x) = z \cdot M$. Clearly, the error-correction features of M are

[7] Actually, these circuits can be constructed in polynomial-time given the seed of the function.

inherited by E. To see that the secrecy requirement holds consider any sequence of $c_{\rm sec} \cdot m$ bits in $E(x)$. The contents of these bit locations is the product of z by the corresponding columns in M; that is, $z \cdot M' = x \cdot A + y \cdot B$, where M' denotes the submatrix corresponding to these columns in M, and A (resp., B) is the matrix resulting by taking the first (resp., last) n rows of M'. By hypothesis B is full rank, and therefore $y \cdot B$ is uniformly distributed (and so is $z \cdot M'$ regardless of x).

What is missing in the above is a specific construction satisfying the hypothesis as well as allowing efficient decoding. Such a construction can be obtained by mimicking Justesen's construction [Jus72]. Recall that Justesen's code is obtained by composing two codes: Specifically, an *outer* linear code over an n-symbol alphabet is composed with an *inner* random linear code.[8] The outer code is obtained by viewing the message as the coefficients of a polynomial of degree $t-1$ over a field with $\approx 3t$ elements and letting the code word consist of the values of this polynomial at all field elements. Using the Berlekamp–Welch Algorithm [BW86], one can efficiently retrieve the information from a code word provided that at most t of the symbols (i.e., the values at field elements) were corrupted. We obtain a variation of this outer-code as follows: Given $x \in \{0,1\}^n$, we set $t \stackrel{\rm def}{=} 2n/\log_2(3n)$, and view x as a sequence of $\frac{t}{2}$ elements in GF$(3t)$.[9] We uniformly select $y \in \{0,1\}^n$ and view it as another sequence of $\frac{t}{2}$ elements in GF$(3t)$. We consider the degree $t-1$ polynomial defined by these t elements, where x corresponds to the high-order coefficients and y to the low-order ones. Clearly, we preserve the error-correcting features of the original outer code. Furthermore, any $t/2$ symbols of the code word yield no information about x. To see this, note that the values of these $t/2$ locations are obtained by multiplying a t-by-$t/2$ Vandermonde with the coefficients of the polynomial. We can rewrite the product as the sum of two products, the first being the product of a $t/2$-by-$t/2$ Vandermonde with the low-order coefficients. Thus, a uniform distribution on these coefficients (represented by y) yields a uniformly distributed result (regardless of x).

Next, we obtain an analogue of the inner code used in Justesen's construction. Here the aim is to encode information of length $\ell \stackrel{\rm def}{=} \log_2 3t$ (i.e., the representation of an element in GF$(3t)$) using code words of length $O(\ell)$. Hence, we do not need an efficient decoding algorithm, since maximum likelihood decoding via exhaustive search is affordable (as $2^\ell = O(t) = O(n)$). Furthermore, any code that can be specified by $\log(n)$ many bits will do (as we can try and check all possibilities in poly(n)-time), which means that we can use a randomized argument provided that it utilizes only $\log(n)$ random bits. For example, we may use a linear code specified by a (random) 2ℓ-by-4ℓ Toeplitz matrix.[10] Using a probabilistic argument one can show that with positive probability such a random matrix yields a "nice" code as required in the motivating discussion.[11] In the rest of the discussion, one such good Toeplitz matrix is fixed.

We now get to the final step in mimicking Justesen's construction: the composition of the two codes. Recall that we want to encode $x \in \{0,1\}^n$ and that using a random string $y \in \{0,1\}^n$ we have generated a sequence of $3t$ values in GF$(3t)$, denoted $x_1, \ldots, x_{3t}$, each represented by a binary string of length ℓ. (This was done by

[8] Our presentation of Justesen's code is inaccurate but suffices for our purposes.

[9] Here we assume that $3t$ is a prime power. Otherwise, we use the first prime power greater than $3t$. Clearly, this has a negligible effect on the construction.

[10] A Toeplitz matrix, $T = (t_{i,j})$, satisfies $t_{i,j} = t_{i+1,j+1}$ for every i, j.

[11] The proof uses the fact that any (nonzero) linear combination of rows (columns) in a random Toeplitz matrix is uniformly distributed.

the outer code.) Now, using the inner code (i.e., the Toeplitz matrix) and additional $3t$ random ℓ-bit strings, denoted $y_1, \ldots, y_{3t}$ we encode each of the above x_i's by a 4ℓ-bit long string. Specifically, x_i is encoded by the product of the Toeplitz matrix with the vector $x_i y_i$.

Clearly, we preserve the error-correcting features of Justesen's construction [Jus72]. The secrecy condition is shown analogously to the way in which the error correction feature is established in [Jus72]. Specifically, we consider the partition of the code word into consecutive 4ℓ-bit long subsequences corresponding to the code words of the inner code. Given a set I of locations (as in the secrecy requirement), we consider the relative locations in each subsequence, denoting the induced locations in the ith subsequence by I_i. We classify the subsequences into two categories depending on whether the size of the induced I_i is above the secrecy threshold for the inner code. By a counting argument, only a small fraction of the subsequences have I_i's above the threshold. For the rest we use the secrecy feature of the inner code to state that no information is revealed about the corresponding x_i's. Using the secrecy feature of the outer code, we conclude that no information is revealed about x. $\square$

Efficient coding for the wire-tap channel problem. Using Theorem 2.2, we obtain an efficient coding scheme for (a strong version of) the wire-tap channel problem (cf. [Wyn75]). Actually, we consider a seemingly harder version introduced by Csiszár and Körner [CK78]. To the best of our knowledge no *computationally efficient* coding scheme was presented for this problem before.[12]

PROPOSITION 2.3. *Let* (E, D) *be a coding scheme as in Theorem* 2.2 *and let* $\mathsf{bsc}_p(\alpha)$ *be a random process that represents the transmission of a string* α *over a binary symmetric channel with crossover probability*[13] p. *Then,*

1. (error correction) *for every* $x \in \{0,1\}^*$

$$\Pr(D(\mathsf{bsc}_{\frac{c_{\mathrm{err}}}{2}}(E(x))) = x) \;=\; 1 - \exp(-\Omega(|x|));$$

2. (secrecy) *for every* $x \in \{0,1\}^*$

$$\sum_{\alpha \in \{0,1\}^{|E(x)|}} \left| \Pr(\mathsf{bsc}_{\frac{1}{2} - \frac{c_{\mathrm{sec}}}{4}}(E(x)) = \alpha) \;-\; 2^{-|E(x)|} \right|$$

is exponentially vanishing in $|x|$.

Proof. Item 1 follows by observing that, with overwhelmingly high probability, the channel complements less than a c_{err} fraction of the bits of the code word. Item 2 follows by representing $\mathsf{bsc}_{(1-\gamma)/2}(\alpha)$ (where $\gamma = c_{\mathrm{sec}}/2$) as a two-stage process: In the first stage each bit of α is set (to its current value) with probability γ, independently of the other bits. In the second stage each bit that was not set in the first stage is assigned a uniformly chosen value in $\{0,1\}$. Next, we observe that, with overwhelmingly high probability, at most $2\gamma|E(x)| = c_{\mathrm{sec}}|E(x)|$ bits were set in the first stage. Suppose that this is indeed the case. Then, applying Item 3 of Theorem 2.2, the bits set in Stage 1 are uniformly distributed regardless of x, and due to Stage 2 the bits that are not set in Stage 1 are also random. $\square$

[12]We note that Maurer has shown that this version of the problem can be reduced to the original one by using bidirectional communication [Mau91]. Crépeau (private communication, April 1997) has informed us that, using the techniques in [BBCM95, CM97], one may obtain an alternative efficient solution to the original wire-tap channel problem, again by using bidirectional communication.

[13]The *crossover probability* is the probability that a bit is complemented in the transmission process.

Remark 2.1. The above proof can be easily adapted to assert that, with overwhelmingly high probability, no information about x is revealed when obtaining both $\frac{c_{\rm sec}}{2} \cdot |E(x)|$ of the bits of $E(x)$ as well as the entire $\mathsf{bsc}_{\frac{1}{2}-\frac{c_{\rm sec}}{8}}(E(x))$.

3. Proof of Theorem 1.2. We start by describing a construction that satisfies the gap requirement of Theorem 1.2 for a fixed ϵ, say $\epsilon = 0.1$. That is, we only show that there exists a concept class $\mathcal{C}$ which for $\epsilon = 0.1$ has sample complexity $\mathsf{itsc}(n, \epsilon) = \Theta(k(n, \epsilon))$ and computational sample complexity $\mathsf{csc}(n, \epsilon) = \Theta(g(n, \epsilon) \cdot k(n, \epsilon))$. The construction is later generalized to handle variable ϵ.

3.1. Motivation: Construction for constant ϵ. We view a function $f \in C_n$ as an array of 2^n bits. This array is divided into the following three (consecutive) *slices* which have sizes 2^{n-1}, 2^{n-2}, and 2^{n-2}, respectively.

Slice I: This slice, called the *pseudorandom slice*, is determined by a pseudorandom function $f_s : \{0,1\}^{n-1} \to \{0,1\}$, where the seed s is of length n. (See subsection 2.1.)

Slice II: This slice, called the *seed encoder*, is determined by the above-mentioned seed s and an additional string r of length $O(n)$. More precisely, first we employ the probabilistic encoding scheme of subsection 2.2 to encode the message s using r as the randomness required by the scheme. The result is a code word of length $m \stackrel{\rm def}{=} O(n)$. Next we repeat each bit of the code word in $\frac{2^{n-2}}{g(n,0.1)\cdot k(n,0.1)}$ specified locations. All other locations in this slice are set to zero.

Slice III: This slice, called the *sample equalizer*, is determined by a binary string u of length $k(n, 0.1)$. The slice consists entirely of $k(n, 0.1)$ blocks of equal length, each repeating the corresponding bit of u. The purpose of this slice is to dominate the (information theoretic) sample complexity and allow us to easily derive tight bounds on it.

Information theoretic bounds. Applying Occam's Razor [BEHW87] to the class $\mathcal{C}$, we obtain $\mathsf{itsc}(\mathcal{C}; n, 0.1) = O(\log |C_n|) = O(n+O(n)+k(n, 0.1)) = O(k(n, 0.1))$, where the last equality is due to $k(n, 0.1) > n$. On the other hand, in order to learn a function in the class with error at most 0.1, it is necessary to learn Slice III with error at most 0.4. Thus, by virtue of Slice III alone, we have

$$\mathsf{itsc}(\mathcal{C}; n, 0.1) \geq \mathsf{itsc}(\text{Slice III}; n, 0.4) \geq 0.2 \cdot k(n, 0.1)$$

where the last inequality is due to the fact that learning a random string with error ϵ requires obtaining at least $1 - 2\epsilon$ of its bits. Thus, we have established the desired information-theoretic bounds. We now turn to analyzing the computational sample complexity.

Computational lower bound. The computationally bounded learner cannot learn Slice I from examples (or even queries) in the slice. Still, it must learn Slice I with error at most 0.2. Hence, the role of Slice I is to force the computationally bounded learner to obtain the function's seed from Slice II. By Item 3 of Theorem 2.2, in order to attain any information (from Slice II) regarding the seed, the learner must obtain $\Omega(n)$ bits of the code word (residing in Slice II). By Item 1 of Theorem 2.2, this means obtaining a constant fraction of the bits of the code word. Recall that the probability of getting any bit in the code word is $\frac{O(n)}{g(n,0.1)\cdot k(n,0.1)}$. Therefore, by a Chernoff bound, for every fraction $\alpha < 1$ there exists a constant $\beta < 1$ such that the probability of obtaining a fraction α of the code word given $(\beta \cdot g(n, 0.1) \cdot k(n, 0.1))$ examples is exponentially small. Thus, $\mathsf{csc}(\mathcal{C}; n, 0.1) = \Omega(g(n, 0.1) \cdot k(n, 0.1))$.

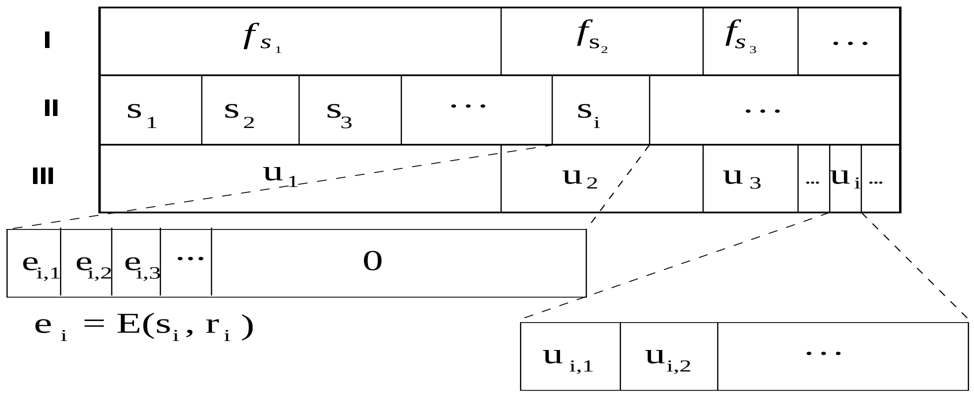


FIG. 3.1. *Construction for Theorem* 1.2.

Computational upper bound. By Chernoff bound a sample of $O(g(n, 0.1) \cdot k(n, 0.1))$ examples contains, with overwhelmingly high probability, an occurrence of each bit of the code word of the seed. Thus, by (a special case of) Item 2 of Theorem 2.2, the learner can efficiently retrieve the seed and so derive all of Slices I and II of the concept. However, by $g(n, 0.1) \geq 1$, the above sample will also allow obtaining (with high probability) all but at most a 0.1 fraction of the bits in Slice III, and thus $\mathsf{csc}(\mathcal{C}; n, 0.1) = O(g(n, 0.1) \cdot k(n, 0.1))$.

3.2. General construction—variable ϵ. We adopt the basic structure of the construction above, except that each of the three slices is further subpartitioned into *blocks*. Specifically, each slice has $t \stackrel{\text{def}}{=} n - \log_2 O(n)$ (consecutive) blocks, so that each block corresponds to a different possible value of $\epsilon = 2^{-i}$, for $i = 1, \ldots, t$. We start with a detailed description of each of the three slices (see Figure 3.1).

Slice I: *The pseudorandom part.* The ith block has size 2^{n-1-i} and is determined by a pseudorandom function $f_{s_i} : \{0,1\}^{n-1-i} \to \{0,1\}$, where the seed s_i is of length n. Note that the blocks are shrinking in size and that they are all pseudorandom.

Slice II: *The seeds encoder.* Here the blocks are of equal size $B \stackrel{\text{def}}{=} \frac{1}{t} \cdot 2^{n-2}$. The ith block encodes the ith seed, s_i, using an additional string r_i of length $O(n)$. Let e_i be the code word obtained by employing the probabilistic encoding scheme (of Theorem 2.2) on input s_i using randomness r_i. Recall that $m \stackrel{\text{def}}{=} |e_i| = O(n)$. The ith block is further divided into m (consecutive) *information fields*, each of size $\frac{2^{n-2}}{g(n,2^{-i}) \cdot k(n,2^{-i})}$, and an additional *empty field* (of size $B - \frac{m \cdot 2^{n-2}}{g(n,2^{-i}) \cdot k(n,2^{-i})}$). Thus the m information fields have relative density $\frac{m \cdot t}{g(n,2^{-i}) \cdot k(n,2^{-i})} = \Theta(\frac{n^2}{g(n,2^{-i}) \cdot k(n,2^{-i})})$ with respect to the total size B of the block. All bits in the jth information field are set to the value of the jth bit of e_i, and all bits in the empty field are set to zero.

Slice III: The sample equalizer. The ith block has size $B_i \stackrel{\text{def}}{=} 2^{n-2-i}$ and is determined by a binary string u_i of length $K \stackrel{\text{def}}{=} 2^{-i} \cdot k(n, 2^{-i}) = n^d$. The ith block is further divided into K (consecutive) subblocks, each of size $\frac{B_i}{K}$. All bits in the jth subblock are set to the value of the jth bit of u_i. Note that the blocks are shrinking in size and that for each block it is possible to obtain most bits in the block by viewing $\Theta(K)$ random examples residing in it.

We first observe that Slice III above gives rise to a concept class for which tight bounds of the form $k(n,\epsilon) = \mathrm{poly}(n)/\epsilon$, on the information theoretic and computational sample complexities, can be given.

PROPOSITION 3.1. *For* Slice III *described above we have*

1. $\mathsf{itsc}(\text{Slice III}; n, 4\epsilon) = \Omega(k(n,\epsilon))$;
2. $\mathsf{csc}(\text{Slice III}; n, \epsilon) = O(k(n,\epsilon))$.

Proof. Item 1. Let $i \stackrel{\text{def}}{=} \lfloor \log_2(1/8\epsilon) \rfloor$. In order to learn Slice III with error at most 4ϵ, one must learn the ith block reasonably well. Specifically, one must obtain examples from at least half of the K subblocks of the ith block, and hence must have at least $K/2$ examples in the ith block. By Chernoff bound, this implies that the total number of random samples must be at least $2^i \cdot \frac{K}{4} = 2^{i-2}\epsilon \cdot k(n,\epsilon) > 2^{-7} \cdot k(n,\epsilon)$. Item 1 follows.

Item 2. On the other hand, we consider the fraction of the third slice that is determined (with constant probability close to 1) given a sample of $16 \cdot k(n,\epsilon) = 16K/\epsilon$ random examples. It suffices to show that the total area left undetermined in the first $\ell \stackrel{\text{def}}{=} \lceil \log_2(4/\epsilon) \rceil$ blocks is at most an $\epsilon/2$ fraction of the total domain (since the remaining blocks cover at most an $\epsilon/2$ fraction of the total). Fixing any $i \leq \ell$, we consider the expected number of subblocks determined in the ith blocks (out of the total K subblocks). A subblock is determined if and only if we obtain a sample in it, and the probability for the latter event not to occur in $16K/\epsilon$ trials is

$$\left(1 - \frac{2^{-i}}{K}\right)^{16K/\epsilon} = \exp\left(-\frac{16}{2^i \cdot \epsilon}\right). \tag{3.1}$$

It follows that the expected fraction of bits that are not determined in the first ℓ blocks is bounded above by

$$\sum_{i=1}^{\ell} 2^{-i} \cdot 2^{-\frac{16}{2^i \epsilon}} = \sum_{j=0}^{\ell-1} 2^{-(\ell-j)} \cdot 2^{-\frac{16}{2^{\ell-j} \cdot 2^{-\ell+2}}} \tag{3.2}$$

$$= 2^{-\ell} \cdot \sum_{j=0}^{\ell-1} 2^j \cdot 2^{-2^{j+2}} \tag{3.3}$$

$$< \frac{\epsilon}{4} \cdot 2^{-4} \sum_{j=0}^{\infty} 2^{-j} \tag{3.4}$$

$$< \frac{\epsilon}{32} \tag{3.5}$$

where (3.4) follows from the fact that $\forall j \geq 0,\ 2^{j-4\cdot 2^j} \leq 2^{-4} \cdot 2^{-j}$. Item 2 follows. □

LEMMA 3.1. *The concept class described above has* (information theoretic) *sample complexity* $\mathsf{itsc}(n,\epsilon) = \Theta(k(n,\epsilon))$.

Proof. Clearly, it suffices to learn each of the three slices with error ϵ. Applying Occam's Razor [BEHW87] to Slices I and II of the class, we obtain

$$\mathsf{itsc}(\text{Slices I and II}; n, \epsilon) = O(n^2/\epsilon) = O(k(n,\epsilon)),$$

where the last equality is due to the hypothesis regarding the function $k(\cdot,\cdot)$ (i.e., that it is $\Omega(n^2/\epsilon)$). Using Item 2 of Proposition 3.1, we obtain $\mathsf{itsc}(\text{Slice III}; n, \epsilon) \leq \mathsf{csc}(\text{Slice III}; n, \epsilon) = O(k(n,\epsilon))$, and $\mathsf{itsc}(n,\epsilon) = O(k(n,\epsilon))$ follows. (Note that in order to obtain the desired tight bound we cannot simply apply Occam's Razor to Slice III since it is determined by roughly $2n \cdot K = 2n \cdot \epsilon \cdot k(n, \epsilon)$ bits.)

On the other hand, in order to learn a function in the class, we must learn Slice III with error at most 4ϵ. Using Item 1 of Proposition 3.1, we obtain $\mathsf{itsc}(\text{Slice III}; n, 4\epsilon) = \Omega(k(n, 4\epsilon))$, and $\mathsf{itsc}(n, \epsilon) = \Omega(k(n, 4\epsilon)) = \Omega(k(n, \epsilon))$ follows. □

LEMMA 3.2. *The concept class described above has* computational *sample complexity* $\mathsf{csc}(n, \epsilon) = \Theta(g(n, \epsilon) \cdot k(n, \epsilon))$.

Proof. Using Item 2 of Proposition 3.1, we have that $\mathsf{csc}(\text{Slice III}; n, \epsilon) = O(k(n, \epsilon))$, and using $g(n, \epsilon) \geq 1$ we infer that Slice III can be efficiently learned with ϵ error given a sample of size $O(g(n, \epsilon) \cdot k(n, \epsilon))$. Next we show that such a sample suffices for learning Slice I and Slice II as well. Since the information fields in the ith block of Slice II have density bounded above by 2^{-i}, in order to learn Slice II with error at most ϵ, it suffices to learn the first $\ell \overset{\text{def}}{=} \lceil \log_2(4/\epsilon) \rceil$ with error at most $\epsilon/2$. However, such an approximation might not suffice for learning Slice I sufficiently well. Nonetheless, we next show that a sample of size $O(g(n, \epsilon) \cdot k(n, \epsilon))$ suffices for efficiently determining (*exactly*) the first ℓ seeds (residing in the first ℓ blocks of Slice II) and thus determining the first ℓ blocks of Slice I.

Let $i \leq \ell$ and consider the m information fields in the ith block of the seed encoder. Suppose that for some constant c' (to be specified), we have $2c' \cdot (g(n, \epsilon) \cdot k(n, \epsilon))$ random examples. Then the expected number of examples residing in the information fields of the ith block is

$$2c' \cdot (g(n, \epsilon) \cdot k(n, \epsilon)) \cdot \frac{m}{g(n, 2^{-i}) \cdot k(n, 2^{-i})} \ . \tag{3.6}$$

Since $g(n, \epsilon) \cdot k(n, \epsilon) = \Omega(g(n, 2^{-i})k(n, 2^{-i}))$, for a suitable constant c'' (that depends on c' and on the constants in the $\Omega(\cdot)$ notation), this expected number is $2c''m$. With overwhelmingly high probability (i.e., $1 - \exp(-\Omega(m))$), there are at least $c''m$ examples in the information fields of the ith block (for every $i \leq \ell$). We set c' so that c'' will be such that, with overwhelmingly high probability, such a sample will miss at most $c_{\text{err}} \cdot m$ of these fields, where c_{err} is the constant in Item 2 of Theorem 2.2 (e.g., $c'' = 2/c_{\text{err}}$ will suffice). Invoking Item 2 of Theorem 2.2 (for the special case in which there are no errors but part of the code word is missing), we obtain the seed encoded in the ith block of Slice II. Since the probability of failure on a particular block is negligible, with very high probability we obtain the seeds in all the first ℓ blocks of Slice II. This concludes the proof of the upper bound.

We now turn to the lower bound and let $i \overset{\text{def}}{=} \lfloor \log_2(1/4\epsilon) \rfloor$. Considering the ith block of Slice I, we will show that too small a sample does not allow information regarding the ith seed to be obtained from Slice II. This will lead to the failure of the computational bounded learner, since without such information the ith block of Slice I looks totally random (to this learner). Specifically, let c_{sec} be the constant in Item 3 of Theorem 2.2, and let $c' = 2/c_{\text{sec}}$. Suppose the learner is given

$$c'' \cdot g(n, \epsilon) \cdot k(n, \epsilon) \quad < \quad c' \cdot g(n, 2^{-i})k(n, 2^{-i}) \tag{3.7}$$

random examples. (The constant c'' is such that the last inequality holds.) Then, using a Chernoff bound we infer that, with overwhelmingly high probability, we will have at most

$$2c' \cdot (g(n, 2^{-i}) \cdot k(n, 2^{-i})) \cdot \frac{m}{g(n, 2^{-i}) \cdot k(n, 2^{-i})} \quad = \quad 2c'm \quad = \quad c_{\text{sec}}m \tag{3.8}$$

random examples in the information fields of the ith block of Slice II. Invoking Item 3 of Theorem 2.2, this yields no information regarding the seed encoded in the ith block of Slice II. Thus, the computationally bounded learner cannot predict any unseen point in the ith block of Slice I better than at random. This means that its error is greater than allowed (i.e., $\geq 2^{-i-1} > \epsilon$). Thus, $\mathsf{csc}(n,\epsilon) > c'' \cdot g(n,\epsilon)k(n,\epsilon)$, where c'' is a constant as required. □

4. The two equalizers (proof of Theorem 1.7). In this section we show that (1) the sample equalizer (i.e., Slice III) described in section 3.2 can be used to prove the first item in Theorem 1.7 (noisy-sample equalizer); (2) another construction, based on *interval functions,* can be used to prove the second item of the theorem (query equalizer).

4.1. Noisy-sample equalizer (item 1 of Theorem 1.7). As noted above, we use Slice III of the construction in section 3.2. Here we think of a concept in the class $\mathcal{S} = \cup_n \mathcal{S}_n$ as being an array of size 2^n (as opposed to 2^{n-2} when it serves as the third slice of a concept). The number of subblocks in each of the $t = n - O(\log(n))$ blocks is $p(n)$. The proof follows the structure of the proof of Proposition 3.1.

LEMMA 4.1. $\mathsf{itsc}(\mathcal{S}, n, \epsilon, \gamma) = \Omega\left(\frac{p(n)}{\epsilon\gamma^2}\right)$.

Proof. Let $i \stackrel{\text{def}}{=} \lfloor \log_2(1/2\epsilon) \rfloor$. In order to learn a function in the class $\mathcal{S}_n$, one must learn the ith block reasonably well. Specifically, one must obtain sufficiently many examples from the ith block or else the information we obtain on the bits residing in this block is too small. In particular, we claim that we must have at least $\Omega(p(n)/\gamma^2)$ examples in the ith block. We prove the claim by noting that it corresponds to the classical information theoretic measure of the mutual information (cf., [CT91]) that these samples provide about the string residing in this block. Each example provides information on a single bit residing in the block and the amount of information is merely the capacity of a binary symmetric channel with crossover probability $\eta = \frac{1}{2} - \gamma$. That is, each example yields $1 - H_2(\eta)$ bits of information, where H_2 is the binary entropy function, which satisfies $1 - H_2(0.5 - \gamma) \approx \Theta(\gamma^2)$. The claim follows by additivity of information.

Let $c > 0$ be a constant so that we must have at least $p(n)/c\gamma^2$ examples in the ith block. Then, in order to have (with high probability) at least $p(n)/c\gamma^2$ examples in the ith block, the total number of random samples must be at least $2^i \cdot \frac{p(n)}{2c\gamma^2} = \Omega\left(\frac{p(n)}{\epsilon\gamma^2}\right)$. □

LEMMA 4.2. $\mathsf{csc}(\mathcal{S}, n, \epsilon, \gamma) = O\left(\frac{p(n)}{\epsilon\gamma^2}\right)$.

Proof. We consider the fraction of a concept in S_n that can be correctly inferred by a sample of $O(p(n)/\epsilon\gamma^2)$ random examples. It suffices to show that the total area left incorrectly inferred in the first $\ell \stackrel{\text{def}}{=} \lceil \log_2(4/\epsilon) \rceil$ blocks is at most an $\epsilon/2$ fraction of the total domain (since the remaining blocks cover at most an $\epsilon/2$ fraction of the total). Fixing any $i \leq \ell$, we consider the expected number of subblocks incorrectly inferred in the ith block (out of the total $p(n)$ subblocks). We use the obvious inference rule—a majority vote. Thus, a subblock is correctly inferred if and only if a strict majority of the examples obtained from it are labeled correctly. (Having obtained examples from this subblock is clearly a necessary condition for having a strict majority.) Using a Chernoff bound, the probability that we do not have the correct majority in $O(p(n)/\epsilon\gamma^2)$ trials is at most $\exp(-\Omega(2^{-i+1}/\epsilon))$. As in the proof of Proposition 3.1, it follows that the expected fraction of bits that are incorrectly inferred in the first ℓ blocks is bounded above by $\epsilon/20$, and the lemma follows. □

4.2. Query equalizer (Item 2 of Theorem 1.7). For every n, we consider the following concept class $\mathcal{S}_n$. Each concept in the class consists of $p(n)$ blocks of equal size $Q \overset{\text{def}}{=} 2^{n-2}/p(n)$. Each block corresponds to an interval function. Namely, the bit locations in each block are associated with $[Q] \overset{\text{def}}{=} \{1, \dots, Q\}$, and the bits themselves are determined by a pair of integers in $[Q]$. If the ith block is associated with a pair (u_i, v_i), then the jth bit in this block is 1 if and only if $u_i \le j \le v_i$. Note that pairs (u_i, v_i) with $u_i > v_i$ determine an all-zero block.

LEMMA 4.3. $\mathsf{itqc}(\mathcal{S}, n, \epsilon) = \Omega(p(n)/\epsilon)$.

Proof. We start by bounding the expected relative error of the algorithm on a single block when making at most q queries to this block. Later we discuss the implication of such a bound on the total error on S_n. □

The single block case. Suppose first that the learner is deterministic. Then, no matter how it chooses its q queries, there exists an interval of relative length $1/q$ that is never queried. Hence the algorithm cannot distinguish the case in which the target concept is all 0's and the case in which the target concept has 1's only in the nonqueried interval, and it must have error at least $1/2q$ on at least one of these concepts.

Next, we consider a probabilistic learner that makes q queries, all of them answered by 0 (as would be the case for the all-zero concept). Then, for every $\delta > 0$, there must exist an interval of relative length δ/q (i.e., actual length $\delta Q/q$) so that the probability that a query was made in this interval is below δ. We again consider the all-zero concept and the concept that has 1's only in this interval. We consider a $1 - \delta$ fraction of the runs of the algorithm in which no query is made to the above interval. In these runs the algorithm cannot distinguish the all-zero concept from the other concept. Thus, with probability $\frac{1-\delta}{2}$ the algorithm has error at least $\delta/2q$ (on some concept). If we set $\delta = 0.2$ and $q = \frac{1}{100\epsilon}$, then we have that with probability at least 0.4 the error is at least 10ϵ on one of the two concepts.

Back to the concept class S_n. To analyze the execution of a learning (with queries) algorithm on a (complete) concept in S_n, we consider the following game, consisting of two stages. In the *first stage*, the algorithm makes q queries in each block. The algorithm is not charged for any of these queries. In the *second stage*, the algorithm makes a choice, for each block, whether to output a hypothesis for this block or to ask for additional queries. In the latter case it is supplied with an infinite number of queries (for this block) and gets charged only for the q original queries made in the first stage. At the end of the second stage the algorithm must output a hypothesis for each of the remaining blocks. The algorithm is required to output hypotheses which together form an ϵ-approximation of the target. Clearly, the charges incurred in the above game provide a lower bound on the actual query complexity of any learning algorithm. The following claim refers to our charging convention.

Claim: *Any algorithm that learns S_n with ϵ error and confidence* 0.9 *incurs a charge of at least* $0.04 \cdot p(n) \cdot q$.

Proof. Consider the following mental experiment in which an algorithm executes only the first stage, and all its queries are answered "0" (as if each block corresponds to the empty interval function). For each i, let I_i be an interval (of maximum length) in the ith block such that the probability that a query is made (in the above mental experiment) to this interval is below δ (for $\delta = 0.2$). Employing the analysis of the single-block case, it follows that $|I_i| \ge \frac{\delta}{q} \cdot Q$ for every i.

We next define a distribution on $2^{p(n)}$ possible target concepts: For the ith block, independently, with probability 1/2, the interval I_i is chosen (to determine the ith block), and with probability 1/2, the empty interval is chosen. Now consider a full (two-stage) execution of an algorithm that learns S_n with ϵ error and confidence 0.9, when the target is chosen according to the above distribution. Since the bound on the error and confidence of the algorithm are with respect to a worst-case choice of a target concept, it must still hold that, with probability at least 0.9 over the randomization of the algorithm *and* the random choice of the target, the error of the algorithm is at most ϵ.

Suppose, in contradiction, that this algorithm incurs charge less than $0.04p(n) \cdot q$ (where $q = (100\epsilon)^{-1}$ is as above). Then, for at least a 0.96 fraction of the blocks, the algorithm outputs a hypothesis at the end of the first stage. By our assumption on the algorithm, with probability at least 0.9, the overall error in these hypotheses must be bounded by ϵ, and so at most one-ninth of these blocks may have *relative* error greater than 10ϵ (as otherwise the overall error is at least $0.96 \cdot \frac{1}{9} \cdot 10\epsilon > \epsilon$). But this implies that, on a uniformly selected block, with probability at least $0.9 \cdot 0.96 \cdot \frac{8}{9} > 0.6$, the algorithm has relative error smaller than 10ϵ while making at most q queries to the block. However, the last assertion contradicts our analysis of the single block case. Specifically, the algorithm queries the relevant interval $I = I_i$ of the ith block with probability at most $\delta = 0.2$, and so for a random concept assigned to this block (uniformly chosen to be either I or the empty interval), with probability at least 0.8/2, the algorithm error will be at least $\frac{|I|/2}{Q} \geq \frac{0.1}{q} = 10\epsilon$. □

Combining the above discussion with the claim, the lemma follows.

LEMMA 4.4. $\mathsf{csc}(\mathcal{S}, n, \epsilon) = O(p(n)/\epsilon)$.

Proof. The computationally bounded learner simply finds a minimal consistent hypothesis for each block in the concept. Namely, for the ith block it lets $\hat{u}_i \in [Q]$ be the smallest index of an example labeled 1 that belongs to ith block, and it lets $\hat{v}_i \in [Q]$ be the largest index of an example labeled 1 in the ith block. If no example in the block is labeled 1, then it lets $\hat{u}_i = [Q]$, and $\hat{v}_i = 1$ (so the hypothesis is all 0).

Assume the learner is given a sample of size $bp(n)/\epsilon$ $(= b \cdot k(n, \epsilon))$ for some constant $b > 1$. Then, for any particular block, the expected number of examples that fall in the block is b/ϵ, and the probability that fewer than $b/(2\epsilon)$ belong to the block is $\exp(-\Omega(b/\epsilon))$. Thus, by Markov's inequality, for sufficiently large b, the probability that the fraction of blocks receiving fewer than $b/2\epsilon$ examples exceeds $\epsilon/2$ is a small constant. It remains to show that with high probability the total error in the blocks receiving a sufficient number of examples is at most $\epsilon/2$. To this end we show that for each such block, the expected error, relative to the size of the block, is at most ϵ/b' for some constant b'.

Consider a particular block that receives at least $s = b/(2\epsilon)$ examples. Let the interval defining the block be $[u, v]$, and let the hypothesis of the learner be $[\hat{u}, \hat{v}]$. First note that by definition of the algorithm, $[\hat{u}, \hat{v}]$ is always a subinterval of $[u, v]$, and hence we have only one-sided error. In particular, in the case that $u > v$ (i.e., the target interval is empty), the error of the hypothesis is 0. Thus assume $u \leq v$. For the sake of the analysis, if $\hat{u} > \hat{v}$ (i.e., the learner did not observe any positive example in the block, and the hypothesis is all 0), redefine $\hat{u}$ to be $v + 1$ and $\hat{v}$ to be v. By definition, the hypothesis remains all 0. Let $\zeta_{\mathrm{L}} \overset{\mathrm{def}}{=} (\hat{u} - u)/Q$, and $\zeta_{\mathrm{R}} \overset{\mathrm{def}}{=} (v - \hat{v})/Q$. The error of the hypothesis (relative to the block) is the sum of these two random variables. We next bound the expected value of ζ_{L} (the expected value of ζ_{R} is bounded analogously).

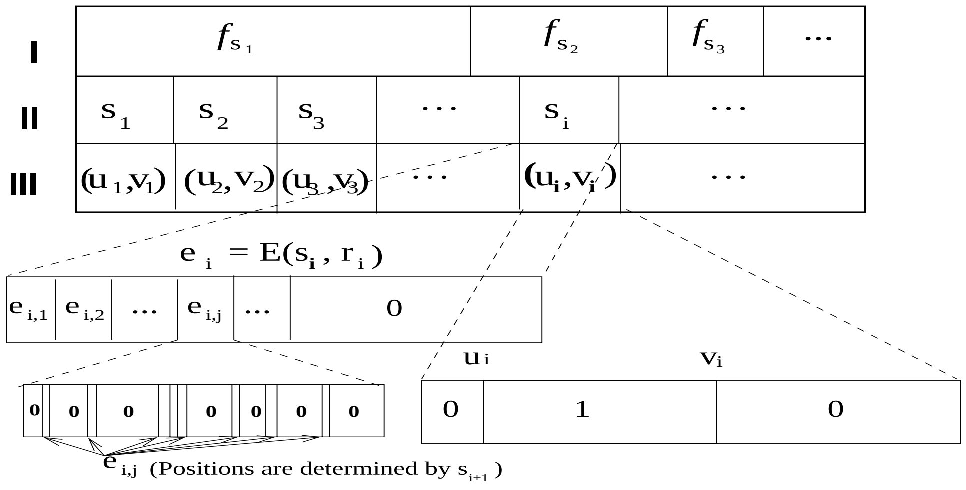


FIG. 5.1. *The general construction with the query equalizer* (*analyzed in Item* 2 *of Theorem* 1.7).

For any integer a, the probability that $\zeta_L > a \cdot \frac{1}{s}$ is the probability that no example fell between u and $u + (a/s) \cdot Q$, which is $(1 - a/s)^s < \exp(-a)$. Therefore,

$$\text{Exp}(\zeta_L) < \sum_{a=0}^{s-1} \Pr\left(\frac{a}{s} < \zeta_L \leq \frac{a+1}{s}\right) \cdot \frac{a+1}{s} \tag{4.1}$$

$$< \sum_{a=0}^{s-1} \Pr\left(\zeta_L > \frac{a}{s}\right) \cdot \frac{a+1}{s} \tag{4.2}$$

$$< \frac{1}{s} \sum_{a=0}^{\infty} (a+1)e^{-a} \tag{4.3}$$

$$< 3/s \;=\; 6\epsilon/d. \tag{4.4}$$

Thus, the (total) expected error of the algorithm on all blocks that receive at least $s = b/(2\epsilon)$ examples is bounded by $12\epsilon/b$. □

5. The general construction.

We adopt the structure of the construction presented in section 3. Specifically, the pseudorandom slice remains the same here, and the sample equalizer is one of the two equalizers analyzed in section 4 (depending on the application). The main modification is in the seed encoder slice (Slice II). For an illustration of the construction, see Figure 5.1.

As in section 3, we encode each seed using the probabilistic coding scheme of Theorem 2.2. In section 3 we repeated each resulting bit (of each encoded seed) in each bit of a corresponding information field, where the information fields occupied only a small fraction of Slice II and their locations were fixed. Here we augment this strategy by having only a few of the bits of these information fields equal the corresponding bit and the rest be set to zero. Thus, only a few locations in each information field are really informative. Furthermore, the locations of the informative bits will not be known a priori but will rather be "random" (as far as the computational bounded learner is concerned). This strategy effects the computational bounded learner both in the query and noisy models. Using queries the computational bounded learner may focus on the informative fields but it cannot hit informative bits within these fields any better than at random. In the presence of noise, the learner has an even harder time determining whether an informative bit is obtained or not.

Suppose that a ρ fraction of the bits in an information field are really informative. Then to distinguish a 0 from a 1 (with constant confidence) the learner must observe $\Theta(1/\rho)$ examples from the information field. Things become even harder in the presence of classification noise at rate $\eta = 0.5 - \gamma$, say, greater than 0.2. In this case, when getting an example in the information field, or even making a query to the information field, the answer may be 0 both in case the bit is not informative (and the answer is correct) and in case the bit is informative. The latter case happens with probability $0.5 + \gamma$ (resp., $0.5 - \gamma$) when the real information is 0 (resp., 1). Thus, in case the information bit is 0 (resp., 1), the answer is 0 with probability $1 - \eta$ (resp., with probability $(1 - \rho) \cdot (0.5 + \gamma) + \rho \cdot (0.5 - \gamma) = 1 - \eta - 2\rho\gamma$). As $1 - \eta$ is bounded away from (both 0 and 1), distinguishing an encoding of 0 from an encoding of 1 (with constant confidence) requires $\Theta(1/(\rho\gamma)^2)$ queries/examples from the information field.

The above discussion avoids the question of how we can make the informative locations be "random" (as far as the computational bounded learner is concerned). These "random" locations must be part of the specification of the concepts in the class. We cannot have truly random locations if we want to maintain a polynomial-size description of individual concepts. The use of pseudorandom functions is indeed a natural solution. There is still a problem to be resolved—the computational bounded learner must be able to obtain the locations of "information" for those blocks that it needs to learn. Our solution is to use the $(i + 1)$st pseudorandom function (from Slice I) in order to specify the locations in which information regarding the ith seed is given (in Slice II).

The following description is in terms of two density functions, denoted $\rho_1, \rho_2 : \mathcal{N} \times \mathcal{R} \mapsto \mathcal{R}$. Various instantiations of these functions will yield all the results in the paper. Each function in the concept class consists of the three slices, and each slice is further subpartitioned into *blocks* as described below.

Slice I*: The pseudorandom part.* This is exactly as in section 3. That is, the ith block has size 2^{n-1-i} and is determined by a pseudorandom function $f_{s_i} : \{0,1\}^{n-1-i} \rightarrow \{0,1\}$, where the seed s_i is of length n.

Slice II*: The seeds encoder.* As in section 3, this slice is partitioned into $t = n - \log_2 n - O(1)$ blocks (of equal size) so that the ith block has size $B \stackrel{\text{def}}{=} \frac{1}{t} \cdot 2^{n-2}$ and encodes the ith seed, s_i, using an additional string r_i of length $O(n)$. Let e_i be the code word obtained by employing the probabilistic encoding scheme on input s_i using randomness r_i. Recall that $m \stackrel{\text{def}}{=} |e_i| = O(n)$. The ith block is further divided into m (consecutive) *information fields*, each of size $\rho_1(n, 2^{-i}) \cdot \frac{B}{m}$, and an additional *empty field* (of size $(1 - \rho_1(n, 2^{-i}) \cdot B)$.

A $\rho_2(n, 2^{-i})$ fraction of the bits in the jth information field are set to the value of the jth bit of e_i. These bits are called *informative* and their locations are determined ("randomly") by a pseudorandom function, $h_{s_{i+1}} : \{0,1\}^n \mapsto \{0,1\}^n$ (this function uses the $(i + 1)$st seed!). The remaining bits in each information field as well as all bits of the empty field are set to zero.

A few details are to be specified. First, bit locations in Slice II are associated with strings of length $n - 2$. Thus, bit location $\alpha \in \{0,1\}^{n-2}$ that is inside an information field is informative if and only if $h_{s_{i+1}}(11\alpha)$ is among the first $2^n \cdot \rho_2(n, 2^{-i})$ strings in the lexicographic order of all n-bit long strings. The pseudorandom functions (over the domain $\{0,1\}^{n-i-1}$) used in Slice I are determined by the same seeds by letting $f_{s_i}(z) = \mathsf{lsb}(h_{s_i}(0^{i+1}z))$, where $\mathsf{lsb}(\sigma_n \cdots \sigma_1) = \sigma_1$ is the least significant bit of $\sigma_n \cdots \sigma_1$.[14] Thus, there is no "computational observable" interference between the randomness used for determining informative locations and the randomness used in Slice I.

Note that Slice II in the construction of section 3 is obtained by setting $\rho_1(n,\epsilon) = \frac{tm}{g(n,\epsilon)k(n,\epsilon)}$ and $\rho_2 \equiv 1$.

Slice III*: The equalizer.* In this slice we use one of the two equalizers analyzed in Theorem 1.7 (depending on the application).

5.1. Analysis of Slices I and II. It will be convenient to analyze the concept class that results from the above by omitting Slice III (the equalizer). We refer to the resulting class as to the *core* class.

FACT 5.1. *The core class has information theoretic sample complexity* $\mathsf{itsc}(n,\epsilon,\gamma) = O(n^2/\epsilon\gamma^2)$, *which in turn is* $O(k(n,\epsilon)/\gamma^2)$.

Proof. The proof follows from the "noisy Occam razor" [Lai88]. $\square$

We are not interested in an information theoretic lower bound for the core class since the equalizer will dominate the information theoretic complexities. Thus, we turn to analyze the computational complexities of the core class.

LEMMA 5.2. *The core class has*

1. *computational* (noiseless) sample *complexity* $\mathsf{csc}(n,\epsilon) = \Theta(\frac{n^2}{\rho_1(n,\epsilon)\cdot\rho_2(n,\epsilon)})$, *provided that* $\frac{1}{(\rho_1\cdot\rho_2)}$ *is an admissible function and that it is lower bounded by* $1/\epsilon$.
2. *for* $\gamma \le \frac{1}{4}$, *computational* (noisy) sample *complexity* $\mathsf{csc}(n,\epsilon,\gamma) = \Theta(\frac{n^2}{\rho_1(n,\epsilon)\cdot\rho_2(n,\epsilon)^2\cdot\gamma^2})$, *provided that* $\frac{1}{\rho_1\cdot\rho_2^2}$ *is an admissible function and that it is lower bounded by* $1/\epsilon$.
3. *computational* query *complexity* $\mathsf{cqc}(n,\epsilon) = \Theta(\frac{n}{\rho_2(n,\epsilon)})$, *provided that* $\frac{1}{\rho_2}$ *is an admissible function and that it is lower bounded by* $1/\epsilon$.

Proof. Item 1 (noiseless sample complexity). This item follows by observing that arguments used in Lemma 3.2 can be modified to obtain the desired bound. Consider the ith block of Slice II. We first note that a random example hits an information field of the ith block with probability $\rho_1(n,2^{-i})/t$ (i.e., with probability $1/t$ it falls in the ith block and conditioned on being in the ith block it falls in an information field with probability $\rho_1(n,2^{-i})$). Thus, the probability of hitting a specific information field (out of the $m = \Theta(n)$ fields) is $\frac{\rho_1(n,2^{-i})}{t\,m} = \Theta(\frac{\rho_1(n,2^{-i})}{n^2})$. We also know that a random example in an information field is informative (i.e., depends on the encoded bit) with probability $\rho_2(n,2^{-i})$ and is set to zero otherwise.

For the lower bound, consider the ith block for $i \overset{\text{def}}{=} \lfloor \log(1/4\epsilon) \rfloor$. Similar to what was shown in the lower bound of Lemma 3.2, for an appropriate constant c'', a sample of size $c'' \cdot \frac{n^2}{(\rho_1(n,\epsilon)\cdot\rho_2(n,\epsilon)}$ will contain informative examples in less than $c_{\text{sec}}\, m$ of the information fields in the ith block (where c_{sec} is the constant in Item 3 of Theorem 2.2). As a result, the ith seed cannot be obtained and the error of the learner on Slice I is too large.

The proof of the upper bound is easily adapted as well. As in the proof of Lemma 3.2 we have that for a sufficiently large constant c', with very high probability, a sample of size $c' \cdot n^2/(\rho_1(n,\epsilon)\cdot\rho_2(n,\epsilon))$ will contain at least one informative example in all but a small constant fraction of the m information fields in the ith block, for

[14]Our description assumes "pseudorandom functions" mapping n-bit strings to n-bit strings, whereas the definition given in section 2.1 refers to functions mapping n-bit strings to a single bit. However, given pseudorandom functions of the latter type, one can easily construct functions as used here; see [Gol95].

every $i \leq \ell \overset{\text{def}}{=} \lceil \log(8/\epsilon) \rceil$. The only difference here is that while seeing a 1 in an information field in fact means that the bit encoded in it is 1, seeing only 0's in the field only provides statistical evidence towards 0. Thus, while in Lemma 3.2 we only had to deal with missing informative examples (that encode seeds), here we might have errors when inferring that the bit encoded is 0. However, the coding scheme (of Theorem 2.2) allows a constant fraction (i.e., c_{err}) of errors, and hence we can handle a constant fraction of errors in each of the first ℓ blocks. Note that the ℓ corresponding seeds determine not only the first ℓ blocks of Slice I but also the locations of informative bits in the first $\ell - 1$ blocks of Slice II.

Item 2 (noisy sample complexity). The additional difficulty we encounter here (as compared to Item 1 above) is that due to the noise it does not suffice to "hit" informative examples inside information fields in order to infer the encoded bit. Namely, each example (informative or not) has an incorrect label with probability $\eta = \frac{1}{2} - \gamma$. Therefore, seeing a 1 in an information field does not necessarily mean that it is an informative example and the field encodes the bit 1, but rather it could be a noisy bit in an information field encoding the bit 0.

However, there is clearly still a difference between information fields that encode 1, and those that encode 0: In case an information field encodes 0, a random example in it will be labeled 1 with probability η. On the other hand, in case an information field encodes 1, a random example in it will be labeled 1 with probability $\rho_2(n, 2^{-i}) \cdot (1 - \eta) + (1 - \rho_2(n, 2^{-i})) \cdot \eta = \eta + 2\gamma\rho_2(n, 2^{-i})$. Thus, we need to distinguish a 0–1 sample with expectation η from a 0–1 sample with expectation $\eta + \Theta(\gamma\, \rho_2(n, 2^{-i}))$. This is feasible (with high probability) using $O(1/(\gamma^2\, \rho_2(n, 2^{-i}))$ examples. Since our coding scheme can suffer a constant fraction of errors, we can allow that a small fraction of information fields will receive fewer than the required number of examples, and that among those receiving the desired number, a small fraction will be determined incorrectly. The upper bound follows.

For η that is bounded below by some constant (say, $\eta \geq 1/4$), a sample of size $\Omega(1/(\rho_2(n, 2^{-i})\gamma)^2)$ is also required to distinguish between the two cases discussed above with probability greater than, say $1/2 + c_{\text{sec}}/8$, where c_{sec} is the constant defined in Item 3 of Theorem 2.2. Suppose that a sample of size $c'm/(\rho_2(n, 2^{-i})\gamma)^2$ falls in the ith block of Slice II. Then, for sufficiently small c', at most $c_{\text{sec}}/2$ information fields receive a sufficient number of examples. Hence, even if all these information fields are correctly inferred, by Remark 2.1, with very high probability no information about the ith seed will be revealed. In particular, this holds for $i = \lfloor \log_2(1/4\epsilon) \rfloor$. We conclude that, for $\gamma \leq 1/4$,

$$\mathsf{csc}(n, \epsilon, \gamma) = \Theta\left(\frac{n^2}{\rho_1(n, \epsilon)} \cdot \frac{1}{(\rho_2(n, \epsilon) \cdot \gamma)^2} \right). \tag{5.1}$$

Item 3. Using arguments similar to those applied above we show that $O(n/\rho_2(n, \epsilon))$ queries suffice for efficiently determining the first $\ell \overset{\text{def}}{=} \lceil \log_2(8/\epsilon) \rceil$ seeds residing in the first ℓ blocks of Slice II. Specifically, we use a $2^{-\ell-4+i}$ fraction of the $c' \cdot (n/\rho_2(n, \epsilon))$ queries as a random sample into the information fields of the ith block, for $i = 1, \ldots, \ell$ (and ignore the empty fields in all blocks). Thus, we have $c' \cdot 2^{-\ell-4+i} \cdot (n/\rho_2(n, \epsilon)) = 2c'' \cdot m/\rho_2(n, 2^{-i})$ random examples in the ith block, where c'' is a constant related to the constant c'. With overwhelmingly high probability we'll obtain at least $c''m$ informative bits. For a suitable choice of the constants (c' and c''), this suffices to recover the ith seed for every $i \leq \ell$. Observe that the only part in which we have used queries is in the skewing of the random examples among the various blocks.

We now turn to the lower bound and let $i \stackrel{\text{def}}{=} \lfloor \log_2(1/4\epsilon) \rfloor$. Considering the ith block of Slice I, we show that, as long as the informative locations in the ith block of Slice II are unknown, too few queries do not allow us to obtain information regarding the ith seed. This will lead to the failure of the computational bounded learner, since without such information the ith block of Slice I looks totally random (to this learner). The actual argument starts from the last block (i.e., tth block) and proceeds up to the ith block. Assuming that the learner has no knowledge of the jth seed, for $j > i$, we show that he obtains no knowledge of the $j-1$st seed. On top of what is done in the analogous part of the proof of Lemma 3.2, we need to argue that having no knowledge of the jth seed puts the learner in the same situation as if it has selected its queries at random: We can think of it making a query and then having a random biased coin determine if this query (into the seed encoder) carries information. $\square$

5.2. Applications of Lemma 5.2. Combining the analysis of the core class (i.e., Lemma 5.2) with the adequate equalizer of Theorem 1.7, we derive all main results of this paper. Before we do so, recall that Theorem 1.2 is a special case of Theorem 1.3 (i.e., with noise equals zero; $\gamma = 1/2$), whereas the latter is a special case of the first item of Theorem 1.4 (i.e., with $g_2 \equiv 1$). Thus, we focus on proving Theorems 1.4 and 1.5.

Proof of Theorem 1.4. In order to prove the first item, we set $\rho_1(n,\epsilon) \stackrel{\text{def}}{=} \frac{n^2}{g_1(n,\epsilon)\cdot k(n,\epsilon)}$ and $\rho_2(n,\epsilon) \stackrel{\text{def}}{=} \frac{1}{g_2(n,\epsilon)}$. By Item 1 of Theorem 1.7 and Fact 5.1, we have $\mathsf{itsc}(n,\epsilon,\gamma) = \Theta(k(n,\epsilon)/\gamma^2)$. Invoking Item 1 of Lemma 5.2, we have $\mathsf{csc}(n,\epsilon) = \Theta(g_1(n,\epsilon)g_2(n,\epsilon)\cdot k(n,\epsilon))$. Invoking Item 2 of Lemma 5.2, we have $\mathsf{csc}(n,\epsilon,\gamma) = \Theta(g_1(n,\epsilon)g_2(n,\epsilon)^2 \cdot k(n,\epsilon)/\gamma^2)$, for every $\gamma \le 1/4$. Thus, the first item follows.

In order to prove the second item, we set $\rho_1(n,\epsilon) \stackrel{\text{def}}{=} 1$ and $\rho_2(n,\epsilon) \stackrel{\text{def}}{=} \frac{n^2}{k(n,\epsilon)}$. Again, we have $\mathsf{itsc}(n,\epsilon,\gamma) = \Theta(k(n,\epsilon)/\gamma^2)$, and invoking Items 1 and 2 of Lemma 5.2, we have $\mathsf{csc}(n,\epsilon) = \Theta(k(n,\epsilon))$ and $\mathsf{csc}(n,\epsilon,\gamma) = \Theta(k(n,\epsilon)^2/n^2\gamma^2))$, for every $\gamma \le 1/4$. Thus, the second item of the theorem follows. (Actually, we can obtain a more general result by setting $\rho_1(n,\epsilon) \stackrel{\text{def}}{=} n^{-(d-a-2)}\epsilon^{1-\beta}$ and $\rho_2(n,\epsilon) \stackrel{\text{def}}{=} n^{-d}\epsilon^{\beta}$, for any $a \in [0, d-2]$ and $\beta \in [0,1]$.)

Proof of Theorem 1.5*, for $g_2 \ge n$.* Here we set $\rho_1(n,\epsilon) \stackrel{\text{def}}{=} \frac{n}{g_2(n,\epsilon)}$ and $\rho_2(n,\epsilon) \stackrel{\text{def}}{=} \frac{n}{g_1(n,\epsilon)\cdot k(n,\epsilon)}$, and the hypothesis $g_2(n,\epsilon) \ge n$ guarantees that $\rho_1(n,\epsilon) \le 1$ as required by the admissibility condition. By Item 2 of Theorem 1.7 and Fact 5.1, we have $\mathsf{itqc}(n,\epsilon) = \Theta(k(n,\epsilon)) = \Theta(\mathsf{itsc}(n,\epsilon))$. Invoking Item 1 of Lemma 5.2, we have $\mathsf{csc}(n,\epsilon) = \Theta(g_1(n,\epsilon)g_2(n,\epsilon)\cdot k(n,\epsilon))$. Invoking Item 3 of Lemma 5.2, we have $\mathsf{cqc}(n,\epsilon) = \Theta(g_1(n,\epsilon)\cdot k(n,\epsilon))$. The theorem follows.

5.3. Proof of Theorem 1.5 (for arbitrary g_2). The core class (analyzed in Lemma 5.2) provides a computationally bounded learner that uses queries, which is an advantage over a computationally bounded learner that uses only uniformly distributed examples. Whereas the former may focus its queries on the first $\log_2(2/\epsilon)$ blocks of Slice II, the latter may not. Thus, typically, using queries entitles an advantage of a factor of $n/\log(1/\epsilon)$ in trying to learn Slice II above. To close this gap (and allow us to establish Theorem 1.5 for arbitrary g_2), we modify Slice II as follows. The basic idea is to randomly permute the locations of the information fields of the various blocks. Thus the query-learner is forced to look for the bits it needs in all possible locations (rather than "zoom-in" on the appropriate block).

Slice II (*modified*). Let $t \stackrel{\text{def}}{=} n/\log_2 n$ (rather than $t = n-\log_2 n$). This slice is partitioned into $t \cdot m$ fields of equal size, $F \stackrel{\text{def}}{=} \frac{1}{tm} \cdot 2^{n-2}$, where m is (as before) the length of the encoding of an n-bit long seed. Unlike the above construction, we do not have a common empty field (instead each field contains an informative part and an empty part as described below). We use m permutations over $\{1,\ldots,t\}$, denoted $\pi_1,\ldots,\pi_m$, to determine the correspondence between fields and seed-information. Specifically, the jth bit of the ith seed is "encoded" in field number $(j-1)\cdot t + \pi_j(i)$. (The permutations are part of the description of the concept.) Each field corresponding to one of the bits of the ith seed consists of two parts. The first part, containing the first $\rho_1(n, 2^{-i}) \cdot F$ bits of the field, carries information about the corresponding bit of the seed; whereas the second part (the rest of the field's bits) is uncorrelated to the seed. Loosely speaking, the *informative part* contains the results of independent coin flips each with bias $\rho_2(n, 2^{-i})$ towards the correct value of the corresponding bit (i.e., the probability that the answer is correct is 0.5 + bias); whereas the rest contains the results of independent unbiased coin flips. Actually, the random choices are implemented by a pseudorandom function determined by the $i+1$st seed.

LEMMA 5.3. *Assume that $\frac{1}{\rho_1}$ and $\frac{\epsilon}{\rho_2^2}$ are admissible functions. Then the modified core class has*

1. *computational* (noise-less) sample *complexity* $\mathsf{csc}(n,\epsilon) = \Theta\left(\frac{n^2}{\rho_1(n,\epsilon)\cdot\rho_2(n,\epsilon)^2}\right)$.
2. *computational* query *complexity* $\mathsf{cqc}(n,\epsilon) = \Theta\left(\frac{n^2}{\rho_2(n,\epsilon)^2}\right)$.

Proof. We follow the structure of the proof of Lemma 5.2, indicating the necessary modifications.

Item 1. Considering Slice II, we note that a random example hits an information part of a field belonging to the ith seed with probability $\rho_1(n, 2^{-i})/t$. Intuitively, $\Theta(\rho_2(n, 2^{-i})^{-2})$ such hits are required for obtaining reliable information from this field.

For the lower bound, we assume that the learner is given the permutations π_j for free. Still, the arguments used in Lemma 5.2 imply that it needs $\Omega(\rho_2(n, 2^{-i})^{-2} \cdot m)$ hits in the fields of the ith seed in order to recover this seed. Using $t, m = \Omega(n)$, the lower bound follows.

The proof of the upper bound is to be adapted as here we cannot assume that the permutations π_j are known to the learner. For $i = 1,\ldots,\log_2(8/\epsilon)$, the learner determines the ith seed as follows. For $j = 1,\ldots,m$, the learner determines the value of the jth bit in the encoding of the ith seed. It considers only examples in the $\rho_1(n, 2^{-i}) \cdot F$ prefix of each of the relevant fields; that is, fields with indices $(j-1)\cdot t+1,\ldots,(j-1)\cdot t+t$. For each such field it estimates the bias of the field. With high constant probability, the estimated bias of field $(j-1)\cdot m+\pi_j(i)$ is approximately $\rho_2(n, 2^{-i})$ and, under our assumption on ρ_2, every other field corresponding to the jth bit of an encoding of some other seed, has significantly different bias in its $\rho_1(n, 2^{-i}) \cdot F$ prefix. Thus, this bit is obtained correctly with high constant probability. As usual, this allows us to decode correctly the entire seed. Having the ith seed we may determine the ith pseudorandom function as well as the "encoding" of the bits of the $i-1$st code word (i.e., one may efficiently determine the value of each bit in each of the fields corresponding to the $i-1$st seed). The upper bound follows.

Item 2. The upper bound follows easily by the obvious adaptation of the above learning strategy (i.e., when trying to determine the jth bit in the encoding of the ith seed the learner makes queries only to the $\rho_1(n, 2^{-i}) \cdot F$ prefix of each of the relevant fields).

For the lower bound we need to modify the argument given above (as here we cannot afford giving away the permutations π_j for free). In fact, the whole point of the modification was to deprive the learner of such information. Still, when arguing about the ith seed, for $i = \log_2(4/\epsilon)$, we may give the learner $\pi_j(1), \ldots, \pi_j(i-1)$ $(\forall j)$ for free. We may assume without loss of generality that the learner does not make queries to these fields (i.e., to a field with index $(j-1) \cdot t + \pi_j(\ell)$ for $\ell < i$ and any j). Based on our encoding scheme, the learner must infer $\Omega(m)$ of the bits in the encoding of the ith seed. For each bit it must discover $\pi_j(i)$ and make $\Omega\left(\frac{1}{\rho_2(n,2^{-i})}\right)$ queries. However, for every j and $\ell \geq i$, the field corresponding to the jth bit of the encoding of the ℓth seed has bias (in its $\rho_1(n, 2^{-i}) \cdot F$ prefix) that is bounded above by $\rho_2(n, 2^{-i})$. Hence, for each j, the learner must perform $\Omega(n/(\rho_2(n, 2^{-i})^2)$ queries to these fields in order to infer the desired bit. The lower bound follows. ◻

Deriving Theorem 1.5. To prove Theorem 1.5 (in its general form), we set $\rho_1(n,\epsilon) \stackrel{\text{def}}{=} \frac{1}{g_2(n,\epsilon)}$ and $\rho_2(n,\epsilon) \stackrel{\text{def}}{=} \sqrt{\frac{n^2}{g_1(n,\epsilon)\cdot k(n,\epsilon)}}$. Invoking Item 1 of Lemma 5.3, we have $\mathsf{csc}(n,\epsilon) = \Theta(g_1(n,\epsilon)g_2(n,\epsilon) \cdot k(n,\epsilon))$. Invoking Item 2 of Lemma 5.3, we have $\mathsf{cqc}(n,\epsilon) = \Theta(g_1(n,\epsilon) \cdot k(n,\epsilon))$. So all that is left is to analyze $\mathsf{itsc}(n,\epsilon)$ and $\mathsf{itqc}(n,\epsilon)$, the information theoretic sample and query complexity of the entire class. By Item 2 of Theorem 1.7, both information theoretic complexities of Slice III are $\Theta(k(n,\epsilon))$. By Occam's Razor [BEHW87], the (information theoretic) sample complexity is at most $O(s/\epsilon)$, where 2^s is the number of concepts in the core class. Here we have $s = t \cdot n + m \cdot \log_2(t!)$, where the term $t \cdot n < n^2$ is due to the t seeds and $m \cdot \log_2(t!) = O(mt \log t)$ is due to the $m = O(n)$ permutations. We use[15] $t = O(n/\log n)$ to obtain $mt \cdot \log t = O(n^2)$, which in turn yields $s/\epsilon = O(k(n,\epsilon))$. Hence, $\mathsf{itqc}(n,\epsilon) = \Theta(k(n,\epsilon)) = \Theta(\mathsf{itsc}(n,\epsilon))$, and the theorem follows.

6. Proof of Theorem 1.6.

Here we merely use a pseudorandom generator [BM84, Yao82]. Specifically, we need a generator, G, which stretches seeds of length n into sequences of length $p(n)$. The concept class, $\mathcal{C} = \{C_n\}$, will correspond to all possible choices of a seed for the generator. Specifically, for every seed $s \in \{0,1\}^n$, we get a concept $f_s \in C_n$ defined so that $f_s(x) \stackrel{\text{def}}{=} \sigma_i$, where σ_i is the ith bit of $G(s)$ and x belongs to the ith subset in a "nice" $p(n)$-way partition of $\{0,1\}^n$ (e.g., a partition by lexicographic order in which all parts are of about the same size).

By Occam's Razor [BEHW87], the above concept class has $\mathcal{ITSC}(n,\epsilon) = O(n/\epsilon)$. On the other hand, considering a variation of the standard lower-bound distribution (i.e., which assigns probability $1-4\epsilon$ uniformly to all instances in a single subset and is uniform on all other instances), we derive the computational lower bound. Specifically, using $0.1 \cdot p(n)/\epsilon$ samples, with overwhelmingly high probability, the learner only sees $p(n)/2$ different bits of the pseudorandom sequence. As far as a computationally bounded learner is concerned, the rest of the sequence is random and so it will fail with probability at least $\frac{1}{2}$ when trying to predict any example corresponding to an unseen bit. Thus, $\mathcal{CSC}(n,\epsilon) > 0.1 \cdot p(n)/\epsilon$.

It is not hard to see that $10 \cdot p(n)/\epsilon$ samples allow efficient learning up to error ϵ (with respect to any distribution) and so $\mathcal{CSC}(n,\epsilon) \leq 10 \cdot p(n)/\epsilon$.

[15]Alternatively, we could use $t = n - O(\log n)$ (as in previous constructions) and derive the theorem for $k(n,\epsilon) = \Omega(n^2 \log n)/\epsilon$.

Acknowledgments. We are grateful to Moni Naor and Ronny Roth for helpful discussions.

REFERENCES

[AD96] J. Aslam and S. Decatur, *On the sample complexity of noise-tolerant learning*, Inform. Process. Lett., 57 (1996), pp. 189–195.

[AK91] D. Angluin and M. Kharitonov, *When won't membership queries help?*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, New Orleans, LA, 1991, pp. 444–453.

[Ang87] D. Angluin, *Learning regular sets from queries and counterexamples*, Inform. and Comput., 75 (1987), pp. 87–106.

[BBCM95] C. H. Bennett, G. Brassard, C. Crépeau, and U. Maurer, *Generalized privacy amplification*, IEEE Trans. Inform. Theory, 41 (1995), pp. 1915–1923.

[BEHW87] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, *Occam's razor*, Inform. Process. Lett., 24 (1987), pp. 377–380.

[BEHW89] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, *Learnability and the Vapnik-Chervonenkis dimension*, J. ACM, 36 (1989), pp. 929–865.

[BM84] M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudorandom bits*, SIAM J. Comput., 13 (1984), pp. 850–864.

[BW86] E. Berlekamp and L. Welch, *Error Correction of Algebraic Block Codes*, U.S. Patent 4,633,470, 1986.

[CK78] I. Csiszár and J. Körner, *Broadcast channels with confidential messages*, IEEE Trans. Inform. Theory, 24 (1978), pp. 339–348.

[CM97] C. Cachin and U. M. Maurer, *Linking information reconciliation and privacy amplification*, J. Cryptology, 10 (1997), pp. 97–110.

[CT91] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley, New York, 1991.

[EHKV89] A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant, *A general lower bound on the number of examples needed for learning*, Inform. and Comput., 82 (1989), pp. 247–251.

[Gol95] O. Goldreich, *Foundation of Cryptography—Fragments of a Book*, February 1995; revised version, January 1998; http://theory.lcs.mit.edu/~oded/frag.html.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali, *How to construct random functions*, J. ACM, 33 (1986) pp. 792–807.

[HILL] J. Håstad, R. Impagliazzo, L. A. Levin, and M. G. Luby, *A pseudorandom generator from any one-way function*, SIAM J. Comput., 28 (1999), pp. 1364–1396.

[Jus72] J. Justesen, *A class of constructive asymptotically good alegbraic codes*, IEEE Trans. Inform. Theory, 18 (1972), pp. 652–656.

[Kha93] M. Kharitonov, *Cryptographic hardness of distribution-specific learning*, In Proceedings of the 25th Annual ACM Symposium on the Theory of Computing, San Diego, CA, 1993, pp. 372–381.

[KV94] M. J. Kearns and L. G. Valiant, *Cryptographic limitations on learning Boolean formulae and finite automata.* J. ACM, 41 (1994), pp. 67–95.

[Lai88] P. D. Laird, *Learning from Good and Bad Data*, Kluwer Internat. Ser. Engrg. Comput. Sci., Kluwer Academic Publishers, Boston, MA, 1988.

[Mau91] U. M. Maurer, *Perfect cryptographic security from partially independent channels*, in Proceedings of the 23rd Symposium on the Theory of Computing, New Orleans, LA, 1991, pp. 561–571.

[NR95] M. Naor and O. Reingold, *Synthesizers and their application to the parallel construction of pseudo-random functions*, in Proceedings of the 36th Symposium on the Foundations of Computer Science, Milwaukee, WI, 1995, pp. 170–181; full version available online from http://www.eccc.uni-trier.de/eccc/.

[PV88] L. Pitt and L. Valiant, *Computational limitations of learning from examples*, J. ACM, 35 (1988), pp. 965–984.

[Sim93] H. U. Simon, *General bounds on the number of examples needed for learning probabilistic concepts*, in Proceedings of the Sixth Annual ACM Workshop on Computational Learning Theory, Santa Cruz, CA, ACM Press, 1993, pp. 402–411.

[Tal94] M. Talagrand, *Sharper bounds for Gaussian and empirical processes*, Ann. Probab., 22 (1994), pp. 28–76.

[Val84] L. G. Valiant, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[VC71] V. N. Vapnik and A. Ya. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory Probab. Appl., 16 (1971), pp. 264–280.

[Wyn75] A. D. Wyner, *The wire-tap channel*, Bell System Tech. J., 54 (1975), pp. 1355–1387.

[Yao82] A. C. Yao, *Theory and application of trapdoor functions*, in Proceedings of the 23rd Symposium on the Foundations of Computer Science, Chicago, IL, 1982, pp. 80–91.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 3, pp. 880–892

A FASTER AND SIMPLER ALGORITHM FOR SORTING SIGNED PERMUTATIONS BY REVERSALS[*]

HAIM KAPLAN[†], RON SHAMIR[†], AND ROBERT E. TARJAN[‡]

Abstract. We give a quadratic time algorithm for finding the minimum number of reversals needed to sort a signed permutation. Our algorithm is faster than the previous algorithm of Hannenhalli and Pevzner and its faster implementation by Berman and Hannenhalli. The algorithm is conceptually simple and does not require special data structures. Our study also considerably simplifies the combinatorial structures used by the analysis.

Key words. sorting permutations, reversal distance, computational molecular biology

AMS subject classifications. 62P10, 68P10

PII. S0097539798334207

1. Introduction.

In this paper we study the problem of sorting signed permutations by reversals. A *signed permutation* is a permutation $\pi = (\pi_1, \ldots, \pi_n)$ on the integers $\{1, \ldots, n\}$, where each number is also assigned a sign of plus or minus. A *reversal*, $\rho(i,j)$, on π transforms π to

$$\begin{aligned}\pi' &= \pi\rho(i,j)\\ &= (\pi_1, \ldots, \pi_{i-1}, -\pi_j, -\pi_{j-1}, \ldots, -\pi_i, \pi_{j+1}, \ldots, \pi_n).\end{aligned}$$

The minimum number of reversals needed to transform one permutation to another is called the *reversal distance* between them. The problem of *sorting signed permutations by reversals* is to find, for a given signed permutation π, a sequence of reversals of minimum length that transforms π to the identity permutation $(+1, +2, \ldots, +n)$.

The motivation for studying the problem arises from molecular biology: Concurrent with the fast progress of the human genome project, genetic and DNA data on many model organisms are accumulating rapidly, and consequently the ability to compare genomes of different species has grown dramatically. One of the best ways of checking similarity between genomes on a large scale is to compare the order of appearance of identical genes in the two species. In the thirties, Dobzhansky and Sturtevant [7] had already studied the notion of inversions in chromosomes of *drosophila*. In the late eighties, Jeffrey Palmer demonstrated that different species may have essentially the same genes, but the gene order may differ between species. Taking an abstract perspective, the genes along a chromosome can be thought of as points along a line. Numbers identify particular genes and, since genes have directionality, the

[*]Received by the editors February 18, 1998; accepted for publication (in revised form) January 26, 1999; published electronically December 15, 1999. A preliminary version of this paper was presented at the Eighth ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, Philadelphia, PA, 1997.
http://www.siam.org/journals/sicomp/29-3/33420.html

[†]Department of Computer Science, Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel (haimk@math.tau.ac.il, shamir@math.tau.ac.il). The research of the second author was supported in part by a grant from the Israeli Ministry of Science and the Arts and by US Department of Energy grant DE-FG03-94ER61913/A000.

[‡]Department of Computer Science, Princeton University, Princeton, NJ 08544, and InterTrust Technologies Corporation, Sunnyvale, CA 94086 (ret@cs.princeton.edu). The research of this author at Princeton University was partially supported by NSF grants CCR-8920505 and CCR-9626862 and Office of Naval Research contract N00014-91-J-1463.

signs correspond to their direction. Palmer and others have shown that the difference in order may be explained by a small number of reversals [17, 18, 19, 20, 12]. These reversals correspond to evolutionary changes during the history of the two genomes, so the number of reversals reflects the evolutionary distance between the species. Hence, given two such permutations, their reversal distance measures their evolutionary distance.

Mathematical analysis of genome rearrangement problems was initiated by Sankoff [22, 21]. Kececioglu and Sankoff [16] gave the first constant-factor polynomial approximation algorithm for the problem and conjectured that the problem is NP-hard. Bafna and Pevzner [3] and more recently Christie [6] improved the approximation factor, and additional studies have revealed the rich combinatorial structure of rearrangement problems [15, 14, 2, 9, 10]. Quite recently, Caprara [5] has established that sorting *unsigned* permutations is NP-hard, using some of the combinatorial tools developed by Bafna and Pevzner [3].

In 1995, Hannenhalli and Pevzner [11] showed that the problem of sorting a *signed* permutation by reversals is polynomial. They proved a duality theorem that equates the reversal distance with the sum of three combinatorial parameters (see Theorem 2.3 below). Based on this theorem, they proved that sorting signed permutations by reversals can be done in $O(n^4)$ time. More recently, Berman and Hannenhalli [4] described a faster implementation that finds a minimum sequence of reversals in $O(n^2\alpha(n))$ time, where α is the inverse of Ackerman's function [1] (see also [23]).

In this study we give an $O(n^2)$ algorithm for sorting a signed permutation of n elements, thereby improving upon the previous best-known bound [4]. In fact, if the reversal distance is r, our algorithm requires $O(r \cdot n + n\alpha(n))$ time. In addition to giving a better time bound, our work considerably simplifies both the algorithm and the combinatorial structure needed for the analysis, as follows:

- The basic object we work with is an implicit representation of the overlap graph, to be defined later, in contrast with the interleaving graph in [11] and [4]. The overlap graph is combinatorially simpler than the interleaving graph. As a result, it is easier to produce a representation for the overlap graph from the input, and to maintain it while searching for reversals.
- As a consequence of our ability to work with the overlap graph, we need not perform any "padding transformations," nor do we have to work with "simple permutations" as in [11] and [4].
- We deal with the unoriented and oriented parts of the permutation separately, which makes the algorithm much simpler.
- The notion of a *hurdle*, one of the combinatorial entities defined by [11] for the duality theorem, is simplified and is handled in a more symmetric manner.
- The search for the next reversal is much simpler and requires no special data structures. Our algorithm computes connected components only once, and any simple implementation of it suffices to obtain the quadratic time bound. In contrast, in [4] a logarithmic number of connected component computations may be performed per reversal, using the union-find data structure.

The paper is organized as follows: Section 2 gives the necessary preliminaries. Section 3 gives an overview of our algorithm. Sections 4 and 5 give the details of our algorithm. We summarize our results and suggest some further research in section 6.

2. Preliminaries.

This section gives the basic background, primarily the theory of Hannenhalli and Pevzner, on which we base our algorithm. The reader may find it helpful to refer to Figure 2.1, in which the main definitions are illustrated. We start with some definitions for unsigned permutations. Let $\pi = (\pi_1, \dots, \pi_n)$ denote a permutation of $\{1, \dots, n\}$. Augment π to a permutation on $n+2$ vertices by adding $\pi_0 = 0$ and $\pi_{n+1} = n+1$ to the permutation. A pair (π_i, π_{i+1}), $0 \le i \le n$, is called a *gap.* Gaps are classified into two types. A gap (π_i, π_{i+1}) is a *breakpoint* of π if and only if $|\pi_i - \pi_{i+1}| > 1$; otherwise, it is an *adjacency* of π. We denote by $b(\pi)$ the number of breakpoints in π.

A *reversal*, $\rho(i,j)$, on a permutation π transforms π to

$$\begin{aligned}\pi' &= \pi\rho(i,j)\\ &= (\pi_1, \dots, \pi_{i-1}, -\pi_j, -\pi_{j-1}, \dots, -\pi_i, \pi_{j+1}, \dots, \pi_n).\end{aligned}$$

We say that a reversal $\rho(i,j)$ *acts on* the gaps (π_{i-1}, π_i) and (π_j, π_{j+1}).

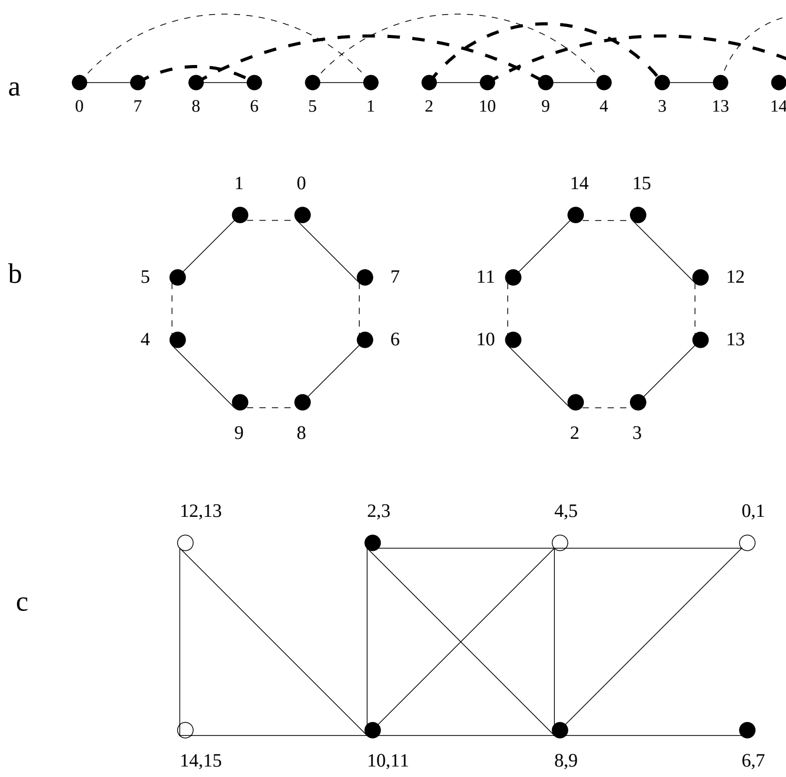


FIG. 2.1. (a) *The breakpoint graph,* $B(\pi)$, *of the permutation* $\pi = (4, -3, 1, -5, -2, 7, 6)$. *Black edges are solid, gray edges are dashed, and oriented edges are bold.* (b) $B(\pi)$ *decomposes into two disjoint alternating cycles.* (c) *The overlap graph,* $OV(\pi)$. *Black vertices correspond to oriented edges.*

2.1. The breakpoint graph. The *breakpoint graph* $B(\pi)$ of a permutation $\pi = (\pi_1, \dots, \pi_n)$ is an edge-colored graph on $n+2$ vertices $\{\pi_0, \pi_1, \dots, \pi_{n+1}\} = \{0, 1, \dots, n+1\}$. We join vertices π_i and π_j by a *black edge* if (π_i, π_j) is a breakpoint in π and by a *gray edge* if (i,j) is a breakpoint in π^{-1}.

We define a one-to-one mapping u from the set of signed permutations of order n into the set of unsigned permutations of order $2n$ as follows. Let π be a signed permutation. To obtain $u(\pi)$, replace each positive element x in π by $2x-1, 2x$ and each negative element $-x$ by $2x, 2x-1$. For any signed permutation π, let $B(\pi) = B(u(\pi))$. Note that in $B(\pi)$ every vertex is either isolated or incident to exactly one black edge and one gray edge. Therefore, there is a unique decomposition of $B(\pi)$ into cycles. The edges of each cycle alternate between gray and black. Refer to a reversal $\rho(i,j)$ such that i is odd and j even as an *even reversal.* The reversal $\rho(2i+1, 2j)$ on $u(\pi)$ mimics the reversal $\rho(i+1, j)$ on π. Thus, sorting π by reversals is equivalent to sorting the unsigned permutation $u(\pi)$ by even reversals. Henceforth we will consider the latter problem, and by a reversal we will always mean an even reversal. Let $b(\pi) = b(u(\pi))$ and let $c(\pi)$ be the number of cycles in $B(\pi)$.

Figure 2.1(a) shows the breakpoint graph of the permutation $\pi = (4, -3, 1, -5, -2, 7, 6)$. It has eight breakpoints and decomposes into two alternating cycles, i.e., $b(\pi) = 8$ and $c(\pi) = 2$. The two cycles are shown in Figure 2.1(b). Figure 2.2(a) shows the breakpoint graph of $\pi' = (4, -3, 1, 2, 5, 7, 6)$, which has seven breakpoints and decomposes into two cycles.

For an arbitrary reversal ρ on a permutation π, define $\Delta b(\pi, \rho) = b(\pi\rho) - b(\pi)$ and $\Delta c(\pi, \rho) = c(\pi\rho) - c(\pi)$. When the reversal ρ and the permutation π are clear from the context, we will abbreviate $\Delta b(\pi, \rho)$ by Δb and $\Delta c(\pi, \rho)$ by Δc. As Bafna and Pevzner [3] observed, the following values are taken by Δb and Δc depending on the types of the gaps $\rho(i,j)$ acts on:

(1) two adjacencies: $\Delta c = 1$ and $\Delta b = 2$;
(2) a breakpoint and an adjacency: $\Delta c = 0$ and $\Delta b = 1$;
(3) two breakpoints each belonging to a different cycle: $\Delta b = 0$, $\Delta c = -1$;
(4) two breakpoints of the same cycle C :
  (a) (π_i, π_{j+1}) and (π_{i-1}, π_j) are gray edges: $\Delta c = -1$, $\Delta b = -2$,
  (b) exactly one of (π_i, π_{j+1}) and (π_{i-1}, π_j) is a gray edge: $\Delta c = 0$, $\Delta b = -1$,
  (c) neither (π_i, π_{j+1}) nor (π_{i-1}, π_j) is a gray edge, and when breaking C at i and j vertices $i-1$ and $j+1$ end up in the same path: $\Delta b = 0$, $\Delta c = 0$,
  (d) neither (π_i, π_{j+1}) nor (π_{i-1}, π_j) is a gray edge, and when breaking C at i and j vertices $i-1$ and $j+1$ end up in different paths: $\Delta b = 0$, $\Delta c = 1$.

Call a reversal *proper* if $\Delta b - \Delta c = -1$, i.e., it is either of type 4a, 4b, or 4d. We say that a reversal ρ *acts on* a gray edge e if it acts on the breakpoints which correspond to the black edges incident with e. A gray edge is *oriented* if a reversal acting on it is proper; otherwise it is *unoriented.* Notice that a gray edge (π_k, π_l) is oriented if and only if $k+l$ is even. For example, the gray edge $(0,1)$ in the graph of Figure 2.1(a) is unoriented, while the gray edge $(7,6)$ is oriented.

2.2. The overlap graph. Two intervals on the real line *overlap* if their intersection is nonempty but neither properly contains the other. A graph G is an *interval overlap graph* if one can assign an interval to each vertex such that two vertices are adjacent if and only if the corresponding intervals overlap (see, e.g., [8]). For a permutation π, we associate with a gray edge (π_i, π_j) the interval $[i,j]$. The *overlap graph* of a permutation π, denoted $OV(\pi)$, is the interval overlap graph of the gray edges of $B(\pi)$. Namely, the vertex set of $OV(\pi)$ is the set of gray edges in $B(\pi)$, and two vertices are connected if the intervals associated with their gray edges overlap. We shall identify a vertex in $OV(\pi)$ with the edge it represents and with its interval in the representation. Thus, the endpoints of a gray edge are actually the endpoints of the interval representing the corresponding vertex in $OV(\pi)$. Note that all the endpoints of intervals in this representation are distinct integers. A connected component of $OV(\pi)$ that contains an oriented edge is called an *oriented component*; otherwise, it is called an *unoriented component.*

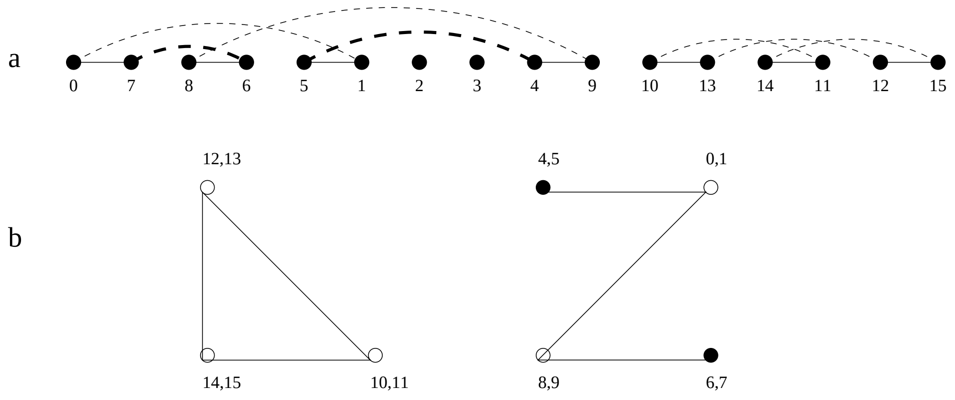


FIG. 2.2. (a) *The breakpoint graph of* $\pi' = (4, -3, 1, 2, 5, 7, 6)$. π' *was obtained from* π *of Figure* 2.1 *by the reversal* $\rho(7, 10)$ *or, equivalently, by the reversal defined by the gray edge* $(2, 3)$. (b) *The overlap graph of* π'.

Figure 2.1(c) shows the interval overlap graph for $\pi = (4, -3, 1, -5, -2, 7, 6)$. It has only one oriented component. Figure 2.2(b) shows the overlap graph of the permutation $\pi' = (4, -3, 1, 2, 5, 7, 6)$, which has two connected components, one oriented and the other unoriented.

2.3. The connected components of the overlap graph. Let X be a set of gray edges in $B(\pi)$. Define $\min(X) = \min\{i \mid (\pi_i, \pi_j) \in X\}$, $\max(X) = \max\{j \mid (\pi_i, \pi_j) \in X\}$, and $span(X) = [\min(X), \max(X)]$. Equivalently, one can look at the interval overlap representation of $OV(\pi)$ mentioned above and define the span of a set of vertices X as the minimum interval which contains all the intervals of vertices in X.

The major object our algorithm will work with is $OV(\pi)$, though for efficiency considerations we will avoid generating it explicitly. In contrast, Pevzner and Hannenhalli worked with the *interleaving graph* H_π, whose vertices are the alternating cycles of $B(\pi)$. Two cycles C_1 and C_2 are connected by an edge in H_π if and only if there exists a gray edge $e_1 \in C_1$ and a gray edge $e_2 \in C_2$ that overlap.

The following lemma and its corollary imply that the partition imposed by the connected components of $OV(\pi)$ on the set of gray edges is identical to the one imposed by the connected components of H_π.

LEMMA 2.1. *If* M *is a set of gray edges in* $B(\pi)$ *that corresponds to a connected component in* $OV(\pi)$, *then* $\min(M)$ *is even and* $\max(M)$ *is odd.*

Proof. Assume $\min(M)$ is odd. Then $\pi_{\min(M)} + 1$ and $\pi_{\min(M)} - 1$ must both be in $span(M)$ (i.e., there exist $l_1, l_2 \in span(M)$ such that $\pi_{l_1} = \pi_{\min(M)} + 1$ and $\pi_{l_2} = \pi_{\min(M)} - 1$). Thus $\pi_{\min(M)}$ is neither the maximum nor the minimum element in the set $\{\pi_i \mid i \in span(M)\}$. Hence, either the maximum element or the minimum element in $span(M)$ is π_j for some $\min(M) < j < \max(M)$. By the definition of $B(\pi)$ there must be a gray edge (π_j, π_l) for some $l \notin span(M)$, contradicting the fact that M is a connected component in $OV(\pi)$. The proof that $\max(M)$ is odd is similar. ∎

As an illustration of Lemma 2.1, consider Figure 2.2(a). Let $M_1 = \{(0, 1), (4, 5), (8, 9), (6, 7)\}$ and $M_2 = \{(10, 11), (12, 13), (14, 15)\}$. Then $span(M_1) = [0, 9]$ and $span(M_2) = [10, 15]$.

COROLLARY 2.2. *Every connected component of* $OV(\pi)$ *corresponds to the set of gray edges of a union of cycles.*

Proof. Assume by contradiction that C is a cycle whose gray edges belong to at least two connected components in $OV(\pi)$. Assume M_1 and M_2 are two of these components such that there are two consecutive gray edges $e_1 \in M_1$ and $e_2 \in M_2$ along C. Since the spans of different connected components in $OV(\pi)$ cannot overlap there are two different cases to consider.

(1) *span*$(M_2) \subseteq$ *span*(M_1) (the case *span*$(M_1) \subseteq$ *span*(M_2) is symmetric). Since e_1 and e_2 are in different components they cannot overlap. Thus, either the right endpoint of e_2 is even and equals $\max(M_2)$ or the left endpoint of e_2 is odd and equals $\min(M_2)$. In both cases we have a contradiction to Lemma 2.1.

(2) *span*(M_2) and *span*(M_1) are disjoint intervals. Without loss of generality assume that $\max(M_1) < \min(M_2)$. The right endpoint of e_1 is even and equals $\max(M_1)$, which contradicts Lemma 2.1. ▯

Note that, in particular, Corollary 2.2 implies that an overlap graph cannot contain isolated vertices.

2.4. Hurdles. Let $\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k}$ be the subsequence of $0, \pi_1, \dots, \pi_n, n+1$ consisting of those elements incident with gray edges that occur in unoriented components of $OV(\pi)$. Order $\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k}$ on a circle CR such that π_{i_j} follows $\pi_{i_{j-1}}$ for $2 \leq j \leq k$ and π_{i_1} follows π_{i_k}. Let M be an unoriented connected component in $OV(\pi)$. Let $E(M) \subset \{\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k}\}$ be the set of endpoints of the edges in M. An unoriented component M is a *hurdle* if the elements of $E(M)$ occur consecutively on CR.

This definition of a hurdle is different from the one given by Hannenhalli and Pevzner [11]. It is simpler in the sense that minimal hurdles and the maximal ones do not have to be treated in different ways. Using Corollary 2.2 above, one can prove that the hurdles as we have defined them are identical to the ones defined by Hannenhalli and Pevzner. Let $h(\pi)$ denote the number of hurdles in a permutation π.

A hurdle is *simple* if when one deletes it from $OV(\pi)$ no other unoriented component becomes a hurdle, and it is a *super hurdle* otherwise. A *fortress* is a permutation with an odd number of hurdles all of which are super hurdles.

The following theorem was proved by Hannenhalli and Pevzner.

THEOREM 2.3 (see [11]). *The minimum number of reversals required to sort a permutation π is $b(\pi) - c(\pi) + h(\pi)$, unless π is a fortress, in which case exactly one additional reversal is necessary and sufficient.*

3. Overview of our algorithm. Denote by $d(\pi)$ the reversal distance of π, i.e., $d(\pi) = b(\pi) - c(\pi) + h(\pi) + 1$ if π is a fortress and $d(\pi) = b(\pi) - c(\pi) + h(\pi)$ otherwise.

Following the theory developed in [11], it turns out that given a permutation π with $h(\pi) > 0$ one can perform $t = \lceil h(\pi)/2 \rceil$ reversals and transform π into a permutation π' such that $h(\pi') = 0$ and $d(\pi') = d(\pi) - t$. If $OV(\pi)$ has unoriented components then our algorithm first finds t such reversals that transform π into a π' which has only oriented components.

Our method of "clearing the hurdles" uses the theory developed by Hannenhalli and Pevzner. In section 5 we describe an efficient implementation of this process which uses the implicit representation of the overlap graph $OV(\pi)$. Our implementation runs in $O(n)$ time assuming $OV(\pi)$ is already partitioned into its connected components. Recently, Berman and Hannenhalli [4] gave an $O(n\alpha(n))$ algorithm for computing the connected components of an interval overlap graph given implicitly by its representation. Using their algorithm we can clear the hurdles from a permutation in $O(n\alpha(n))$ time.

The overlap graph of π', $OV(\pi')$, has only oriented components. In section 4 we prove that in the neighborhood of any oriented gray edge e there is an oriented gray edge e_1 (e_1 could be the same as e) such that a reversal acting on e_1 does not create new hurdles. Call such a reversal a *safe reversal*. We develop an efficient algorithm to locate a safe reversal in a permutation with at least one oriented gray edge. Our algorithm uses only an implicit representation of the overlap graph and runs in $O(n)$ time.

The second stage of our algorithm repeatedly finds a safe reversal and performs it as long as $OV(\pi)$ is not empty. Clearly the overall complexity is $O(r \cdot n + n\alpha(n))$, where r is the number of reversals required to sort π'.

3.1. Representing the overlap graph. We assume that the input is given as a sequence of n signed integers representing π^0. First the permutation $\pi = u(\pi^0)$ is constructed as described in section 2.1 and stored in an array. We also construct an array representing π^{-1}. It is straightforward to verify that with these two arrays we can determine for each element in π whether it is a left or a right endpoint of a gray edge in constant time. In case the element is an endpoint of a gray edge, we can also find the other endpoint and check whether the edge is oriented in constant time.

Thus the arrays π and π^{-1} comprise a representation of $OV(\pi)$. Our algorithm will maintain these two arrays while carrying out the reversals that it finds. The time to update the arrays is proportional to the length of the interval being reversed, which is $O(n)$. We shall give a high-level presentation of our algorithm and use primitives like "Scan the oriented gray edges in increasing left endpoint order." It is easy to see how to implement these primitives using the arrays π and π^{-1}; we shall omit the details.

It is easy to produce a list of the intervals in the representation of $OV(\pi)$ sorted by either the left or right endpoint from the arrays π and π^{-1}. It is also possible to maintain them without increasing the asymptotic time bound of the algorithm. In practice it may be faster to maintain such lists instead of, or in addition to π and π^{-1}.

4. Eliminating oriented components. First we introduce some notation. Recall that the vertices of $OV(\pi)$ are the gray edges of $B(\pi)$. In order to avoid confusion we will usually refer to them as vertices of $OV(\pi)$. Hence a vertex of $OV(\pi)$ is *oriented* if the corresponding gray edge is oriented, and it is *unoriented* otherwise. Let e be a vertex in $OV(\pi)$. Denote by $r(e)$ the reversal acting on the gray edge corresponding to e. Denote by $N(e)$ the set of neighbors of e in $OV(\pi)$ including e itself. Denote by $ON(e)$ the subset of $N(e)$ containing the oriented vertices and by $UN(e)$ the subset of $N(e)$ containing the unoriented vertices.

In this section we prove that if an oriented vertex e exists in $OV(\pi)$ then there exists an oriented vertex $f \in ON(e)$ such that $r(f)$ is proper and safe. We also describe an algorithm that finds a proper safe reversal in a permutation that contains at least one oriented edge.

We start with the following useful observation.

OBSERVATION 4.1. *Let e be a vertex in $OV(\pi)$ and let $\pi' = \pi r(e)$. $OV(\pi')$ could be obtained from $OV(\pi)$ by the following operations.* (1) *Complement the graph induced by $OV(\pi)$ on $N(e) - \{e\}$, and flip the orientation of every vertex in $N(e) - \{e\}$.* (2) *If e is oriented in $OV(\pi)$, then remove it from $OV(\pi)$.* (3) *If there exists an oriented edge e' in $OV(\pi)$ with $r(e) = r(e')$, then remove e' from $OV(\pi)$.*

Note that if e is an oriented vertex in a component M of $OV(\pi)$, $M - \{e\}$ may split into several components in $OV(\pi')$. (Compare Figures 2.1(c) and 2.2(b).) Denote these components by $M'_1(e), \ldots, M'_k(e)$, where $k \geq 1$. We will refer to $M'_i(e)$ simply as M'_i whenever e is clear from the context.

Let C be a clique of oriented vertices in $OV(\pi)$. We say that C is *happy* if for every oriented vertex $e \notin C$ and every vertex $f \in C$ such that $(e, f) \in E(OV(\pi))$ there exists an oriented vertex $g \notin C$ such that $(g, e) \in E(OV(\pi))$ and $(g, f) \notin E(OV(\pi))$. For example, in the overlap graph shown in Figure 2.1(c) $\{(2, 3), (10, 11)\}$ and $\{(6, 7)\}$ are happy cliques, but $\{(2, 3), (10, 11), (8, 9)\}$ is not. Our first theorem claims that one of the vertices in any happy clique defines a safe proper reversal.

THEOREM 4.1. *Let C be a happy clique and let e be a vertex in C such that $|UN(e')| \leq |UN(e)|$ for every $e' \in C$. Then the reversal $r(e)$ is safe.*

Proof. Let $\pi' = \pi r(e)$ and assume by contradiction that $M'_i(e)$ is unoriented for some $1 \leq i \leq k$. Clearly $N(e) \cap M'_i \neq \emptyset$.

Assume there exists $y \in N(e) \cap M'_i$ such that $y \notin C$. Clearly y must be oriented in $OV(\pi)$ and, since C is happy, it must also have an oriented neighbor y' such that $(y', e) \notin E(OV(\pi))$. Since y' is not adjacent to e in $OV(\pi)$ it stays oriented and adjacent to y in $OV(\pi')$, in contradiction with the assumption that M'_i is unoriented. Hence we may assume that $N(e) \cap M'_i \subseteq C$.

Let $y \in N(e) \cap M'_i$ and let $z \in UN(e)$. Vertex z is oriented in $OV(\pi')$ and if it is adjacent to y in $OV(\pi')$ we obtain a contradiction. Hence, z and y are not adjacent in $OV(\pi')$, so they must be adjacent in $OV(\pi)$. Hence we obtain that $UN(e) \subseteq UN(y)$ in $OV(\pi)$. Corollary 2.2 implies that component M'_i cannot contain y alone. Thus y must have a neighbor x in M'_i. Since $N(e) \cap M'_i \subseteq C$, vertex x is not adjacent to e in $OV(\pi)$. Thus we obtain that $(x, y) \in OV(\pi)$, $(x, e) \notin OV(\pi)$, and x is unoriented in $OV(\pi)$. Since we have already proved that $UN(e) \subseteq UN(y)$, this implies that $UN(e) \subset UN(y)$, in contradiction with the choice of e. □

For example, Theorem 4.1 implies that the reversal defined by the gray edge $(10, 11)$ is a safe proper reversal for the permutation of Figure 2.1 (a) since it corresponds to the vertex with maximum unoriented degree in the happy clique $\{(2, 3), (10, 11)\}$. On the other hand, the reversal defined by $(2, 3)$ creates a new unoriented component, as it yields the permutation shown in Figure 2.2.

The following theorem proves that a happy clique exists in the neighborhood of any oriented edge.

THEOREM 4.2. *Let e be an oriented vertex in $OV(\pi)$. There exists an oriented vertex $f \in ON(e)$ such that for $\pi' = \pi r(f)$, all the components in $OV(\pi')$ are oriented.*

Proof. By Theorem 4.1 it suffices to show that there exists a happy clique C in $ON(e)$.

Let $Ext(e) = \{x \in ON(e) \mid$ there exists $y \in ON(x)$ such that $y \notin ON(e)\}$. That is, $Ext(e)$ contains all oriented neighbors of e which have oriented neighbors outside of $ON(e)$.

Case 1. $Ext(e) = ON(e) - \{e\}$. Set $C = \{e\}$.

Case 2. $Ext(e) \subset ON(e) - \{e\}$. Let $D^0 = ON(e) - Ext(e)$. For $j \geq 0$, while D^j is not a clique let K^j be a maximal clique in D^j and define $D^{j+1} = D^j - K^j$. Let D^k, $k \geq 0$ be the final clique and set $C = D^k$.

It is straightforward to verify that in each of the two cases C is indeed a happy clique. □

In the next section we describe an algorithm that will find an oriented edge e such that $r(e)$ is safe given the representation of $OV(\pi)$ described in section 3.1. The algorithm first finds a happy clique C and then searches for the vertex with maximum unoriented degree in C. According to Theorem 4.1 this vertex defines a safe reversal.

Even though Theorem 4.2 guarantees the existence of a happy clique in the neighborhood of any fixed oriented vertex, our algorithm does not search in one particular such neighborhood. We will prove that the algorithm is guaranteed to find a happy clique assuming that there exists at least one oriented edge. Therefore the algorithm provides an alternative proof to a weaker version of Theorem 4.2 that claims only the existence of a happy clique somewhere in the graph.

4.1. Finding a happy clique. In this section we give an algorithm that locates a happy clique in $OV(\pi)$. Let $e_1, \ldots, e_k$ be the oriented vertices in $OV(\pi)$ in increasing left endpoint order. The algorithm traverses the oriented vertices in $OV(\pi)$ according to this order. Let $L(e)$ and $R(e)$ be the left and right endpoints, respectively, of vertex e in the realization of $OV(\pi)$. After traversing $e_1, \ldots, e_i$, $1 \le i \le k$, the algorithm maintains a happy clique C_i in the subgraph of $OV(\pi)$ induced by these vertices. Assume $|C_i| = j$, $j \le i$ and let $e_{i_1}, \ldots, e_{i_j}$ be the vertices in C_i where $i_1 < i_2 < \ldots < i_j$. The vertices of C_i are maintained in a linked list, ordered in increasing left endpoint order. If there exists an interval that contains all the intervals in C_i, then the algorithm maintains a minimal such interval t_i. The clique C_i and the vertex t_i (if it exists) satisfy the following invariant.

INVARIANT 4.1.

(1) Every vertex $e_l \notin C_i$, $l \le i$, such that $L(e_{i_1}) < L(e_l)$ must be adjacent to t_i, i.e., $R(e_l) > R(t_i)$.

(2) Every vertex $e_l \notin C_i$, $L(e_l) < L(e_{i_1})$ that is adjacent to a vertex in C_i is either adjacent to an interval e_p such that $R(e_p) < L(e_{i_1})$ or adjacent to t_i.

The fact that C_i is happy in the subgraph induced by $e_1, \ldots, e_i$ follows from this invariant. We initialize the algorithm by setting $C_1 = \{e_1\}$. Initially, t_1 is not defined. Let the current interval be e_{i+1}. If $R(e_{i_j}) < L(e_{i+1})$ then C_i is guaranteed to be happy in $OV(\pi)$ since all remaining oriented vertices are not adjacent to C_i. Hence the algorithm stops and returns C_i as the answer. See Figure 4.1(a).

We now assume that $L(e_{i+1}) \le R(e_{i_j})$ and show how to obtain C_{i+1} and t_{i+1}. We have to consider the following cases.

Case 1. The interval t_i is defined and $R(t_i) < R(e_{i+1})$. Continue with $C_{i+1} = C_i$ and $t_{i+1} = t_i$. See Figure 4.1(b).

Case 2. The interval t_i is not defined or $R(e_{i+1}) \le R(t_i)$.

(a) $R(e_{i_j}) < R(e_{i+1})$ and $L(e_{i+1}) \le R(e_{i_1})$. C_{i+1} is obtained by adding e_{i+1} to C_i and $t_{i+1} = t_i$. See Figure 4.1(c).

(b) $R(e_{i_j}) < R(e_{i+1})$ and $L(e_{i+1}) > R(e_{i_1})$. The clique C_{i+1} consists of e_{i+1} alone and $t_{i+1} = t_i$. See Figure 4.1(d).

(c) $R(e_{i+1}) < R(e_{i_j})$. As in the previous case $C_{i+1} = \{e_{i+1}\}$. In this case t_{i+1} is set to e_{i_j}, the last interval in C_i. See Figure 4.1(e).

The following theorem proves that the algorithm above produces a happy clique.

THEOREM 4.3. *Let C_l be the current clique when the algorithm stops. Then C_l is a happy clique in $OV(\pi)$.*

Proof. A straightforward induction on the number of oriented vertices traversed by the algorithm proves that C_l and t_l satisfy Invariant 4.1.

The algorithm stops either when $R(e_{i_j}) < L(e_{l+1})$ or when l is equal to the number of oriented vertices. In either case since C_l is happy in the subgraph induced by $e_1, \ldots, e_l$ it must be happy in $OV(\pi)$. ∎

The running time of the algorithm is proportional to the number of oriented vertices traversed since a constant amount of work is performed for each such vertex.

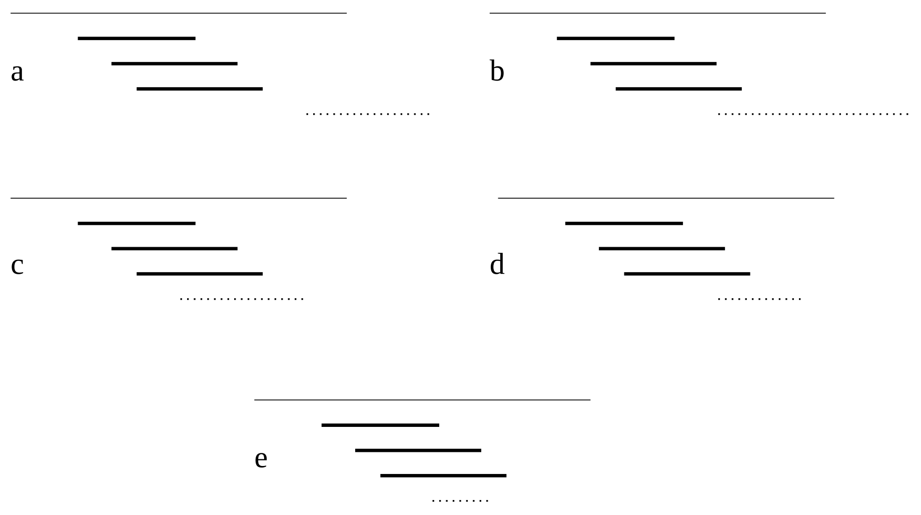


FIG. 4.1. *The various cases of the algorithm for finding a happy clique. The topmost interval is always t_i. The three thick intervals comprise C_i. The dotted interval corresponds to e_{i+1}.*

4.2. Searching the happy clique. After locating a happy clique C in $OV(\pi)$ we need to search it for a vertex with a maximum number of unoriented neighbors. In this section we give an algorithm that performs this task.

Let $e_1, \ldots, e_j$ be the intervals in C ordered in increasing left endpoint order. Clearly, $L(1) < L(2) < \cdots < L(j) < R(1) < R(2) < \cdots < R(j)$. Thus the endpoints of the j vertices in C partition the line into $2j+1$ disjoint intervals $I_0, \ldots, I_{2j}$, where $I_0 = (-\infty, L(1)]$, $I_l = (L(l), L(l+1)]$ for $1 \le l < j$, $I_j = (L(j), R(1)]$, $I_l = (R(l-j), R(l-j+1)]$ for $j < l < 2j$ and $I_{2j} = (R(j), \infty)$. The algorithm consists of the following three stages.

Stage 1. Let e be an unoriented vertex that has a nonempty intersection with the interval $[L(1), R(j)]$. Mark each of e's endpoints with the index of the interval that contains it.

Stage 2. Let o be an array of j counters, each corresponding to a vertex in C. The intention is to assign values to o such that the sum $\sum_{i=1}^{l} o[i]$ is the unoriented degree of the vertex $e_l \in C$. The counters are initialized to zero. For each unoriented vertex e that overlaps with the interval $[L(1), R(j)]$ we change at most four of the counters as follows. Let I_l and I_r be the intervals in which $L(e)$ and $R(e)$ occur, respectively. We may assume $l < r$ as otherwise e is not adjacent to any vertex in C and we can ignore it. We continue according to one of the following cases.

Case 1. $r \le j$. All the vertices from e_{l+1} to e_r are adjacent to e: we increase $o[l+1]$ and decrease $o[r+1]$ (if $r < j$).

Case 2. $j \le l$. All the vertices from e_{l-j+1} to e_{r-j} are adjacent to e: we increase $o[l-j+1]$ and decrease $o[r-j+1]$ (if $r < 2j$).

Case 3. $l < j$ and $j < r$. Let $m = \min\{l, r-j\}$. If $m > 0$ then all the vertices from e_1 to e_m are adjacent to e: we increase $o[1]$ and decrease $o[m+1]$. Similarly let $M = \max\{l, r-j\}$. If $M < j$ then the vertices from e_{l+1} to e_j are adjacent to e: we increase the counter $o[l+1]$.

Stage 3. Compute $f = \max_l \{\sum_{i=1}^{l} o[i] | 1 \le l \le j\}$. Return e_f.

The following theorem summarizes the result of this section. We omit the proof, which is straightforward.

THEOREM 4.4. *Given a clique C, the vertex $e_f \in C$ computed by the algorithm above has maximum unoriented degree among the vertices in C.*

The complexity of the algorithm is proportional to the size of C plus the number of unoriented vertices in $OV(\pi)$, and hence is $O(n)$.

5. Clearing the hurdles.

In case there are unoriented components in $OV(\pi)$, there exists a sequence $r_1, \dots r_t$ of t reversals that transform π into π' such that $d(\pi') = d(\pi) - t$, where $t = \lceil h(\pi)/2 \rceil$. In this section we summarize the characterization given by Hannenhalli and Pevzner for these t reversals and outline how to find them using our implicit representation of $OV(\pi)$.

We will use the following definitions. A reversal *merges* hurdles H_1 and H_2 if it acts on two breakpoints: one incident with a gray edge in H_1 and the other incident with a gray edge in H_2. Recall the circle CR defined in section 2, in which the endpoints of the edges in the unoriented components of $OV(\pi)$ are ordered consistently with their order in π. Two hurdles H_1 and H_2 are *consecutive* if their sets of endpoints $E(H_1)$ and $E(H_2)$ occur consecutively on CR; i.e., there is no hurdle H such that $E(H)$ separates $E(H_1)$ and $E(H_2)$ on CR.

The following lemmas were essentially proved by Hannenhalli and Pevzner though stated differently in their paper.

LEMMA 5.1 (see [11]). *Let π be a permutation with an even number (say $2k$) of hurdles. Any sequence of $k-1$ reversals each of which merges two nonconsecutive hurdles followed by a reversal merging the remaining two hurdles will transform π into π' such that $d(\pi') = d(\pi) - k$ and π' has only oriented components.*

LEMMA 5.2 (see [11]). *Let π be a permutation with an odd number (say $2k+1$) of hurdles. If at least one hurdle H is simple, then a reversal acting on two breakpoints incident with edges in H transforms π into π' with $2k$ hurdles such that $d(\pi') = d(\pi) - 1$. If π is a fortress then a sequence of $k-1$ reversals merging pairs of nonconsecutive hurdles followed by two additional merges of pairs of consecutive hurdles (one merges two original hurdles and the next merges a hurdle created by the first and the last original hurdle) will transform π into π' such that $d(\pi') = d(\pi) - (k+1)$ and π' has only oriented components.*

We now outline how to turn these lemmas into an algorithm that finds a particular sequence of reversals $r_1, \dots, r_t$ with the properties described above. First $OV(\pi)$ is decomposed into connected components as described in [4]. Then one has to identify those unoriented components that are hurdles. This task can be done by traversing the endpoints of the circle CR, counting the number of elements in each run of consecutive endpoints belonging to the same component. If a run contains all endpoints of a particular unoriented component M, then M is a hurdle.

In a similar fashion one can check for each hurdle, whether it is a simple hurdle or a super hurdle. While traversing the cycle, a list of the hurdles in the order they occur on CR is created. At the next stage this list is used to identify the correct hurdles to merge.

We assume that, given an endpoint, one can locate its connected component in constant time. It is easy to verify that the data can be maintained so that this is possible.

THEOREM 5.3. *Given $OV(\pi)$ decomposed into its connected components, the algorithm outlined above finds t reversals such that when we apply them to π we obtain a π' which is hurdle-free and has $d(\pi') = d(\pi) - t$. The algorithm can be implemented to run in $O(n)$ time.*

Proof. Correctness follows from Lemmas 5.1 and 5.2. The time bound is achieved if we always merge hurdles that are separated by a single hurdle. If the ith merge merged hurdles H_1 and H_2 that are separated by H, then H should be merged in the $(i+1)$st merge. Carrying out the merges this way guarantees that the span of each hurdle H overlaps at most two merging reversals, the second of which eliminates H. □

6. Summary.

Figure 6.1 gives a schematic description of the algorithm.

```
algorithm SIGNED REVERSALS($\pi$);
/* $\pi$ is a signed permutation */
1. Compute the connected components of $OV(\pi)$.
2. Clear the hurdles.
3. while $\pi$ is not sorted do :
   /* iteration */
   begin
      a. find a happy clique $C$ in $OV(\pi)$.
      b. find a vertex $e_f \in C$ with maximum unoriented
      degree, and perform a safe reversal on $e_f$;
      c. update $\pi$ and the representation of $OV(\pi)$.
   end
end  4. output the sequence of reversals.
```

FIG. 6.1. *An algorithm for sorting signed permutations.*

THEOREM 6.1. *Algorithm* SIGNED REVERSALS *finds the reversal distance r in $O(n\alpha(n) + r \times n)$ time, and in particular in $O(n^2)$ time.*

Proof. The correctness of the algorithm follows from Theorem 2.3, Theorem 4.1, and Lemmas 5.1 and 5.2.

Step 1 takes $O(n\alpha(n))$ time by the algorithm of Berman and Hannenhalli [4]. Step 2 takes $O(n)$ time by Theorem 5.3. Step 3 takes $O(n)$ time per reversal, by the discussion in section 4. □

It is an intriguing open question whether a faster algorithm for sorting signed permutations by reversals exists. It certainly might be the case that one can find an optimal sequence of reversals faster. To date, no nontrivial lower bound is known for this problem.

Acknowledgments. We thank Donald Knuth, Sridhar Hannenhalli, Pavel Pevzner, and Itsik Pe'er for their comments on a preliminary version of this paper.

REFERENCES

[1] W. ACKERMANN, *Zum hilbertshen aufbau der reelen zahlen*, Math. Ann., 99 (1928), pp. 118–133.
[2] V. BAFNA AND P. PEVZNER, *Sorting permutations by transpositions*, in Proceedings of the 6th Annual Symposium on Discrete Algorithms, ACM, New York, 1995, pp. 614–623.
[3] V. BAFNA AND P. A. PEVZNER, *Genome rearragements and sorting by reversals*, SIAM J. Comput., 25 (1996), pp. 272–289. A preliminary version appeared in Proceedings of the 34th IEEE Symposium of the Foundations of Computer Science, IEEE, Los Alamitos, CA, pp. 148–157, 1994.
[4] P. BERMAN AND S. HANNENHALLI, *Fast sorting by reversals*, in Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 1075, Springer, Berlin, 1996, pp. 168–185.

[5] A. Caprara, *Sorting by reversals is difficult*, in Proceedings of the First International Conference on Computational Molecular Biology (RECOMB), ACM, New York, 1997, pp. 75–83.

[6] D. A. Christie, *A 3/2-approximation algorithm for sorting by reversals*, in Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 98), ACM Press, New York, 1998, pp. 244–252.

[7] T. Dobzhansky and A. H. Sturtevant, *Inversions in the chromosomes of Drosophila pseudoobscura*, Genetics, 23 (1938), pp. 28–64.

[8] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[9] S. Hannenhalli, *Polynomial algorithm for computing translocation distance between genomes*, Discrete Appl. Math., 71 (1996), pp. 137–151.

[10] S. Hannenhalli and P. Pevzner, *Transforming men into mice (polynomial algorithm for genomic distance problems*, in Proceedings of the IEEE Symposium on the Foundations of Computer Science, 1995, pp. 581–592.

[11] S. Hannenhalli and P. A. Pevzner, *Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals)*, in Proceedings of the 27th annual ACM Symposium on Theory of Computing, Las Vegas, NV, 1995, pp. 178–189.

[12] S. B. Hoot and J. D. Palmer, *Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera*, J. Molecular Evolution, 38 (1994), pp. 274–281.

[13] H. Kaplan, R. Shamir, and R. E. Tarjan, *Faster and simpler algorithm for sorting signed permutations by reversals*, in Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, 1997, pp. 344–351. Also in Proceedings of the First International Conference on Computational Molecular Biology (RECOMB), ACM, New York, 1997, p. 163.

[14] J. Kececioglu and R. Ravi, *Physical mapping of chromosomes using unique probes*, in Proceedings of the 6th annual ACM-SIAM Symposium on Discrete Algorithms (SODA 95), ACM Press, New York, 1995, pp. 604–613.

[15] J. Kececioglu and D. Sankoff, *Efficient bounds for oriented chromosome inversion distance*, in Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 807, Springer, Berlin, 1994, pp. 307–325.

[16] J. Kececioglu and D. Sankoff, *Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement*, Algorithmica, 13 (1995), pp. 180–210. A preliminary version appeared in Proceedings CPM 93, Springer, Berlin, 1993, pp. 87–105.

[17] J. D. Palmer and L. A. Herbon, *Tricircular mitochondrial genomes of Brassica and Raphanus: reversal of repeat configurations by inversion*, Nucleid Acids Research, 14 (1986), pp. 9755–9764.

[18] J. D. Palmer and L. A. Herbon, *Unicircular structure of the Brassica hirta mitochondrial genome*, Current Genetics, 11 (1987), pp. 565–570.

[19] J. D. Palmer and L. A. Herbon, *Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence*, J. Molecular Evolution, 28 (1988), pp. 87–97.

[20] J. D. Palmer, B. Osorio, and W. Thompson, *Evolutionalry significance of inversions in legume chloroplast DNAs*, Current Genetics, 14 (1988), pp. 65–74.

[21] D. Sankoff, *Edit distance for genome comparison based on nonlocal operations*, in Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 644, Springer, Berlin, 1992, pp. 121–135.

[22] D. Sankoff, R. Cedergren, and Y. Abel, *Genomic divergence through gene rearrangement.*, Methods in Enzymology, 183 (1990), pp. 428–438.

[23] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. ACM, 22 (1979), pp. 215–225.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 893–911

COMPRESSION OF LOW ENTROPY STRINGS WITH LEMPEL–ZIV ALGORITHMS[*]

S. RAO KOSARAJU[†] AND GIOVANNI MANZINI[‡]

Abstract. We compare the compression ratio of the Lempel–Ziv algorithms with the empirical entropy of the input string. This approach makes it possible to analyze the performance of these algorithms without any assumption on the input and to obtain worst case results. We show that in this setting the standard definition of optimal compression algorithm is not satisfactory. In fact, although Lempel–Ziv algorithms are optimal according to the standard definition, there exist families of *low entropy strings* which are not compressed optimally. More precisely, the compression ratio achieved by LZ78 (resp., LZ77) can be much higher than the zeroth order entropy H_0 (resp., the first order entropy H_1).

For this reason we introduce the concept of λ-optimal algorithm. An algorithm is λ-optimal with respect to H_k if, loosely speaking, its compression ratio is asymptotically bounded by λ times the kth order empirical entropy H_k. We prove that LZ78 cannot be λ-optimal with respect to any H_k with $k \geq 0$. Then, we describe a new algorithm which combines LZ78 with run length encoding (RLE) and is 3-optimal with respect to H_0. Finally, we prove that LZ77 is 8-optimal with respect to H_0, and that it cannot be λ-optimal with respect to H_k for any $k \geq 1$.

Key words. data compression, Lempel–Ziv parsing, empirical entropy

AMS subject classifications. 68Q25, 68P20

PII. S0097539797331105

1. Introduction.

The most widely used data compression algorithms are based on the well known procedures LZ77 and LZ78 [23, 24]. These procedures compress the input string by replacing phrases with a pointer to a previous occurrence of the same phrase. This simple scheme achieves a good compression ratio and both coding and decoding can be done on-line very efficiently.

Given the practical relevance of Lempel–Ziv algorithms, many efforts have been done to analyze their performance. In [23] it is shown that LZ77 is optimal for a certain family of sources, and in [24] it is shown that LZ78 achieves asymptotically the best compression ratio attainable by a finite-state compressor. Other results have been obtained assuming that the input string is a random sequence $\{X_i\}_{-\infty}^{+\infty}$ which is stationary, ergodic, and takes values from a finite alphabet. Under these hypotheses, it has been shown that LZ77 and LZ78 are optimal, that is, their compression rate approaches the entropy of the random sequence [16, 20, 24]. Similar results have been obtained also for other variants of the Lempel–Ziv algorithms [2, 10, 18, 19, 22] and considering different probability distributions for the input string [6, 8].

[*]Received by the editors December 8, 1997; accepted for publication (in revised form) December 16, 1998; published electronically December 15, 1999. A preliminary version of this paper appeared in the Proceedings of the Symposium on Compression and Complexity of Sequences (SEQUENCES), Positano, Italy, IEEE Computer Society, Los Alamitos, CA, 1997.

http://www.siam.org/journals/sicomp/29-3/33110.html

[†]Department of Computer Science, Johns Hopkins University, Baltimore, MD (kosaraju@cs.jhu.edu). The work of this author was supported by NSF under grant CCR-9508545 and ARO under grant DAAH04-96-1-0013.

[‡]Dip. Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, I-15100 Alessandria, Italy, and Istituto di Matematica Computazionale, CNR, Pisa, Italy (manzini@mfn.al.unipmn.it). The work of this author was supported by a CNR short term fellowship and by MURST 40% and 60% funds. Part of this work was done when the author was visiting the Department of Computer Science of the Johns Hopkins University, Baltimore, MD.

More recently, a careful analysis of the Lempel–Ziv parsing rules (see, for example, [7, 17, 21]) and the use of renewal theory have made it possible to estimate the *redundancy* of Lempel–Ziv algorithms, that is, the amount by which the expected compression ratio exceeds the entropy of the source (the redundancy is therefore a measure of the speed at which the compression ratio approaches the entropy). In [9, 14] it is shown that for LZ78 and many of its variants the redundancy is $O\big((\log n)^{-1}\big)$, where n is the length of the input string. In [15] it is shown that for two variants of LZ77 the redundancy is $O(\log\log n/\log n)$. Note that for both families of algorithms we also have bounds on the size of the constants hidden in the big-o notation. Finally, in [21] it is shown that for the fixed database version of LZ78 the redundancy is *lower bounded* by $\Omega(\log\log n/\log n)$.

Although these results have great theoretical value, they are not completely satisfying since the same string s can be generated by ergodic sources with different entropies. In addition, most of these results do not hold for every string but only in the average case or for a set of strings which has probability one in the underlying probabilistic model.

In order to get results which hold for every string, in this paper we analyze the performance of the Lempel–Ziv algorithms under a different perspective. Instead of making assumptions on the input, we consider the so-called *empirical entropy* which is based on a probability distribution defined implicitly by the input string. More precisely, for any string s, we define the kth order entropy $H_k(s)$ by looking at the number of occurrences of each symbol following each k-length substring inside s. We say that an algorithm is *coarsely optimal* if, for all k, there exists a function f_k, with $\lim_{n\to\infty} f_k(n) = 0$, such that for any string s the compression ratio $\rho(s)$ is bounded by

$$\rho(s) \le H_k(s) + f_k(|s|). \tag{1}$$

In other words, an algorithm is coarsely optimal if its compression ratio differs from $H_k(s)$ by a quantity which depends only on the length $|s|$ and vanishes as $|s| \to \infty$. The coarse optimality of LZ78 has been proven by Plotnik et al. [11, Corollary 3] and in section 4 we show that LZ77 is coarsely optimal as well.

Having defined optimality using (1), we can analyze an important phenomenon which went undetected using the previous approaches. If a string s is such that $\lim_{|s|\to\infty} H_k(s) = 0$, it is possible that (1) holds with a function f_k such that $H_k(s) = o(f_k(|s|))$. Hence, we could have a coarsely optimal algorithm with a compression ratio much higher than the entropy. To avoid this counterintuitive phenomenon, it is natural to require that the function f_k is such that $f_k(|s|) = o(H_k(s))$. Unfortunately, it is possible to prove that, even for $k = 0$, neither LZ78 nor LZ77 is optimal according to this more satisfactory definition (Lemmas 3.1 and 4.7). For this reason we introduce the concept of λ-optimality with respect to the kth order entropy H_k. Loosely speaking, the compression ratio of a λ-optimal algorithm must be bounded by $\lambda H_k(s) + o(H_k(s))$. Therefore, a λ-optimal algorithm is guaranteed to compress efficiently also the *low entropy strings*, that is, those strings such that $H_k(s) \to 0$.

In this paper we prove that LZ78 cannot be λ-optimal with respect to any H_k with $k \ge 0$. This is not surprising since it is well known that LZ78 is not able to compress efficiently long runs of identical symbols. Then, we describe an algorithm which combines LZ78 with RLE which is 3-optimal with respect to H_0. Finally, we prove that LZ77 is 8-optimal with respect to H_0, and that it cannot be λ-optimal with respect to H_k for any $k \ge 1$. Our techniques are of general interest since they rely on properties of LZ78 and LZ77 which are shared by other compression algorithms based on parsing such as LZW, LZC, and LZSS (see [1, 13] for a description of these variants).

In [14, 15] Savari analyzes several variants of LZ78 and LZ77 and compares their output size to the empirical entropy of the string. For the compression ratio of these algorithms Savari proves a bound of the form $g\big(k, |s|, H_k(s)\big) H_k(s) +$ lower order terms. The function g is given explicitly for each algorithm, and additional bounds are given for the case $H_k(s) = 0$. The analysis is not limited to the case in which each symbol depends on the k previous symbols, but considers also more general models. However, the results in [14, 15] are valid only if $H_k(s)/|s|$ is bounded away from zero and therefore do not apply to the low entropy strings. For this reason these bounds complement the coarse optimality results for LZ78 and LZ77, but are not directly related with the concept of λ-optimality, which is mainly concerned with the compression of the low entropy strings.

The results of this paper do not aim to substitute results based on probabilistic assumptions. The latter are very deep and rich and they provide information also for the nonprobabilistic setting (see, for example, the results on the empirical entropy in [11, 14, 15]). One of the merits of our worst case analysis is that, as we will see, it provides new insight on Lempel–Ziv algorithms by revealing strengths and weaknesses which cannot be properly analyzed in the probabilistic setting.

2. Definitions and notation.

Let s be a string of length n over the alphabet $\mathcal{A} = \{\alpha_1, \ldots, \alpha_h\}$, and let n_i denote the number of occurrences of the symbol α_i inside s. We define the zeroth order entropy as

$$H_0(s) = -\sum_{i=1}^{h} \frac{n_i}{n} \log\left(\frac{n_i}{n}\right),$$

where we assume $0 \log 0 = 0$. The value $|s|H_0(s)$ represents the output size of an ideal compressor which uses $-\log \frac{n_i}{n}$ bits for coding the symbol α_i. Although, as we will see, it is by no means easy to compress a string up to its zeroth order entropy, often even this is not enough. For example, suppose we want to compress an English text. Clearly, we would like to take advantage not only of the different frequencies of the single letters, but also of the fact that these frequencies depend upon the context. The degree of compression that we can achieve by considering a context of length k, is expressed by the conditional kth order entropy $H_k(s)$ defined as follows. For any string w and $\alpha_i \in \mathcal{A}$, let $n_{w\alpha_i}$ denote the number of occurrences in s of the string w followed by α_i. Let $n_w = \sum_i n_{w\alpha_i}$. We define

$$H_k(s) = -\frac{1}{|s|} \sum_{w \in \mathcal{A}^k} \left(\sum_{i=1}^{h} n_{w\alpha_i} \log\left(\frac{n_{w\alpha_i}}{n_w}\right)\right). \tag{2}$$

The value $|s|H_k(s)$ is the output size of an ideal algorithm which uses $-\log(n_{w\alpha_i}/n_w)$ bits for the symbol α_i when it appears after the "context" w. This means that the code for α_i depends on the k characters preceding it. Note that we are assuming that this algorithm codes the first k characters of s for free. As k increases, the entropy $H_k(s)$ is defined in terms of longer, and therefore more accurate, contexts. It is not difficult to prove that H_k is a decreasing function of k.

Formula (2) is not the only possible definition of entropy in terms of frequencies of characters. Another definition found in the literature is

$$H'_k(s) = -\frac{1}{k} \sum_{w \in \mathcal{A}^k} \frac{n_w}{n} \log\left(\frac{n_w}{n}\right),$$

that is, H'_k is the first order entropy of the k-letter words. Disregarding low order terms, we have $H'_k(s) \simeq \frac{1}{k}[H_1(s) + \cdots + H_k(s)]$ (we get an equality if we consider also the k-letter words which wrap around the string s). Since $H_k(s)$ decreases with k, we have $H'_k(s) \geq H_k(s)$. Since we are interested in bounding the output size of compression algorithms in terms of the entropy, by considering H_k we get stronger results.

Having defined the kth order entropy, we can define the class of optimal compression algorithms. The natural extension of the definition given in the information theoretic setting (see, for example, [5]) is the following.

DEFINITION 2.1. *A compression algorithm* $\mathtt{A}$ *is* coarsely optimal*, if for all* $k \geq 0$ *there exists a function* f_k*, with* $\lim_{n\to\infty} f_k(n) = 0$*, such that for all strings* s *we have*

$$\frac{\mathtt{A}(s)}{|s|} \leq H_k(s) + f_k(|s|), \tag{3}$$

where $\mathtt{A}(s)$ *denotes the output size of algorithm* $\mathtt{A}$ *on input* s*.*

As we have already pointed out, the above definition is not completely satisfactory since, if $H_k(s) \ll f_k(|s|)$, we can have an optimal algorithm for which the compression ratio is much greater than the entropy. For this reason, we introduce a more restrictive definition of optimality in which we require that the difference between the compression ratio and the kth-order entropy is a lower order term.

DEFINITION 2.2. *A compression algorithm* $\mathtt{A}$ *is* λ-optimal *with respect to* H_k*, if it is coarsely optimal and there exists a function* g_k*, with* $g_k(t) = o(t)$*, such that for any string* s *with* $H_k(s) \neq 0$

$$\mathtt{A}(s) \leq \lambda|s|H_k(s) + g_k(\mathtt{A}(s)), \tag{4}$$

where $\mathtt{A}(s)$ *denotes the output size of algorithm* $\mathtt{A}$ *on input* s*.*

Note that (4) implies that $g_k(\mathtt{A}(s)) = o(|s|H_k(s))$. We use (4) since it is easier to prove that a certain quantity is a lower order term with respect to the output size of the algorithm rather than with respect to the usually unknown kth order entropy. Note also that λ-optimality is defined with respect to a single entropy H_k. As we will see, to ask (4) to hold for every $k \geq 0$ would be too strong a requirement. However, a λ-optimal algorithm must be coarsely optimal; hence (3) must hold for every $k \geq 0$. By studying the λ-optimality of an algorithm we measure how well it behaves when the input is a low entropy string; this is something which is not possible using the concept of coarse optimality alone.

In this paper we make use of the following lemma which is a generalization of a similar result proven in [5] for stationary ergodic sequences.

LEMMA 2.3. *Let* $y_1, \ldots, y_t$ *denote a parsing of the string* s *such that each word* y_i *appears at most* M *times. For any* $k \geq 0$ *we have*

$$t \log t \leq |s|H_k(s) + t \log\left(\frac{|s|}{t}\right) + t \log M + \Theta(kt + t)\,. \qquad \square \tag{5}$$

The proof of this lemma is given in the appendix. Note that if the words in the parsing are all distinct the above inequality becomes

$$t \log t \leq |s|H_k(s) + t \log\left(\frac{|s|}{t}\right) + \Theta(kt + t)\,. \tag{6}$$

In the rest of the paper we use the following notation. The length of the output produced by the compression algorithm $\mathtt{A}$ on input s is denoted by $\mathtt{A}(s)$. The compression ratio achieved is $\rho(s) = \mathtt{A}(s)/|s|$. Given a nonempty string s, s^- will denote the string obtained from s by removing the last character. If $|s| > 1$, we set $s^{--} = (s^-)^-$. Given k words $w_1, w_2, \dots, w_k$, their concatenation will be denoted by $w_1 w_2 \cdots w_k$.

3. Compression of low entropy strings with `LZ78`. In this section we consider the algorithm `LZ78`. First, we show that for certain strings `LZ78` compression ratio is well above the zeroth order entropy H_0. Then we prove that by combining `LZ78` with RLE we get a new algorithm which is 3-optimal with respect to H_0.

The algorithm `LZ78` works as follows (see [1] for details). Assuming the words $w_1, w_2, \dots, w_{i-1}$ have been already parsed, `LZ78` selects the ith word as the longest word that can be obtained by adding a single character to a previous word. Hence, $w_i^- = w_j$ for some $j < i$, but unless w_i is the last word in the parsing, we have $w_i \neq w_k$, for all $k < i$. As soon as w_i has been found, `LZ78` outputs an encoding of the pair (j, α_i), where j is such that $w_i^- = w_j$ and α_i is the last character of w_i. If we are working with an alphabet with h symbols, such encoding takes $\lceil \log i \rceil + \lceil \log h \rceil$ bits (all logarithms in this paper are taken to the base 2). Hence, if the input string is parsed into t words, the output of `LZ78` has size $t \log t + t \log h + O(t)$. In the following we assume that all the words are distinct since this greatly simplifies our analysis and changes the word count by at most one.

LEMMA 3.1. *There exists a constant $c > 0$ such that for every $n \geq 1$ there exists a string s of length n satisfying* $\mathtt{LZ78}(s)/|s| \geq c\sqrt{n}H_0(s)$.

Proof. Consider the string $s = 01^{(n-1)}$. `LZ78` parses s into $\Theta(\sqrt{n})$ words, hence the compression ratio is $\Theta((\log n)/\sqrt{n})$, whereas $H_0(s) = (\log n)/n$. □

Despite all the optimality results for `LZ78`, we can hardly say that it compresses optimally the string s. As we have already pointed out, using the standard definition of optimal compression given by (3), the problem is that if $H_k(s) \to 0$ and $H_k(s) = o(f_k(|s|))$, the compression ratio $\rho(s)$ can be much higher than $H_k(s)$.

Lemma 3.1 suggests that the inability of `LZ78` to compress efficiently low entropy strings is due to the inability to cope with long runs of identical symbols. In view of this, it is natural to ask if we can compress optimally also the low entropy strings by combining `LZ78` with RLE. RLE is a technique frequently used when the input may contain long runs of identical symbols. RLE is used either as a fast stand-alone compressor, or as a preprocessing step for more complex compression procedures. We now present an algorithm, which we call $\mathtt{LZ78}_{RL}$, which follows the latter approach and uses RLE as a preprocessing step for `LZ78`.

3.1. The algorithm $\mathtt{LZ78}_{RL}$. In this section we describe and analyze an algorithm which combines `LZ78` with RLE. We underline that our analysis does not rely on the particular parsing realized by `LZ78`. Indeed, the analysis is based only on the two facts that `LZ78` parses the input string into distinct words, and that its output size is $t \log t + O(t)$, t being the number of words in the parsing. Hence, our analysis can be applied also to other parsing-based compression algorithms. With some additional work it can be applied also to the algorithms `LZW` and `LZC` (the core of the Unix utility `compress`) in which the same word can appear in the parsing more than once.

Let s denote a string over the alphabet $\mathcal{A}$, and let s' be a substring of s. We call s' an *α-segment* if it contains only the symbol α, and the two symbols adjacent to s' are different from α. Let $\mathtt{0}$ denote a symbol not belonging to $\mathcal{A}$. For $m > 0$, $\alpha \in \mathcal{A}$, let $B(\alpha^m)$ denote the string obtained by writing m in binary using α as 1 and $\mathtt{0}$ as 0. For example, $B(\alpha\alpha\alpha\alpha\alpha) = \alpha\mathtt{0}\alpha$. Given the string s, we define the run length encoding of s as the string $\tilde{s}$ obtained from s by replacing each δ-segment δ^m with the string $B(\delta^m)$. For example, if $s = \alpha\alpha\alpha\beta\beta\alpha\gamma\gamma\gamma\gamma$, then

$$\tilde{s} = \alpha\alpha\beta\mathtt{0}\alpha\gamma\mathtt{00}.$$

Clearly, given $\tilde{s}$ we can obtain s. In fact, δ is always the first symbol in $B(\delta^m)$, and we know that we have reached the end of a δ-segment when we encounter a symbol different from δ or $\mathtt{0}$. Note that, since $|B(\delta^m)| = \lfloor \log m \rfloor + 1 \le m$, we have $|\tilde{s}| \le |s|$. In the following $\mathtt{LZ78}_{RL}$ will denote the algorithm which applies `LZ78` to the string $\tilde{s}$.

Our implementation of RLE is certainly somewhat different from the more common strategy to use a byte to store a repetition count for the last seen character. However, this simpler strategy is not powerful enough to make `LZ78` overcome the problem outlined in Lemma 3.1. An obvious drawback of our RLE implementation is that it requires an extra character not belonging to $\mathcal{A}$. Note that this is not a problem when RLE is used together with algorithms like `LZW` and `LZC` which never output single characters. We have also devised a RLE strategy in which we do not need to extend the alphabet. However, this alternative encoding is less efficient, and when combined with `LZ78` it produces an algorithm which is 6-optimal with respect to H_0 ($\mathtt{LZ78}_{RL}$ is 3-optimal with respect to H_0; see Theorem 3.6).

THEOREM 3.2. *The algorithm* $\mathtt{LZ78}_{RL}$ *is coarsely optimal with respect to the entropy* $H_k(s)$*, for any* $k \ge 0$.

Proof. Let t denote the number of words in the `LZ78` parsing of the string $\tilde{s}$. In the Appendix (Lemma B.1) we show that this parsing induces a t-word parsing of the string s in which each word appears at most $2\log|s|$ times. By (5) we get

$$t\log t \le |s|H_k(s) + t\log\left(\frac{|s|}{t}\right) + t\log\log|s| + \Theta(kt)\,. \tag{7}$$

Since `LZ78` parses $\tilde{s}$ into distinct words, we have (see, for example, [5, Lemma 12.10.1]) $t = O\Big(\frac{|\tilde{s}|}{\log|\tilde{s}|}\Big) = O\Big(\frac{|s|}{\log|s|}\Big)$. Hence, $\log|s| = O\Big(\frac{|s|}{t}\Big)$, and (7) becomes

$$t\log t \le |s|H_k(s) + 2t\log\left(\frac{|s|}{t}\right) + \Theta(kt)\,.$$

Since $t = O\Big(\frac{|s|}{\log|s|}\Big)$, we get

$$\frac{t\log t}{|s|} \le H_k(s) + O\left(\frac{\log\log|s|}{\log|s|} + \frac{k}{\log|s|}\right),$$

which proves the coarse optimality of $\mathtt{LZ78}_{RL}$. □

We now show that, in addition to being coarsely optimal, the algorithm $\mathtt{LZ78}_{RL}$ compresses almost optimally (with respect to H_0) also the low entropy strings. In our analysis we will use the following notation. Given a string s over the alphabet $\mathcal{A} = \{\alpha_1, \dots, \alpha_h\}$, for $i = 1, \dots, h$, let n_i denote the number of occurrences of α_i in s. We assume that $n_1 = \max_i n_i$, we set $r = n_2 + n_3 + \cdots + n_h$, and $\tau = \frac{|s|}{r}$. Define

$$G(n_1, n_2, \dots, n_h) = |s|H_0(s) = -\sum_{i=1}^{h} n_i \log\left(\frac{n_i}{|s|}\right). \tag{8}$$

By setting $F(x) = x\log x$, we get the following alternative representation of G:

$$G(n_1, \dots, n_h) = F(n_1 + n_2 + \cdots + n_h) - [F(n_1) + F(n_2) + \cdots + F(n_h)]\,. \tag{9}$$

In the following, we will use the two forms $|s|H_0(s)$ and $G(n_1,\dots,n_h)$ interchangeably.

Let t denote the number of words into which LZ78 parses the string s. By (6), we know that

$$\texttt{LZ78}(s) \le |s|H_0(s) + t\log\left(\frac{|s|}{t}\right) + \Theta(t)\,. \tag{10}$$

Using elementary calculus one can easily prove that

$$t\log(|s|/t) \le t\log\tau + \max[G(n_1,\dots,n_h) - t\log t, t\log\log t + 2t],$$

which yields

$$\texttt{LZ78}(s) \le |s|H_0(s) + t\log\tau + O(t\log\log t)\,. \tag{11}$$

The following lemma formalizes the intuitive notion that as $\tau = \frac{|s|}{r}$ increases, the value $|s|H_0(s)$ becomes much smaller than $|s|$.

LEMMA 3.3. *For any string s, we have $H_0(s) \le \frac{\log(\tau h e)}{\tau}$.*

Proof. Let $n = |s|$. Using elementary calculus we get

$$\begin{aligned} G(n_1,\dots,n_h) &= G(n_1, n-n_1) + G(n_2,\dots,n_h)\\ &\le G(n-n/\tau, n/\tau) + (n/\tau)\log h\\ &= (n/\tau)[F(\tau) - F(\tau-1) + \log h]\\ &\le (n/\tau)[\log(\tau e) + \log h], \end{aligned}$$

where we have used that F is convex and $F'(t) = \log(et)$. ∎

The following two lemmas bound the entropy and the length of the "encoded" string $\tilde{s}$ in terms of the entropy of s.

LEMMA 3.4. *For any string s, we have $|\tilde{s}|H_0(\tilde{s}) \le |s|H_0(s) + |\tilde{s}|$.*

Proof. For $i = 1,\dots,h$, let m_i denote the number of occurrences of α_i inside $\tilde{s}$, and z_i denote the number of 0's created converting the α_i-segments. Let $v_i = m_i + z_i$. By (9), we have

$$\begin{aligned} |\tilde{s}|H_0(\tilde{s}) &= G\left(m_1,\dots,m_h,\sum_{i=1}^h z_i\right)\\ &= F\left(m_1+\cdots+m_h+\sum_{i=1}^h z_i\right) - \left[\sum_{i=1}^h F(m_i) + F\left(\sum_{i=1}^h z_i\right)\right]\\ &\le F(v_1+\cdots+v_h) - \sum_{i=1}^h [F(m_i) + F(z_i)]. \end{aligned}$$

By Jensen inequality, we know that $F(m_i) + F(z_i) \ge 2F(v_i/2) = F(v_i) - v_i$, hence

$$\begin{aligned} |\tilde{s}|H_0(\tilde{s}) &\le F(v_1+\cdots+v_h) - \sum_{i=1}^h [F(v_i) - v_i]\\ &= G(v_1,\dots,v_h) + \sum_{i=1}^h v_i\\ &\le |s|H_0(s) + |\tilde{s}|. \qquad ∎ \end{aligned}$$

LEMMA 3.5. *For any string s, we have $|\tilde{s}| \le 2|s|H_0(s)$.*

Proof. Let $n_1 = \max_i n_i$, $r = |s| - n_1 = n_2 + \cdots + n_h$. If $n_1 \le |s|/2$, we have

$$|s|H_0(s) = \sum_{i=1}^{h} n_i \log\left(\frac{|s|}{n_i}\right) \ge \sum_{i=1}^{h} n_i = |s| \ge |\tilde{s}|,$$

and the lemma follows. If $n_1 > |s|/2$, then

$$\begin{aligned}|s|H_0(s) &\ge n_1 \log\left(\frac{n_1 + r}{n_1}\right) + \sum_{i=2}^{h} n_i \log\left(\frac{|s|}{r}\right)\\ &= r \log\left(1 + \frac{r}{n_1}\right)^{\frac{n_1}{r}} + r \log\left(\frac{|s|}{r}\right).\end{aligned}$$

Since, $(1 + 1/t)^t \ge 2$, for $t \ge 1$, we have

$$|s|H_0(s) \ge r + r \log\left(\frac{|s|}{r}\right). \tag{12}$$

In order to bound $|\tilde{s}|$, we note that the string s contains at most $r + 1$ α_1-segments. Hence,

$$|\tilde{s}| \le \sum_{i=1}^{r+1} (\log q_i + 1) + \sum_{i=2}^{h} n_i,$$

with $q_1 + \cdots + q_{r+1} = n_1$. From the concavity of $\log t$ we obtain

$$|\tilde{s}| \le (r + 1) \log\left(\frac{n_1}{r + 1}\right) + 2r + 1.$$

For $r = 1$, we have

$$\frac{|\tilde{s}|}{2|s|H_0(s)} \le \frac{2 \log n_1 + 1}{2 + 2 \log(n_1 + 1)} \le 1.$$

For $r \ge 2$, we set $t = n_1/r$, and we get

$$\frac{|\tilde{s}|}{2|s|H_0(s)} \le \frac{2r + (r + 1) \log t + 1}{2r + 2r \log(1 + t)}.$$

A straightforward computation shows that for $t \ge 1$, $2r \log(1 + t) \ge (r + 1) \log t + 1$, and the lemma follows. ◻

We are now ready to establish the main result of this section.

THEOREM 3.6. *The algorithm* $\texttt{LZ78}_{RL}$ *is 3-optimal with respect to H_0.*

Proof. We have already shown that $\texttt{LZ78}_{RL}$ is coarsely optimal (Theorem 3.2). Hence, we need to prove that for any string s with $H_0(s) \ne 0$ we have

$$\texttt{LZ78}_{RL}(s) \le 3|s|H_0(s) + \text{lower order terms.}$$

By (11) we know that

$$\texttt{LZ78}_{RL}(s) \le |\tilde{s}|H_0(\tilde{s}) + t \log \tau + O(t \log \log t),$$

where t denotes the number of words in the LZ78 parsing of the string $\tilde{s}$. If $\tau \le 16h$ the theorem trivially holds, since, by Lemmas 3.4 and 3.5, we have $|\tilde{s}|H_0(\tilde{s}) \le 3|s|H_0(s)$. Assume now $\tau > 16h$. By (10), we know that

$$\mathtt{LZ78}_{RL}(s) \le |\tilde{s}|H_0(\tilde{s}) + t\log\left(\frac{|\tilde{s}|}{t}\right) + O(t)\,.$$

It is straightforward to verify that, for $0 < x < |\tilde{s}|$, we have $x\log(\tilde{s}/x) \le (|\tilde{s}|\log e)/e \le (2|\tilde{s}|)/3$. Hence, we can write

$$\mathtt{LZ78}_{RL}(s) \le |\tilde{s}|H_0(\tilde{s}) + (2|\tilde{s}|)/3 + O(t)\,.$$

Since $\tau > 16h$, we have $\frac{\log(\tau he)}{\tau} \le \frac{5}{6}$. Thus, by Lemmas 3.3 and 3.5 we have

$$\begin{aligned}\mathtt{LZ78}_{RL}(s) &\le |\tilde{s}|\left(\frac{\log(\tau he)}{\tau}\right) + \frac{2}{3}|\tilde{s}| + O(t)\\ &\le \frac{3}{2}|\tilde{s}| + O(t)\\ &\le 3|s|H_0(s) + O(t)\,.\end{aligned}$$

This completes the proof. ∎

4. Compression of low entropy strings with LZ77**.** In this section we consider the algorithm LZ77. First, we prove that this algorithm is coarsely optimal according to Definition 2.1. This is not surprising since it is known that LZ77 compresses much better than LZ78. In view of the results of section 3, what is more interesting is to understand how well LZ77 compresses the low entropy strings. To this end, we prove that LZ77 is 8-optimal with respect to H_0. We also show that this result cannot be substantially improved: we present a family of low entropy strings for which $\mathtt{LZ77}(s) \ge 2.5|s|H_0(s)$ and we prove that LZ77 cannot be λ-optimal with respect to H_1.

The algorithm LZ77 works as follows (see [1] for details). Assuming the words $w_1, w_2, \dots, w_{i-1}$ have been already parsed, LZ77 selects the ith word as the longest word that can be obtained by adding a single character to a substring of $(w_1w_2\cdots w_i)^{--}$. Hence, w_i has the property that w_i^- is a substring of $(w_1\cdots w_i)^{--}$, but w_i is not a substring of $(w_1\cdots w_i)^-$. Note that although this is a recursive definition there is no ambiguity. In fact, if $|w_i| > 1$ at least the first character of w_i belongs to $w_1w_2\cdots w_{i-1}$. Once w_i has been found, LZ77 outputs an encoding of the triplet (p_i, l_i, α_i), where p_i is the starting position of w_i^- within $w_1w_2\cdots w_{i-1}$, $l_i = |w_i|$, and α_i is the last character of w_i.

Compared to LZ78, the algorithm LZ77 parses the input into fewer words and generally achieves a better compression. In addition, it still has the nice property that both coding and decoding can be done in $O(|s|)$ time (see [12]). Note that in the original formulation of LZ77 [23] p_i is allowed to denote a position only within the last L characters of $w_1w_2\cdots w_{i-1}$ (the so called "sliding window"). The use of a sliding window makes the coding procedure faster, and in most cases it affects the compression ratio only slightly. However, as we will see (Lemma 4.3), if the window has a fixed size the algorithm cannot be coarsely optimal.

4.1. Coarse optimality of LZ77**.** To analyze the compression ratio achieved by LZ77 we need to specify the encoding of the triplet (p_i, l_i, α_i) representing the ith word in the parsing of s. For our analysis we assume the following simple scheme.

Since $1 \le p_i \le \sum_{j<i} l_j$, we use $\left\lceil \log(\sum_{j<i} l_j) \right\rceil$ bits for encoding p_i. Assuming the input alphabet has h symbols, we use $\lceil \log h \rceil$ bits for encoding α_i. Finally, since we cannot bound in advance the size of l_i, we use a scheme which allows the encoding of unbounded integers. More precisely, we code l_i using $1 + \lfloor \log l_i \rfloor + 2\lfloor \log(1 + \lfloor \log i \rfloor) \rfloor$ bits (for details on this and other similar schemes, see, for example, [3]). Although there are other possible methods for describing these triplets, we believe our analysis can be adapted to all "reasonable" encodings.

Assuming we use the above encoding scheme for each word in the parsing, the output of LZ77 has size

$$\mathtt{LZ77}(s) = \sum_{i=1}^{t} \Big(\log(\textstyle\sum_{j<i} l_j) + \log l_i + 2\log(1 + \log l_i) \Big) + O(t)\,.$$

Obviously, $(\sum_{j<i} l_j) \le |s|$. In addition, from the concavity of the function $\log x + 2\log(1+\log x)$, we get $\sum_i (\log l_i + 2\log(1+\log l_i)) \le t[\log(|s|/t) + 2\log(1+\log(|s|/t))]$. Hence,

$$\mathtt{LZ77}(s) \le t\log|s| + t\log\left(\frac{|s|}{t}\right) + O(t\log\log(|s|/t))\,,$$

or, equivalently,

$$\mathtt{LZ77}(s) \le t\log t + 2t\log\left(\frac{|s|}{t}\right) + O(t\log\log(|s|/t))\,. \tag{13}$$

Since LZ77 parses its input into distinct words, we can use (6) which yields

$$\mathtt{LZ77}(s) \le |s|H_k(s) + 3t\log\left(\frac{|s|}{t}\right) + O\left(kt + t\log\log\left(\frac{|s|}{t}\right)\right). \tag{14}$$

We can now easily prove that LZ77 is optimal according to Definition 2.1.

THEOREM 4.1. *The algorithm* LZ77 *is coarsely optimal.*

Proof. Let t denote the number of words in the LZ77 parsing. Since the words are distinct, we have $t = O\Big(\frac{|s|}{\log|s|}\Big)$. From (14) we get

$$\frac{\mathtt{LZ77}(s)}{|s|} \le H_k(s) + O\bigg(\frac{\log\log|s|}{\log|s|} + \frac{k}{\log|s|}\bigg)\,,$$

which proves the optimality of LZ77. ∎

Despite the above optimality result, the following lemma shows that, if $H_1(s) \to 0$, LZ77 compression ratio can be asymptotically greater than H_1.

LEMMA 4.2. *There exists a constant $c > 0$ such that for every $n > 1$ there exists a string s of length $|s| \ge n$ satisfying*

$$\frac{\mathtt{LZ77}(s)}{|s|} \ge c\,\frac{\log|s|}{\log\log|s|}H_1(s).$$

Proof. Consider the following string:

$$s = 1\,0^k\,2^{2^k}\,1\,101\,10^21\,10^31\,10^41\cdots10^k1.$$

We have $|s| = 2^k + O(k^2)$. A simple computation shows that $|s|H_1(s) = k \log k + O(k)$. The LZ77 parsing of s is

$$1 \quad 0 \quad 0^{(k-1)}2 \quad 2^{(2^k-1)}1 \quad 101 \quad 10^21 \quad 10^31 \quad \cdots \quad 10^k1.$$

For $i \geq 5$, we have $\sum_{j<i} l_j > 2^k$. Hence, the encoding of p_i takes $\Omega(k)$ bits. Since there are $k+4$ words, we have $\mathtt{LZ77}(s) = \Omega(k^2)$ and the lemma follows. ∎

Let us comment on the above lemma. We can clearly see that the inefficiency of LZ77 is due to the cost of encoding p_i, and it is possible that, using a clever encoding, we can reduce LZ77 output size significantly. However, for any encoding scheme we have been able to think of, it was always possible to find a "bad" string which LZ77 is not able to compress up to the first order entropy. Lemma 4.2 also shows that it is sometimes not convenient to search for the longest match over the entire parsed portion of the input string. Indeed, a common practice is to restrict the search to a sliding window containing the last L characters seen by the algorithm (this is in fact the original formulation of LZ77 found in [23]). This strategy, which we call $\mathtt{LZ77}_L$, usually reduces dramatically the cost of encoding the pointers p_i's, at the expense of a moderate increment in the number of words. For example, for $k < L < 2^k$, $\mathtt{LZ77}_L$ parses the string given in the proof of Lemma 4.2 in $k+5$ words. The output size is $\approx k(\log k + \log L)$ which, for $L = k^{O(1)}$, differs from $|s|H_1(s)$ only by a constant factor. Unfortunately, the following lemma shows that $\mathtt{LZ77}_L$ is not coarsely optimal since it sometimes fails to compress up to the kth order entropy when $k \geq L-1$.

LEMMA 4.3. *The algorithm* $\mathtt{LZ77}_L$ *is not coarsely optimal for any* $L > 0$.

Proof. For $k = L-1$ and $n > 0$, we exhibit a string s of length $2kn+1$ such that $|s|H_k(s) = \Theta(\log n)$ and $\mathtt{LZ77}_L(s) = \Theta(n)$. Let

$$s = (0^k1^k)^n1.$$

The $\mathtt{LZ77}_L$ parsing of s consists of the following $2n+1$ words

$$0 \quad 0^{k-1}1 \quad 1^{k-1}0 \quad 0^{k-1}1 \quad \cdots \quad 1^{k-1}0 \quad 0^{k-1}1 \quad 1^k.$$

To see this, consider for example the situation after the parsing of the third word. The unparsed portion of the string is $0^{k-1}1^k0^k1^k0^k\cdots$. Since at that moment the sliding window contains 1^k0, the only possible match is the one starting at the last character of the sliding window. Hence, every word has length at most k and $\mathtt{LZ77}_L(s) = \Theta(n)$. Note that LZ77 parses the above string into four words only.

For the computation of $H_k(s)$ we notice that s contains only $2k$ distinct k-letter words, namely

$$0^i1^{k-i}, \quad i = 1, \ldots, k \qquad \text{and} \qquad 1^i0^{k-i} \quad i = 1, \ldots, k.$$

In addition, every occurrence of a word 0^i1^{k-i} is always followed by a 1, and, for $i < k$, every occurrence of 1^i0^{k-i} is followed by a 0. Finally, the word 1^k is followed $n-1$ times by a 0 and once by a 1. As a result, we have $|s|H_k(s) = \Theta(\log n)$ and the lemma follows. ∎

4.2. λ-optimality of LZ77.

We now show that LZ77 is 8-optimal with respect to H_0. We start our analysis with the following lemma which establishes an optimality result for the number of words in the LZ77 parsing.

LEMMA 4.4. *Let* $w_1, w_2, \ldots, w_t$ *denote the* LZ77 *parsing of the string* s. *Suppose* $y_1, y_2, \ldots, y_{t'}$ *is another parsing of* s *such that, for* $i = 1, \ldots, t'$, $|y_i| = 1$ *or* y_i^- *is a substring of* $(y_1 \cdots y_i)^{--}$. *Then,* $t \leq t'$.

Proof. One can easily prove by induction that, for $j = 1, \ldots, t$, the string $y_1 y_2 \cdots y_j$ is a prefix of $w_1 w_2 \cdots w_j$. ∎

Given a string s, in the following n_i will denote the number of occurrences of α_i in s. We assume $n_1 = \max_i n_i$, and we set $r = |s| - n_1$. The following lemma provides a bound on the number of words in the `LZ77` parsing in terms of r. The bound is tight when $r \ll |s|$.

LEMMA 4.5. *Let t denote the number of words in which the algorithm* `LZ77` *parses the string s. We have $t \leq 2(r+1)$.*

Proof. We prove the lemma by showing that there exists a parsing $y_1 y_2 \cdots y_{t'}$, $t' \leq 2(r+1)$, for s which satisfies the hypothesis of Lemma 4.4. Let k denote the number of α_1-segments appearing inside s. Since there are r characters different from α_1 we have $1 \leq k \leq r+1$. Consider the following parsing. Each α_1-segment and the character following it is parsed in two words: the first consisting of the single character α_1, the second of the form $\alpha_1^k \alpha_j$. After this, we are left with at most $r-k+1$ unparsed characters, all of them different from α_1. By parsing these characters using length-one words, we get a parsing for s consisting of $2k + (r-k+1) \leq 2(r+1)$ words. ∎

We are now ready to establish the main result of this section.

THEOREM 4.6. *The algorithm* `LZ77` *is* 8*-optimal with respect to H_0.*

Proof. We have already shown that `LZ77` is coarsely optimal (Theorem 4.1). Hence, we need to prove that, for any string s with $H_0(s) \neq 0$, we have

$$\texttt{LZ77}(s) \leq 8|s|H_0(s) + \text{lower order terms.}$$

Let t denote the number of words in the `LZ77` parsing of s, and let r be defined as in Lemma 4.5. We consider four cases.

Case 1. $1 \leq r < 6$.

By Lemma 4.5 we know that $t \leq 12$. Using (13) we get

$$\texttt{LZ77}(s) \leq 2t\left(\log \frac{|s|}{t}\right) + O(t \log\log(|s|/t)). \tag{15}$$

Let $g(x) = x\log(|s|/x)$. For $0 < x < (|s|/e)$, we have $g'(x) > 0$. Since $r+1 \leq 2r$, we get

$$t \log\left(\frac{|s|}{t}\right) \leq 4r \log\left(\frac{|s|}{r}\right) \leq 4\left[\sum_{i=2}^{h} n_i \log\left(\frac{|s|}{n_i}\right)\right] \leq 4|s|H_0(s).$$

Combining this inequality with (15) we get the thesis.

Case 2. $6 \leq r < 3|s|/7e$.

By Lemma 4.5 we know that $t \leq 2(r+1) \leq \frac{7}{3}r \leq |s|/e$. Reasoning as in Case 1 we get

$$t \log\left(\frac{|s|}{t}\right) \leq \frac{7}{3}\left[r \log\left(\frac{|s|}{r}\right)\right] \leq \frac{7}{3}|s|H_0(s).$$

Substituting this inequality into (14) we get the thesis.

Case 3. $3|s|/7e \leq r < |s|/2$.

Let $G(n_1, n_2)$ be defined as in (8). We have

$$|s|H_0(s) \geq G(|s|-r, r) = |s|G(1 - \tfrac{r}{|s|}, \tfrac{r}{|s|}) \geq |s|G\left(1 - 3/(7e), 3/(7e)\right) \geq |s|/2.$$

By elementary calculus we know that, for $0 < x < |s|$, $x \log(|s|/x) \leq (|s|/e) \log e \leq 2|s|/3$. Hence

$$t \log\left(\frac{|s|}{t}\right) \leq \frac{2|s|}{3} \leq \frac{4}{3}|s|H_0(s).$$

Substituting this inequality into (14) we get the thesis.

Case 4. $r \geq |s|/2$.

Since, for $i = 1, \ldots, h$, $n_i < |s|/2$ we have

$$|s|H_0(s) = \sum_{i=1}^{h} n_i \log\left(\frac{|s|}{n_i}\right) \geq |s|.$$

Combining this inequality with the fact that $t \log(|s|/t) \leq 2|s|/3$ we get $\texttt{LZ77}(s) \leq 3|s|H_0(s) + O(t \log\log(\frac{|s|}{t}))$, and the theorem follows. □

Finally, we show that the above theorem cannot be substantially improved since there exists an infinite family of strings such that the compression ratio of LZ77 is greater than $2.5H_0$.

Lemma 4.7. *For every n, there exists a string s of length $|s| \geq n$ such that $\frac{\texttt{LZ77}(s)}{|s|} \geq 2.5H_0(s)$.*

Proof. Consider the following string

$$s = 010^4 10^9 1 \cdots 10^{i^2} 1 \cdots 10^{k^2} 1.$$

We have $|s| = k^3/3 + O\big(k^2\big)$, and $|s|H_0(s) = 2k \log k + O(k)$. The LZ77 parsing of s is

$$0 \quad 1 \quad 00 \quad 0^2 10^5 \quad 0^4 10^{10} \quad \cdots \quad 0^{2(i-1)} 10^{i^2+1} \quad \cdots \quad 0^{2(k-2)} 10^{(k-1)^2+1} \quad 0^{2(k-1)} 1.$$

Let p_i and l_i denote, respectively, the starting position and the length of the ith word. We have $l_i = \Theta\big(i^2\big)$. Hence, neglecting the log log term, encoding l_i takes $\approx 2 \log i + \log\log i$ bits. Similarly, since $\sum_{j<i} l_j = \Theta\big(i^3\big)$, encoding p_i takes $\approx 3 \log i$ bits. Neglecting lower order terms, the output size of LZ77 is therefore $5(\sum_{i \leq k} \log i) \approx 5k \log k$ bits, and the lemma follows. □

5. Conclusions. In order to analyze the performance of the Lempel–Ziv algorithms without any assumption on the input, we have compared the compression ratio of LZ77 and LZ78 with the so-called empirical entropy of the input string. We have shown that the standard definition of optimal compression does not take into account the performance of compression algorithms when the input is a low entropy string. For this reason we have introduced the concept of λ-optimality which makes it possible to measure how well an algorithm performs when the input is a low entropy string. We have proven that by combining LZ78 with RLE we get an algorithm which is 3-optimal with respect to H_0, and that LZ77 is 8-optimal with respect to H_0.

A natural open question is whether there exist parsing-based compression algorithms which are λ-optimal with respect to H_k for $k \geq 1$. Theorem 4.2 shows that LZ77 is not λ-optimal for $k \geq 1$ and one can easily verify that the same is true for $\texttt{LZ78}_{RL}$ as well. Theorem 4.2 also shows that to improve LZ77 performance one should try to reduce the cost of the backward pointers. Some interesting ideas in this direction are described in [1, section 3.4] and [4]. Lemma 4.3 shows that the simple use of a fixed size sliding window does not yield an optimal algorithm. However, the algorithm $\texttt{LZ77}_L$ deserves further investigation for several reasons. First, it is the variant which is the basis for the most popular compressors (`zip`, `gzip`, `arj`, `lha`, `zoo`, etc.). Second, we have been able to show that its compression ratio can be higher than H_k only when $k \geq L-1$. Since in typical implementations $L \approx 10^5$, our result has only a theoretical value. An interesting open problem is to prove or disprove that for $\texttt{LZ77}_L$ inequalities (3) and (4) hold for $k \leq \theta(L)$ for some appropriate function θ.

Appendix A. Proof of Lemma 2.3. In this appendix we prove Lemma 2.3. Our proof follows closely the proof given in [5, section 12.10] for a similar result involving the entropy of a stationary ergodic source.

Let $s = x_1 x_2 \cdots x_{n+k}$ be a string of length $n+k$. For $w \in \mathcal{A}^k$ and $\alpha \in \mathcal{A}$ let $n_{w\alpha}$ and n_w be defined as in section 2. We define the empirical probability $P(w\alpha)$ as $P(w\alpha) = n_{w\alpha}/n_w$ (if $n_w = 0$ we set $P(w\alpha) = 0$). For $i > k$ let s_i denote the length-k string preceding x_i in s. Using this notation, the kth order entropy can be written as

$$|s|H_k(s) = -\sum_{i=k+1}^{n+k} \log P(s_i x_i). \tag{16}$$

Let $v = v_1 v_2 \cdots v_h$ and $w = w_1 \cdots w_k$ be two strings over $\mathcal{A}$. For $i = 1, \dots, h$ let $w^{(i)}$ denote the length-k string preceding the symbol v_i in wv (for example, $w^{(2)} = w_2 \cdots w_k v_1$). We define

$$\mathcal{P}(wv) = \prod_{i=1}^{h} P(w^{(i)} v_i). \tag{17}$$

The value $\mathcal{P}(wv)$ corresponds to the conditional probability $P(v|w)$ introduced in Lemma 12.10.3 of [5]. The following lemma shows that, in some sense, $\mathcal{P}$ does behave as a conditional probability.

LEMMA A.1. *For any string w and $l \geq 1$ we have $\sum_{v \in \mathcal{A}^l} \mathcal{P}(wv) \leq 1$.*

Proof. We prove the result by induction on l. If $l = 1$ we have $\sum_{\alpha \in \mathcal{A}} \mathcal{P}(w\alpha) = \sum_{\alpha \in \mathcal{A}} P(w\alpha) \leq 1$. For $l > 1$, let $v = v_1 \cdots v_l$ and $v' = v_2 \cdots v_l$. We have

$$\begin{aligned}
\sum_{v \in \mathcal{A}^l} \mathcal{P}(wv) &= \sum_{v \in \mathcal{A}^l} P(wv_1) \mathcal{P}(w^{(2)} v') \\
&= \sum_{\alpha \in \mathcal{A}} P(w\alpha) \left(\sum_{v' \in \mathcal{A}^{l-1}} \mathcal{P}(w^{(2)} v') \right) \\
&\leq \sum_{\alpha \in \mathcal{A}} P(w\alpha) \\
&\leq 1. \qquad \square
\end{aligned}$$

COROLLARY A.2. *Let $V = \{v_1, \dots, v_m\}$ be a collection of length-l strings in which each v_i appears at most M times. For any string w we have*

$$\sum_{v \in V} \mathcal{P}(wv) \leq M. \qquad \square$$

Let $y_1, \dots, y_c$ denote any parsing of the string $x_{k+1} \cdots x_{n+k}$, and let z_i denote the length-k substring preceding y_i in $s = x_1 \cdots x_{n+k}$. For $l \geq 1$ and $w \in \mathcal{A}^k$ we define c_{lw} as the number of words y_i such that $|y_i| = l$ and $z_i = w$. By construction, the values c_{lw} satisfy

$$\sum_{l,w} c_{lw} = c, \qquad \sum_{l,w} l c_{lw} = n. \tag{18}$$

The following lemma is a generalization of Ziv's inequality [5, Lemma 12.10.3].

LEMMA A.3. *Let $y_1, \ldots, y_c$ denote a parsing of the string $x_{k+1} \cdots x_{n+k}$ in which each word appears at most M times. We have*

$$|s|H_k(s) \geq \sum_{w \in \mathcal{A}^k,\, l \geq 1} c_{lw} \log c_{lw} - c \log M.$$

Proof. By (16) and (17) we have

$$\begin{aligned}
|s|H_k(s) &= -\sum_{i=k+1}^{n+k} \log P(s_i x_i) \\
&= -\sum_{i=1}^{c} \log \mathcal{P}(z_i y_i) \\
&= -\sum_{l,w} \sum_{|y_i|=l,\ z_i=w} \log \mathcal{P}(z_i y_i) \\
&= -\sum_{l,w} c_{lw} \left(\sum_{|y_i|=l,\ z_i=w} \frac{1}{c_{lw}} \log \mathcal{P}(z_i y_i) \right).
\end{aligned}$$

By Jensen's inequality and Corollary A.2 we get

$$\begin{aligned}
|s|H_k(s) &\geq -\sum_{l,w} c_{lw} \log\left(\sum_{|y_i|=l,\ z_i=w} \frac{1}{c_{lw}} \mathcal{P}(z_i y_i) \right) \\
&\geq -\sum_{l,w} c_{lw} \left(\log \frac{1}{c_{lw}} + \log\left(\sum_{|y_i|=l,\ z_i=w} \mathcal{P}(z_i y_i) \right) \right) \\
&\geq \sum_{l,w} c_{lw} \log c_{lw} - c \log M. \qquad \square
\end{aligned}$$

THEOREM A.4. *Let $y_1, \ldots, y_c$ denote a parsing of the string $x_{k+1} \cdots x_{n+k}$ in which each word appears at most M times. We have*

$$c \log c \leq |s|H_k(s) + c \log M + c \left(k \log |\mathcal{A}| + \log \frac{n}{c} \right) + \Theta(c).$$

Proof. Let $\pi_{lw} = c_{lw}/c$. We have

$$\begin{aligned}
\sum_{l,w} c_{lw} \log c_{lw} &= c\left(\sum_{l,w} \frac{c_{lw}}{c} (\log \frac{c_{lw}}{c} + \log c) \right) \\
&= c \log c + c\left(\sum_{l,w} \pi_{lw} \log \pi_{lw} \right).
\end{aligned}$$

By (18), the values π_{lw} satisfy

$$\sum_{l,w} \pi_{lw} = 1, \qquad \sum_{l,w} l \pi_{lw} = n/c. \tag{19}$$

Using Lagrange multipliers to maximize $-\sum_{l,w} \pi_{lw} \log \pi_{lw}$ under the constraint (19), or reasoning as in the proof of [5, Theorem 12.10.1] we get

$$-\sum_{l,w} \pi_{lw} \log \pi_{lw} \leq k \log |\mathcal{A}| + \log \left(1 + \frac{n}{c} \right) + \frac{n}{c} \log \left(1 + \frac{c}{n} \right).$$

Combining the above inequalities with Lemma A.3 we get

$$c\log c \le |s|H_k(s) + c\log M + c\left(k\log|\mathcal{A}| + \log\left(1+\frac{n}{c}\right) + \frac{n}{c}\log\left(1+\frac{c}{n}\right)\right).$$

The thesis follows observing that $\log(1+c/n)^{(n/c)} \le \log e$ and $\log(1+n/c) \le \log(n/c) + (c/n)\log e$. ◻

Proof of Lemma 2.3. Let $s = x_1 \cdots x_n$ with $n > k$. Let $y_1, \dots, y_t$ denote a parsing of s in which each word appears at most M times. This parsing induces a parsing of $x_{k+1}\cdots x_n$ in which each word appears at most $M+1$ times. The thesis follows by Theorem A.4. ◻

Appendix B. $\texttt{LZ78}_{RL}$-induced parsing.

LEMMA B.1. *Let $\tilde{s}$ be derived from s as described in section 3, and let $w_1, w_2, \dots, w_t$ denote the* LZ78 *parsing of $\tilde{s}$. Then, it is possible to build a parsing $w'_1, w'_2, \dots, w'_t$ of s such that each word appears at most $2\log|s|$ times.*

Proof. The parsing of $w'_1, w'_2, \dots, w'_t$ is defined as follows. For $i = 1, \dots, |\tilde{s}|$, we associate to the ith character of $\tilde{s}$ a nonempty string ω_i with the property that

$$s = \omega_1\omega_2\cdots\omega_{|\tilde{s}|}. \tag{20}$$

Then, from each word w_j in the parsing of $\tilde{s}$ we get the word w'_j by simply concatenating the strings ω_i's corresponding to the characters of w_j.

The strings ω_i's are defined by partially reversing the binary encoding utilized for the construction of the string $\tilde{s}$. More precisely, let $b = b_l b_{l-1}\cdots b_1 b_0$, $b_i \in \{\alpha, \mathtt{0}\}$ denote the encoding of the α-segment α^m (that is, with the notation of section 3 $b_l \cdots b_0 = B(\alpha^m)$). We associate to each character b_j the string $\omega(b_j)$ defined by

$$\omega(b_j) = \begin{cases} \alpha & \text{if } j = l; \\ \alpha^{2^j} & \text{if } j < l \text{ and } b_j = \mathtt{0}; \\ \alpha^{2^{j+1}} & \text{if } j < l \text{ and } b_j = \alpha. \end{cases} \tag{21}$$

Note that, for $j < l$, $\omega(b_j)$ contains 2^j more α's than if we simply had reversed the binary encoding. This is done at the expense of b_l (which translates to a single α) with the purpose of ensuring that every $\omega(b_j)$ is nonempty. One can easily verify that $\omega(b_l)\omega(b_{l-1})\cdots\omega(b_0) = \alpha^m$, which proves that the strings ω_i's satisfy (20).

We now show that in the parsing $w'_1, w'_2, \dots, w'_t$ each word appears at most $2\log|s|$ times. Since the words w_j's are distinct, we need to show that when we replace the single characters with the strings ω_i's, at most $2\log|s|$ distinct words translate into the same word. To prove this we introduce the following notation. For each substring σ of $\tilde{s}$ we denote by $\omega(\sigma)$ the string obtained by replacing each character of σ with the corresponding ω_i's. For $\alpha \in \mathcal{A}$, we denote with $B_{ij}(\alpha^m)$ the string obtained by removing from $B(\alpha^m)$ the first i and the last j characters. For example,

$$B_{10}(\alpha^5) = \mathtt{0}\alpha, \qquad B_{01}(\alpha^4) = \alpha\mathtt{0}, \qquad B_{00}(\alpha^{17}) = \alpha\mathtt{000}\alpha.$$

Finally, given any string $\sigma \in \{\alpha, \mathtt{0}\}^*$ we denote with $\mathtt{Bin}(\sigma)$ the integer we get by replacing α with 1 and interpreting the result as a binary number. For example,

$$\mathtt{Bin}(\alpha) = \mathtt{Bin}(\mathtt{00}\alpha) = 1, \qquad \mathtt{Bin}(\alpha\mathtt{00}) = 4.$$

By construction we know that $\tilde{s}$ consists in a concatenation of "compressed" α-segments. Hence, a generic word w_i in the parsing of $\tilde{s}$ has the form[1]

$$w_i = B_{r0}(\alpha_{i_1}^{n_1})B(\alpha_{i_2}^{n_2})\cdots B(\alpha_{i_{k-1}}^{n_{k-1}})B_{0s}(\alpha_{i_k}^{n_k}).$$

We have

$$\begin{aligned}\omega(w_i) &= \omega(B_{r0}(\alpha_{i_1}^{n_1}))\,\omega(B(\alpha_{i_2}^{n_2}))\,\cdots\,\omega(B(\alpha_{i_{k-1}}^{n_{k-1}}))\,\omega(B_{0s}(\alpha_{i_k}^{n_k}))\\ &= \omega(B_{r0}(\alpha_{i_1}^{n_1}))\,\alpha_{i_2}^{n_2}\,\cdots\,\alpha_{i_{k-1}}^{n_{k-1}}\,\omega(B_{0s}(\alpha_{i_k}^{n_k})).\end{aligned} \tag{22}$$

Thus, in order to count how many distinct w_i's translate into the same word, we need to study when two distinct strings σ_1, σ_2 of the form

$$\sigma_1 = B_{i0}(\alpha^{n_1}) \qquad \sigma_2 = B_{j0}(\alpha^{n_2})$$

or

$$\sigma_1 = B_{0i}(\alpha^{m_1}) \qquad \sigma_2 = B_{0j}(\alpha^{m_2}),$$

are such that $\omega(\sigma_1) = \omega(\sigma_2)$.

Consider first the case $\sigma_1 = B_{i0}(\alpha^{n_1}), \sigma_2 = B_{j0}(\alpha^{n_2})$. By (21) we know that the number of α's in $\omega(\sigma_1)$ is given by

$$|\omega(\sigma_1)| = |\omega(B_{i0}(\alpha^{n_1}))| = \begin{cases}\mathtt{Bin}(\sigma_1), & \text{if } i = 0;\\ \mathtt{Bin}(\sigma_1) + 2^{|\sigma_1|} - 1, & \text{if } i > 0;\end{cases}$$

and a similar result holds for $\omega(\sigma_2)$. We now show that $\omega(\sigma_1) = \omega(\sigma_2)$ with $\sigma_1 \neq \sigma_2$ only if $i = 0$ and $j \neq 0$ or, vice versa, $i \neq 0$ and $j = 0$. If $i = j = 0$, then

$$|\omega(\sigma_1)| = |\omega(\sigma_2)| \quad\Rightarrow\quad \mathtt{Bin}(\sigma_1) = \mathtt{Bin}(\sigma_2) \quad\Rightarrow\quad \sigma_1 = \sigma_2.$$

In fact, we cannot have, say, $\sigma_1 = \mathtt{000}\sigma_2$, since $\sigma_1 = B_{00}(\alpha^{n_1})$ and its leading character is different from `0` by construction. Assume now $i, j \neq 0$. If $|\sigma_1| = |\sigma_2|$, then

$$|\omega(\sigma_1)| = |\omega(\sigma_2)| \quad\Rightarrow\quad \mathtt{Bin}(\sigma_1) = \mathtt{Bin}(\sigma_2),$$

which again implies $\sigma_1 = \sigma_2$ since the two strings have the same length. Finally, if $i, j \neq 0$ and $|\sigma_1| \neq |\sigma_2|$, for example, $|\sigma_1| > |\sigma_2|$, we have

$$\begin{aligned}|\omega(\sigma_1)| &= \mathtt{Bin}(\sigma_1) + 2^{|\sigma_1|} - 1\\ &\geq 2^{|\sigma_2|} + 2^{|\sigma_2|} - 1\\ &> \mathtt{Bin}(\sigma_2) + 2^{|\sigma_2|} - 1\\ &= |\omega(\sigma_2)|,\end{aligned}$$

which implies $\omega(\sigma_1) \neq \omega(\sigma_2)$. In summary, we can conclude that there exist at most two distinct strings $\sigma_1 = B_{i0}(\alpha^{n_1}), \sigma_2 = B_{j0}(\alpha^{n_2})$ such that $\omega(\sigma_1) = \omega(\sigma_2)$.

Consider now the case $\sigma_1 = B_{0i}(\alpha^{m_1}), \sigma_2 = B_{0j}(\alpha^{m_2})$. By (21) we get that the number of α's in $\omega(\sigma_1)$ is given by

$$|\omega(\sigma_1)| = |\omega(B_{0i}(\alpha^{m_1}))| = \mathtt{Bin}(\sigma_1)2^i - 2^i - 1 = 1 + (\mathtt{Bin}(\sigma_1) - 1)2^i. \tag{23}$$

[1]We are assuming that w_i contains at least two distinct characters of $\mathcal{A}$. Therefore, we do not consider the case $w_i = B_{jk}(\alpha^n)$ which, however, can be handled with a similar analysis.

We observe that we can have $\omega(\sigma_1) = \omega(\sigma_2)$ with $\sigma_1 \neq \sigma_2$ only if $i \neq j$. In fact, if $i = j$, by (23) we have

$$|\omega(\sigma_1)| = |\omega(\sigma_2)| \quad \Rightarrow \quad \mathtt{Bin}(\sigma_1) = \mathtt{Bin}(\sigma_2),$$

which implies $\sigma_1 = \sigma_2$, since the leading character of both strings is different from 0. Note that, by construction, the length of every "compressed" α-segment $B(\alpha^n)$ is at most $\log|s|$. Hence, the above observation implies that there can be at most $\log|s|$ distinct strings $\sigma_1, \dots, \sigma_k$, $\sigma_i = B_{0j_i}(\alpha^{n_i})$, such that $\omega(\sigma_1) = \omega(\sigma_2) = \cdots = \omega(\sigma_k)$. In fact, the indices j_i's must be distinct and none of them can be greater than $\log|s|$.

We can now conclude our analysis of the induced parsing $w'_1, \dots, w'_t$. Since there are only two possible distinct prefixes and $\log|s|$ possible distinct suffixes which translate to the same substring, by (22) we have that at most $2\log|s|$ (distinct) words, w_i's, can translate to the same w'_j.

This completes the proof. $\square$

REFERENCES

[1] T. Bell, I. Witten, and J. Cleary, *Modelling for text compression*, ACM Computing Surveys, 21 (1989), pp. 557–592.

[2] P. Bender and J. Wolf, *New asymptotic bounds and improvements on the Lempel-Ziv data compression algorithm*, IEEE Trans. Inform. Theory, 37 (1991), pp. 721–729.

[3] J. Bentley, D. Sleator, R. Tarjan, and V. Wei, *A locally adaptive data compression scheme*, Comm. ACM, 29 (1986), pp. 320–330.

[4] C. Bloom, *LZP: A new data compression algorithm*, in Proceedings of the IEEE Data Compression Conference, Snowbird, UT, 1996.

[5] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, 1991.

[6] G. Hansel, D. Perrin, and I. Simon, *Compression and entropy*, in Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 577, Springer, Berlin, 1992, pp. 515–528.

[7] P. Jacquet and W. Szpankowski, *Asymptotic behaviour of the Lempel-Ziv parsing scheme in digital search trees*, Theoret. Comput. Sci., 144 (1995), pp. 161–197.

[8] T. Kawabata, *Exact analysis of the Lempel-Ziv algorithm for I. I. D. sources*, IEEE Trans. Inform. Theory, 39 (1993), pp. 698–708.

[9] G. Louchard and W. Szpankowski, *On the average redundancy rate of the Lempel-Ziv code*, IEEE Trans. Inform. Theory, 43 (1997), pp. 2–8.

[10] H. Morita and K. Kobayashi, *On asymptotic optimality of a sliding window variation of Lempel-Ziv codes*, IEEE Trans. Inform. Theory, 39 (1993), pp. 1840–1846.

[11] E. Plotnik, M. Weinberger, and J. Ziv, *Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm*, IEEE Trans. Inform. Theory, 38 (1992), pp. 66–72.

[12] M. Rodeh, V. Pratt, and S. Even, *Linear algorithm for data compression via string matching*, J. ACM, 28 (1981), pp. 16–24.

[13] D. Salomon, *Data Compression: The Complete Reference*, Springer-Verlag, New York, 1997.

[14] S. Savari, *Redundancy of the Lempel-Ziv incremental parsing rule*, IEEE Trans. Inform. Theory, 43 (1997), pp. 9–21.

[15] S. Savari, *Redundancy of the Lempel-Ziv string matching code*, IEEE Trans. Inform. Theory, 44 (1998), pp. 787–791.

[16] D. Sheinwald, *On the Ziv-Lempel proof and related topics*, Proc. IEEE, 82 (1994), pp. 866–871.

[17] W. Szpankowski, *Asymptotic properties of data compression and suffix trees*, IEEE Trans. Inform. Theory, 39 (1993), pp. 1647–1659.

[18] A. Wyner and A. Wyner, *Improved redundancy of a version of the Lempel-Ziv algorithm*, IEEE Trans. Inform. Theory, 41 (1995), pp. 723–731.

[19] A. Wyner and J. Ziv, *Fixed data base version of the Lempel-Ziv data compression algorithm*, IEEE Trans. Inform. Theory, 37 (1991), pp. 878–880.

[20] A. Wyner and J. Ziv, *The sliding-window Lempel-Ziv algorithm is asymptotically optimal*, Proc. IEEE, 82 (1994), pp. 872–877.

[21] A. J. Wyner, *The redundancy and distribution of the phrase lengths in the fixed-database Lempel-Ziv algorithm*, IEEE Trans. Inform. Theory, 43 (1997), pp. 1452–1464.

[22] E. Yang and J. Kieffer, *On the performance of data compression algorithms based on string matching*, IEEE Trans. Inform. Theory, 44 (1998), pp. 47–65.

[23] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23 (1977), pp. 337–343.

[24] J. Ziv and A. Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Trans. Inform. Theory, 24 (1978), pp. 530–536.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 912–953

VERTICAL DECOMPOSITION OF SHALLOW LEVELS IN 3-DIMENSIONAL ARRANGEMENTS AND ITS APPLICATIONS[*]

PANKAJ K. AGARWAL[†], ALON EFRAT[‡], AND MICHA SHARIR[§]

Abstract. Let $\mathcal{F}$ be a collection of n bivariate algebraic functions of constant maximum degree. We show that the combinatorial complexity of the vertical decomposition of the $(\leq k)$-level of the arrangement $\mathcal{A}(\mathcal{F})$ is $O(k^{3+\varepsilon}\psi(n/k))$ for any $\varepsilon > 0$, where $\psi(r)$ is the maximum complexity of the lower envelope of a subset of at most r functions of $\mathcal{F}$. This bound is nearly optimal in the worst case and implies the existence of shallow cuttings, in the sense of [J. Matoušek, *Comput. Geom.*, 2 (1992), pp. 169–186], of small size in arrangements of bivariate algebraic functions. We also present numerous applications of these results, including (i) data structures for several generalized 3-dimensional range-searching problems; (ii) dynamic data structures for planar nearest- and farthest-neighbor searching under various fairly general distance functions; (iii) an improved (near-quadratic) algorithm for minimum-weight bipartite Euclidean matching in the plane; and (iv) efficient algorithms for certain geometric optimization problems in static and dynamic settings.

Key words. arrangements, divide-and-conquer, range searching, nearest-neighbor searching, parametric searching, geometric optimization

AMS subject classifications. 57N80, 68Q20, 68Q40, 68R05, 68U05

PII. S0097539795295936

1. Introduction.

In this paper we extend the recent range-searching techniques of Matoušek [52] and of Agarwal and Matoušek [6] to arrangements of bivariate algebraic functions and derive many applications of the new techniques. As one motivation, consider the following *dynamic nearest-neighbor searching* problem: We dynamically maintain a set S of points in the plane under some metric δ. At any time, we wish to answer nearest-neighbor queries, in which we specify a point q and ask for the point of S nearest to q under the metric δ.

If δ is the Euclidean metric, a standard lifting transformation to $\mathbb{R}^3$ (as in [30]) reduces the problem to that of dynamically maintaining the lower envelope of planes in $\mathbb{R}^3$ so that, at any time during the maintenance, we can answer efficiently queries in which we specify a point $q \in \mathbb{R}^2$ and ask for the plane attaining the lower envelope at q. Using the parametric-searching technique of [4], this problem can be solved by a dynamic range-searching mechanism in which we wish to determine whether a query point in $\mathbb{R}^3$ lies below all the planes in the current set. The techniques in [6, 52] are based on the notion of *shallow cuttings*: Given an arrangement $\mathcal{A}$ of n planes in $\mathbb{R}^3$ and parameters $k, r \leq n$, there exists a $(1/r)$-*cutting* of the first k

---

[*]Received by the editors December 8, 1995; accepted for publication (in revised form) November 17, 1997; published electronically December 15, 1999. This research was supported by a grant from the U.S.-Israeli Binational Science Foundation.

http://www.siam.org/journals/sicomp/29-3/29593.html

[†]Department of Computer Science, Box 1029, Duke University, Durham, NC 27708-0129 (pankaj@cs.duke.edu). This author was supported by National Science Foundation (NSF) grant CCR-93-01259, by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, and by an NYI award and matching funds from Xerox Corporation.

[‡]School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel.

[§]School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012. This author was supported by NSF grants CCR-94-24398 and CCR-93-11127 and by a grant from the German-Israeli Foundation for Scientific Research and Development.

levels of $\mathcal{A}$ of size $O(r(1 + kr/n)^2)$, i.e., there exists a decomposition of the (union of the) cells constituting these levels into $O(r(1 + kr/n)^2)$ simplices, so that the interior of each simplex is intersected by at most n/r planes. Using the existence of such shallow cuttings, Matoušek [52] constructs an efficient static data structure for halfspace-emptiness queries in $\mathbb{R}^3$; it requires $O(n \log \log n)$ storage and $O(n \log n)$ preprocessing time and answers a halfspace-emptiness query in $O(\log^2 n)$ time (see also [11]). This structure was later dynamized in [6]. These techniques also apply to arrangements of hyperplanes in higher dimensions.

Although these techniques can be extended to arrangements of curves in the plane, they fail for arrangements of general surfaces in three and higher dimensions. Such an arrangement does arise, for example, in the nearest-neighbor searching problem if the underlying distance function δ is not Euclidean. A specific case of this kind, which we will face in one of our applications (minimum-weight bipartite Euclidean matching in the plane), is where each point $s \in S$ has an additive weight $w(s)$ and the distance from a query point q to s is defined as $|qs| + w(s)$, where $|qs|$ is the Euclidean distance between these points. To tackle such cases efficiently, we need to extend the results of [6, 52] to the case of arrangements of more general functions, and this is one of the main goals of this paper.

The main technical result that we establish is the existence of shallow cuttings of small size in arrangements of (the graphs of) low-degree algebraic bivariate functions. To obtain such cuttings, we use the technique of *vertical decomposition* of cells in arrangements. This technique, described in detail in [18], is the only known general-purpose technique for decomposing cells in such arrangements into a small number of subcells of *constant description complexity* (that is, semialgebraic subcells, each defined by a constant number of polynomial equalities and inequalities, each of constant maximum degree). It is well known that the complexity of the vertical decomposition of a cell (2-face) C in a planar arrangement of low-degree algebraic curves is proportional to the number of vertices of C. However, this property does not hold in higher dimensions (even for an arrangement of planes in $\mathbb{R}^3$): One can construct a cell in an arrangement of n planes in $\mathbb{R}^3$ that has $O(n)$ vertices and whose vertical decomposition consists of $\Omega(n^2)$ subcells. In three dimensions, the complexity (number of subcells) of the vertical decomposition of the *entire* arrangement of n low-degree algebraic bivariate functions is shown in [18] to be close to $O(n^3)$ and is thus almost optimal (in higher dimensions, though, the bounds are considerably weaker; see [18, 24, 28, 37] for some partial results). No similarly sharp bounds were known for the vertical decomposition of the cells that lie only in the first k levels of the arrangement (even for arrangements of planes). Using a standard probabilistic analysis technique by Clarkson and Shor ([25]; see also [66]), one can easily show that the combinatorial complexity of these (undecomposed) cells is $O(k^3\psi(n/k))$, where $\psi(r)$ is the maximum combinatorial complexity of the lower envelope of any subset of at most r of the given surfaces. For low-degree algebraic bivariate functions, the results of [38, 65] imply that $\psi(n) = O(n^{2+\varepsilon})$, for any $\varepsilon > 0$,[1] but in certain favorable cases, such as the case of planes, this complexity is smaller.

The first main result of the paper, derived in section 2, is that the combinatorial complexity of the vertical decomposition of the cells in the first k levels of the

[1]Throughout this paper, ε denotes an arbitrarily small positive constant and $g(n) = O(f(n) \cdot n^{\varepsilon})$ means that for any given $\varepsilon > 0$, we can choose a constant c_{ε} so that $g(n) \le c_{\varepsilon} f(n) \cdot n^{\varepsilon}$. A complexity bound of the form $O(f(n) \cdot n^{\varepsilon})$ for an algorithm means that for any prespecified $\varepsilon > 0$, we can tune the algorithm so that its complexity is bounded by $c_{\varepsilon} f(n) \cdot n^{\varepsilon}$ for an appropriate constant c_{ε}.

arrangement is $O(k^{3+\varepsilon}\psi(n/k))$. This bound is close to optimal in the worst case. We then apply this bound to obtain a sharp bound on the size of shallow cuttings in arrangements of bivariate functions. Specifically, we show, in Theorem 3.1, that there exists a $(1/r)$-cutting of the first k levels in an arrangement of n low-degree, algebraic, bivariate functions in $\mathbb{R}^3$, as above, whose size is $O(q^{3+\varepsilon}\psi(r/q))$, where $q = k(r/n)+1$. This bound is almost tight in the worst case and almost coincides with the bound given in [52] for the case of planes. The proof adapts the analysis technique of [52]. If $r = O(1)$, a $(1/r)$-cutting of this size can be computed in $O(n)$ time.

An immediate consequence of Theorem 3.1 is an efficient algorithm for the following problem: Preprocess a set $\mathcal{F}$ of n bivariate functions into a data structure so that all functions whose graphs lie below a query point in $\mathbb{R}^3$ can be reported efficiently. We present a data structure of size $O(\psi(n)n^{\varepsilon})$, which can be constructed in $O(\psi(n)n^{\varepsilon})$ time, so that a query can be answered in $O(\log n + \xi)$ time, where ξ is the output size. If we are interested only in determining whether w lies below the graphs of all functions of $\mathcal{F}$, the query time is $O(\log n)$. (Here we are assuming a model of computation in which various operations involving a constant number of fixed-degree algebraic functions can be performed in $O(1)$ time; see below for details.) We can also modify this structure to obtain an efficient data structure for maintaining a set of bivariate algebraic functions dynamically, as above, so that we can efficiently determine whether a query point lies below all the function graphs in the current set. The modified data structure also requires $O(\psi(n)n^{\varepsilon})$ storage, the cost of an update is $O(\psi(n)/n^{1-\varepsilon})$, and a query can be answered in $O(\log n)$ time.[2] The technique for constructing these data structures adapts ideas from [6, 52]. Data structures with similar bounds are given in [6, 11, 21, 52] for a set of linear functions.

We also obtain an efficient algorithm for constructing and searching in the first k levels of an arrangement of n bivariate functions, as above. These levels can be constructed in $O(k^{3+\varepsilon}n^{\varepsilon}\psi(n/k))$ time, in an appropriate model of computation, and can be stored into a data structure of similar size so that we can determine, in $O(\log n)$ time, whether the level of a query point $p \in \mathbb{R}^3$ with respect to the arrangement is at most k and, if so, return the level of p.

We next apply the new mechanisms to a variety of geometric problems. First, in section 6, we derive an efficient technique for dynamically maintaining a set S of points (or more general objects) in the plane so that we can efficiently compute the nearest neighbor (or the farthest neighbor) of a query point in the current set of points, under any "reasonable" distance function δ (defined more precisely later), which can be fairly arbitrary and does not have to satisfy any metric-like properties. Assume that the complexity of the lower (or upper) envelope of the functions $f_i(\mathbf{x}) = \delta(p_i, \mathbf{x})$, over all objects p_i in the current set S, is at most $g(|S|)$. (This is the same as the complexity of the nearest (or farthest) neighbor Voronoi diagram of S if δ is a metric or a convex distance function.) Then our solution requires $O(g(n)n^{\varepsilon})$ storage, $O(g(n)/n^{1-\varepsilon})$ time for each update, and $O(\log n)$ time for each nearest- (or farthest-)neighbor query. We give applications of this technique to dynamic maintenance of a bichromatic closest pair between two planar point sets, under any "reasonable" metric, and of a minimum spanning tree of a planar point set under any L_p metric.

Another application of our technique is an improved algorithm for computing a

[2]Throughout this paper, the update-time bounds are amortized. We believe that the same bounds can be achieved in the worst case, using the known (albeit complicated) techniques of [60]. Nevertheless, for the sake of simplicity we will stick to amortized bounds, which will not affect the applications that we study here.

minimum-weight bipartite Euclidean matching for point sets in the plane (see section 7). That is, we are given a set of n blue points and a set of n red points in the plane, and we wish to find a matching between the blue points and the red points that minimizes the sum of the distances between the pairs of matched points. Our solution is based on the algorithm of Vaidya [70], which requires a data structure for answering nearest-neighbor queries in a dynamic setting, where the distance to each of the maintained sites is the Euclidean distance plus some additive weight associated with the site. Using our dynamic nearest-neighbor searching technique, we can improve the running time of Vaidya's algorithm from $O(n^{2.5} \log n)$ to $O(n^{2+\varepsilon})$. We also obtain an $O(n^{7/3+\varepsilon})$ algorithm for an arbitrary "reasonable" metric. Recently, Varadarajan [72] has obtained an improved algorithm for the nonbipartite case, for which the above nearest-neighbor searching mechanism is not sufficient.

Another application of our results is to dynamic maintenance of the intersection of congruent balls in $\mathbb{R}^3$, in the strong sense that we wish to determine, after each update, whether the current intersection is empty. For this, we combine our technique with the variant of parametric searching recently proposed in [69]. We obtain a data structure of size $O(n^{1+\varepsilon})$, which can be updated in $O(n^{\varepsilon})$ time per insertion/deletion and which supports "intersection-emptiness" queries in $O(\log^4 n)$ time.

We next apply our technique in section 9 to another problem in geometric optimization, namely, the *dynamic smallest stabbing-disk* problem: We maintain dynamically a set $\mathcal{C}$ of (possibly intersecting) simply shaped, compact, convex sets in the plane, and we wish to compute, after each insertion or deletion of such a set, the smallest disk, or the smallest homothetic copy of any compact convex set of simple shape, that intersects all the sets in the current $\mathcal{C}$. The case in which we have points instead of general convex sets was recently studied in [6]. Our solution requires $O(n^{1+\varepsilon})$ storage and preprocessing and recomputes the smallest stabbing disk after each update in $O(n^{\varepsilon})$ time. A byproduct of our analysis, which we believe to be of independent interest, is a near-linear bound on the complexity of the farthest-neighbor Voronoi diagram of (possibly intersecting) simply shaped, compact, convex sets in the plane, under any simply shaped, convex distance function. The bound is linear for (possibly intersecting) line segments under the Euclidean distance.

We finally present, in section 10, a few more applications, where most of the details are omitted. The paper concludes in section 11 with a discussion of our results and with some open problems. The collection of applications described in this paper is by no means exhaustive, and the new techniques obtained in this paper have many additional applications. The main message of this paper, in our opinion, is that there is a real need to adapt and extend range-searching and related techniques that were originally developed for arrangements of planes or hyperplanes to arrangements of algebraic surfaces. This adaptation is by no means easy, and the present study achieves it only for 3-dimensional arrangements, but it significantly enlarges the scope of range-searching applications.

2. Vertical decomposition of levels in 3-dimensional arrangements.

Let $\mathcal{F}$ be a collection of n bivariate functions satisfying the following conditions:[3]

(F1) Each $f \in \mathcal{F}$ is a continuous, totally defined, algebraic function of constant maximum degree b.

(F2) The functions in $\mathcal{F}$ are in *general position.* This excludes degenerate con-

[3]Abusing the notation slightly, we will not distinguish between a function and its graph. We will use $\mathcal{F}$ to denote a collection of functions as well as the family of surfaces representing their graphs.

figurations where four function graphs meet at a point, a pair of graphs are tangent to each other, etc.

With some modifications of the analysis, we can also handle the case in which the functions in $\mathcal{F}$ are only partially defined and the boundary of the domain of each function is defined by a constant number of polynomial equalities and inequalities of constant maximum degree, say, b, too. The simplest way of doing this is to extend each $f \in \mathcal{F}$ to a totally defined function by forming the Minkowski sum of the graph of f with a sufficiently narrow vertical cone, whose equation is $z = c\sqrt{x^2 + y^2}$, for a sufficiently large c and by taking the lower boundary of this sum as the graph of the extended function. The resulting extended functions are easily seen to be totally defined continuous functions, each of whose graphs is a semialgebraic set of constant description complexity. One can then check that the analysis given below also applies to collections of such functions. Concerning the general position assumption, we refer the reader to the papers [38, 65] for more details about the definition and properties of this concept. Following arguments given in these papers, it will follow that no real loss of generality is made by assuming general position, in the sense that the maximum combinatorial complexity of the structures we study here is attained when functions are in general position.

The *arrangement* of $\mathcal{F}$, denoted as $\mathcal{A}(\mathcal{F})$, is the subdivision of $\mathbb{R}^3$ induced by the graphs of the functions in $\mathcal{F}$ (see [66] for more details). We will refer to the 3-dimensional cells of $\mathcal{A}(\mathcal{F})$ simply as the *cells* of $\mathcal{A}(\mathcal{F})$. The complexity of a cell C, denoted as $|C|$, is the number of faces of all dimensions on the boundary of C. The *level* in $\mathcal{A}(\mathcal{F})$ of a point $p = (x_p, y_p, z_p) \in \mathbb{R}^3$, denoted as $\mu(p) = \mu_{\mathcal{F}}(p)$, is defined as the number of functions $f \in \mathcal{F}$ for which $z_p > f(x_p, y_p)$. The level of all points lying on a face ϕ (of any dimension) of $\mathcal{A}(\mathcal{F})$ is the same, which we denote by $\mu(\phi)$. The *k-level* of $\mathcal{A}(\mathcal{F})$, for any $0 \le k \le n-1$, is the closure of the union of all 2-dimensional faces of $\mathcal{A}(\mathcal{F})$ whose level is k; the 0-level (resp., $(n-1)$-level) is the graph of the lower (resp., upper) envelope of $\mathcal{F}$. The *$(\le k)$-level* of $\mathcal{A}(\mathcal{F})$, denoted as $\mathcal{A}_{\le k}(\mathcal{F})$, is the collection of all cells of $\mathcal{A}(\mathcal{F})$ whose level is at most k. Let

$$\psi(\mathcal{F}, k) = \sum_{C \in \mathcal{A}_{\le k}(\mathcal{F})} |C|$$

denote the combinatorial complexity of $\mathcal{A}_{\le k}(\mathcal{F})$. Let $\mathbf{F}$ denote a (possibly infinite) family of bivariate functions that satisfies (F1), and let

$$\psi(n, k) = \psi^{\mathbf{F}}(n, k)$$

be any upper bound on the quantity $\max_{\mathcal{F}} \ \psi(\mathcal{F}, k)$, where the maximum is taken over all collections $\mathcal{F} \subseteq \mathbf{F}$ of at most n functions (that satisfy (F1) and (F2)). To simplify the presentation, we will also use $\psi(n)$ to denote $\psi(n, 0)$ and will allow n in this notation to be any real nonnegative number.

A straightforward application of the probabilistic analysis technique of Clarkson and Shor [25] (see also [66]) implies that

$$\psi(n, k) = O(k^3 \psi(n/k)). \tag{2.1}$$

We also note that, by the results of [38, 65], we always have $\psi(n) = O(n^{2+\varepsilon})$, where the constant of proportionality depends on ε and on the maximum degree of the given surfaces. Hence, in the worst case we have $\psi(n, k) = O(k^{1-\varepsilon} n^{2+\varepsilon})$. However, these bounds can be much smaller for certain favorable underlying families $\mathbf{F}$. A typical

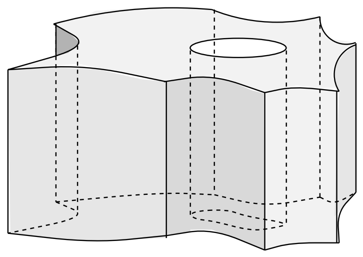


FIG. 2.1. *Vertical decomposition: A subcell created in step* I.

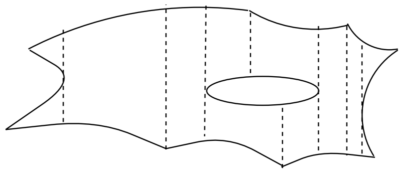


FIG. 2.2. *Vertical decomposition: step* II.

example is the case of Voronoi diagrams, where each function $f \in \mathbf{F}$ has the form $f(x, y) = \rho((x, y), q)$. Here ρ is a fixed metric and q is an arbitrary point in the plane. In this case $\psi(n) = O(n)$, so $\psi(n, k) = O(nk^2)$.

Because of technical reasons, we make the following assumption on $\mathbf{F}$.

(F3) The upper bound $\psi(n)$ on the complexity of the lower envelope (and also of the upper envelope) of any collection $\mathcal{F} \subseteq \mathbf{F}$ of n functions is of the form $\psi(n) = n^\alpha \beta(n)$, where $\alpha \le 2$ is a constant and $\beta(n)$ is a positive function so that for any $\delta > 0$, the sequence $\beta(n)/n^\delta$ is eventually nonincreasing and $\lim_{n\to\infty} \beta(n)/n^\delta = 0$.

This assumption involves no real loss of generality. In fact, all known bounds for $\psi(n)$ have this form with either $\alpha = 1$ or $\alpha = 2$.

The *vertical decomposition* of a (3-dimensional) cell $C \in \mathcal{A}(\mathcal{F})$, denoted as $C^\star$, is defined in the following standard manner (see [18, 66] for more details):

I. For each edge e of ∂C, we erect a z-vertical wall from e, which is the union of all maximal z-vertical segments passing through points of e and lying within (the closure of) C. The collection of these walls partitions C into subcells, each of which has a unique top facet (2-dimensional face) and a unique bottom facet, each contained in some facet of C. Every z-vertical line cuts such a subcell in a (possibly empty) interval; see Figure 2.1.

II. We take each of the cells Δ generated in the first step, project it onto the xy-plane, and construct the 2-dimensional vertical decomposition of the projection Δ_2 of Δ by erecting, from each vertex of Δ_2 and from each locally x-extreme point of $\partial\Delta_2$, a maximal y-vertical segment contained in the closure of Δ_2. These segments partition Δ_2 into trapezoidal-like subcells; each subcell $\tau_2 \subseteq \Delta_2$ induces a subcell τ of Δ, obtained by intersecting Δ with the vertical cylinder $\tau_2 \times \mathbb{R}$ (see Figure 2.2).

The cells obtained in this two-step decomposition form the vertical decomposition of C. We define $\mathcal{A}^\star_{\le k}(\mathcal{F})$ (resp., $\mathcal{A}^\star(\mathcal{F})$), the vertical decomposition of $\mathcal{A}_{\le k}(\mathcal{F})$ (resp., of $\mathcal{A}(\mathcal{F})$), as the union of the vertical decompositions of all cells in $\mathcal{A}_{\le k}(\mathcal{F})$ (resp., in $\mathcal{A}(\mathcal{F})$). The complexity of such a vertical decomposition is the total number of its cells. Abusing the notation slightly, we will use $\mathcal{A}^\star(\mathcal{F})$ to denote a spatial subdivision as well as a set of cells. Notice that for each cell $\tau \in \mathcal{A}^\star(\mathcal{F})$, there is a subset $D = D(\tau)$ of at most six functions of $\mathcal{F}$ so that $\tau \in \mathcal{A}^\star(D)$. Since all the functions in $\mathcal{F}$ have constant maximum degree, it follows that each cell of $\mathcal{A}^\star(\mathcal{F})$ has *constant description complexity* (in the sense defined above) [18], a property that is crucial for the construction of cuttings, which will be studied in the next section. We refer to the set $D(\tau)$ as the set of functions *defining* the cell τ. The main result of this section is the following theorem.

THEOREM 2.1. *Let $\mathcal{F}$ be a collection of n bivariate functions satisfying conditions* (F1)–(F3). *For any integer $k \le n$ and any $\varepsilon > 0$, the combinatorial complexity of $\mathcal{A}^\star_{\le k}(\mathcal{F})$ is $O(k^{3+\varepsilon}\psi(n/k))$, where the constant of proportionality depends on ε and on the maximum degree of the functions in $\mathcal{F}$.*

Proof. Let C be a cell of $\mathcal{A}_{\le k}(\mathcal{F})$, and let Δ be a subcell of C created in step I of the vertical decomposition. If $|\Delta|$ is the number of vertical edges in Δ plus the number of locally x-extremal points of edges of $\mathcal{A}_{\le k}(\mathcal{F})$ on $\partial\Delta$, then step II partitions Δ into $O(|\Delta| + 1)$ subcells. Indeed, each vertical edge of Δ (resp., each locally x-extremal point as above) corresponds to a vertex of Δ_2 (resp., to a locally x-extremal point on an edge of $\partial\Delta_2$), and the number of subcells in the 2-dimensional vertical decomposition of Δ_2 is easily seen to be proportional to 1 plus the number of vertices of Δ_2 plus the number of locally x-extremal points on the edges of $\partial\Delta_2$. Summing over all cells of $\mathcal{A}_{\le k}(\mathcal{F})$, the number of subcells in $\mathcal{A}^\star_{\le k}(\mathcal{F})$ is proportional to the sum of the number of vertical edges in the subcells created in step I, the number of edges of $\mathcal{A}_{\le k}(\mathcal{F})$, and the number of subcells created in step I that do not contain any vertical edge or any locally x-extremal point on an edge of $\mathcal{A}_{\le k}(\mathcal{F})$. The third quantity is bounded by $O(k^3\psi(n/k))$ because each such subcell contains at least one edge or face of $\mathcal{A}_{\le k}(\mathcal{F})$ and each such feature of $\mathcal{A}_{\le k}(\mathcal{F})$ can be incident to $O(1)$ subcells.

Hence, it suffices to bound the number of vertical edges in the subcells created in step I. There are two types of vertical edges of Δ:

(V1) a z-vertical segment erected from a vertex of C, or

(V2) the intersection segment of two z-vertical walls erected from two edges e, e' of C, where e lies on the top portion of ∂C and e' lies on the bottom portion of ∂C.

Since the number of vertical edges of type (V1) is bounded by the number of vertices in C, the total number of such edges, over all cells of $\mathcal{A}_{\le k}(\mathcal{F})$, is at most $O(k^3\psi(n/k))$. Hence, it suffices to bound the number of vertical edges of type (V2).

If two edges e, e' of C generate a vertical edge of type (V2), then their xy-projections must intersect. Furthermore, every intersection point between the xy-projections of any two edges of ∂C, one on the upper portion and one on the bottom portion of ∂C, generates exactly one vertical edge of type (V2) (this follows from the fact that the cell C is xy-monotone). We obtain an upper bound on the number of these intersection points over all cells in $\mathcal{A}_{\le k}(\mathcal{F})$, which in turn yields an upper bound on the number of cells in $\mathcal{A}^\star_{\le k}(\mathcal{F})$.

For a fixed pair of edges $e, e' \in \mathcal{A}(\mathcal{F})$ such that $\mu(e') < \mu(e)$, we define (e, e', σ) to be an *edge-crossing* if σ is an intersection point of the xy-projections of the (relative interiors of the) edges e and e'; see Figure 2.3. Let ℓ_σ denote the z-vertical line passing through σ. We define the *crossing number* of (e, e', σ) to be $\mu(e) - \mu(e') - 2$, which, by our general position assumption, is equal to the number of functions whose graphs intersect ℓ_σ strictly between e and e'. For an integer $\rho \le \mu(e) - 2$, let $\mathcal{C}_\rho(e) = \mathcal{C}_\rho(e, \mathcal{F})$ denote the set of edge-crossings of the form (e, e', σ) whose crossing number is ρ; we

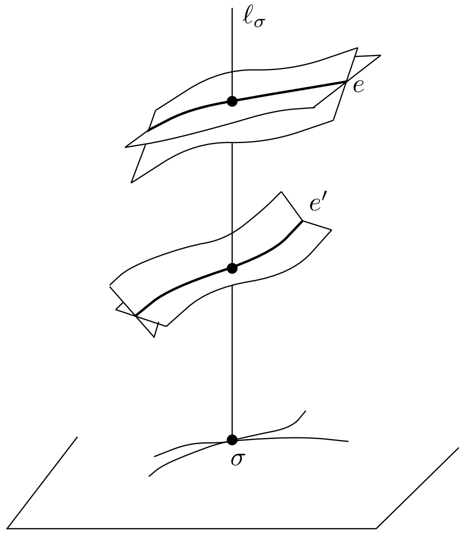


FIG. 2.3. *An edge-crossing* (e, e', σ).

put $\mathcal{C}_\rho(e, \mathcal{F}) = \emptyset$ for $\rho > \mu(e) - 2$. Set

$$\omega_\rho(e, \mathcal{F}) = |\mathcal{C}_\rho(e, \mathcal{F})| \quad \text{and} \quad \omega_{\leq\rho}(e, \mathcal{F}) = \sum_{i=0}^{\rho} \omega_i(e, \mathcal{F}).$$

Define

$$\varphi_\rho(\mathcal{F}, k) = \sum_{e \in \mathcal{A}_{\leq k}(\mathcal{F})} \omega_\rho(e, \mathcal{F})$$

to be the number of edge-crossings in $\mathcal{A}_{\leq k}(\mathcal{F})$ whose crossing number is ρ. Set

$$\varphi_\rho(n, k) = \max\ \varphi_\rho(\mathcal{F}, k),$$

where the maximum is taken over all subsets $\mathcal{F} \subseteq \mathbf{F}$ of size at most n. The corresponding quantities $\varphi_{\leq\rho}(\mathcal{F}, k)$ and $\varphi_{\leq\rho}(n, k)$ are defined in an obvious and analogous manner.

As follows from the above considerations, our goal is to show that

$$\varphi_0(n, k) \leq ck^{3+\varepsilon}\psi(n/k) \tag{2.2}$$

for some appropriate constant c (depending on ε and on the maximum degree of the functions in $\mathbf{F}$). This will be achieved by deriving a recurrence relationship for $\varphi_0(n, k)$, whose solution will yield the desired bound. The recurrence is obtained using the following counting argument, which borrows ideas from Agarwal, Schwarzkopf, and Sharir [8].

Let e be an edge of $\mathcal{A}_{\leq k}(\mathcal{F})$, and let C be the cell of $\mathcal{A}(\mathcal{F})$ lying immediately below e. Let V_e be the vertical 2-manifold obtained as the union of all z-vertical rays emanating from the points of e in the negative z-direction. The intersection of the graph of each function $f \in \mathcal{F}$ with V_e is an algebraic arc $f^{(e)}$ of constant maximum

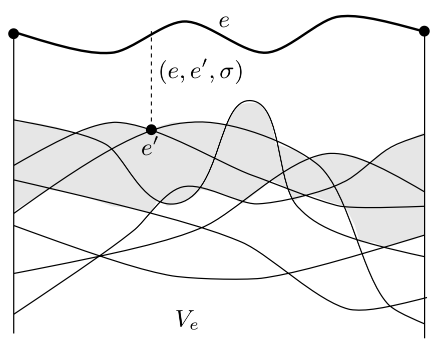


FIG. 2.4. *The arrangement $\mathcal{A}^{(e)}(\mathcal{F})$; the shaded region consists of points whose levels are between* 3 *and* 6.

degree, so each pair of these arcs intersect in at most a constant number of points, say, s, which depends only on the maximum degree b of the functions of $\mathcal{F}$. Let $\mathcal{A}^{(e)}(\mathcal{F})$ denote the cross section of $\mathcal{A}(\mathcal{F})$ with V_e. See Figure 2.4 for an illustration. The following lemma is a simple but crucial observation.

LEMMA 2.2. *Let e' be an edge of $\mathcal{A}_{\le k}(\mathcal{F})$ such that $\mu(e') < \mu(e)$. Then (e,e',σ) is an edge-crossing with crossing number ξ if and only if the intersection point $e' \cap \ell_\sigma$ is a vertex at level $\mu(e) - \xi - 2$ in $\mathcal{A}^{(e)}(\mathcal{F})$.*

The lemma implies that each edge-crossing of the form (e,e',σ) with zero crossing number corresponds to a vertex of the cross section $C^{(e)}$ of C with V_e (which lies on the lower portion of ∂C). Let $\mathcal{F}^{(e)} \subseteq \mathcal{F}$ be the set of functions that appear on the lower portion of $\partial C^{(e)}$, and let $t_e = |\mathcal{F}^{(e)}|$. For each function $f \in \mathcal{F}^{(e)}$, we consider the cross section $f^{(e)}$ of f within V_e, as defined above. It is easily seen that the lower portion of $\partial C^{(e)}$ is the upper envelope of the functions in $\tilde{\mathcal{F}}^{(e)} \equiv \{f^{(e)} \mid f \in \mathcal{F}^{(e)}\}$. By the standard Davenport–Schinzel theory [9, 39, 66],

$$\omega_0(e,\mathcal{F}) \le \lambda_s(t_e), \tag{2.3}$$

where s is as above and where $\lambda_s(t)$ is the (near-linear) maximum length of a (t,s)-Davenport–Schinzel sequence.

Since the endpoints of all arcs in $\tilde{\mathcal{F}}^{(e)}$ lie on the vertical boundary of V_e and $\mu(e) \le k$, it is easily seen that $t_e \le k$, and thus $\omega_0(e,\mathcal{F}) \le \lambda_s(k)$. Summing these bounds over all edges of $\mathcal{A}_{\le k}(\mathcal{F})$, we get

$$\varphi_0(\mathcal{F},k) = \sum_{e \in \mathcal{A}_{\le k}(\mathcal{F})} \omega_0(e,\mathcal{F}) \le \lambda_s(k)\psi(\mathcal{F},k).$$

This yields the following weaker bound on $\varphi_0(n,k)$:

$$\varphi_0(n,k) \le \lambda_s(k)\cdot\psi(n,k) \le c_1\lambda_s(k)k^3\psi(n/k) \tag{2.4}$$

for some constant c_1. The bound in (2.4) is asymptotically what we want for $k = O(1)$, and it is at most k times the desired bound for larger values of k.

We proceed now to prove the sharper bound that we are after. We fix (a sufficiently small) $\varepsilon > 0$ and choose some threshold constant k_0 (depending on ε) so that

$$4^\varepsilon k < c\lambda_s(k) < k^{1+\varepsilon^2/2} \tag{2.5}$$

for all $k > k_0$ and for some absolute constant c whose value will be determined later (the upper and lower bounds on $\lambda_s(k)$ in [9] imply that such a k_0 exists). If $k \le k_0$, then the bound asserted in Theorem 2.1 follows from (2.4) with an appropriate choice of the constant of proportionality. Henceforth, we assume that $k > k_0$. Put

$$q = \left\lceil (c\lambda_s(k)/k)^{1/\varepsilon} \right\rceil ;$$

$q \ge 4$, provided that k_0 is sufficiently large.

As above, fix an edge e of $\mathcal{A}_{\le k}(\mathcal{F})$, and continue to use the notations introduced above. If $t_e \le q$, then, by Lemma 2.2, $\omega_0(e, \mathcal{F})$, which is the same as the number of vertices of $C^{(e)}$, is at most $\lambda_s(q)$. Since the number of edges in $\mathcal{A}_{\le k}(\mathcal{F})$ is at most $\psi(n, k)$, the overall number of edge-crossings with crossing number 0 and involving such edges e is at most $\lambda_s(q)\psi(n, k)$.

Next, assume that $t_e > q$. Let f, f' be a pair of distinct functions in $\mathcal{F}^{(e)}$. By continuity, f and f' must intersect within V_e at least once. Thus each function $f \in \mathcal{F}^{(e)}$ must cross at least $t_e - 1$ other functions of $\mathcal{F}$ within V_e; that is, each function $f \in \mathcal{F}^{(e)}$ is incident to at least $t_e - 1$ vertices of $\mathcal{A}^{(e)}(\mathcal{F})$. Since the graph of f contains points at level $\mu(e) - 2$ in this cross section, it follows that f is incident to at least q vertices of $\mathcal{A}^{(e)}(\mathcal{F})$ whose levels are between $\mu(e) - q - 2$ and $\mu(e) - 2$. The number of vertices of $\mathcal{A}^{(e)}(\mathcal{F})$ whose levels fall in this range is therefore $\Omega(t_e q)$, which, by (2.3), implies that

$$\begin{aligned} \omega_{\le q}(e, \mathcal{F}) = \Omega(t_e q) &= \Omega\left(q t_e \cdot \frac{\omega_0(e, \mathcal{F})}{\lambda_s(t_e)} \right) \\ &\ge \frac{q}{\beta(k)} \cdot \omega_0(e, \mathcal{F}), \end{aligned} \tag{2.6}$$

where $\beta(k) = \Theta(\lambda_s(k)/k)$ is an extremely slowly growing function of k [9, 39].

Summing (2.6) over all edges e of $\mathcal{A}_{\le k}(\mathcal{F})$ for which $t_e > q$, adding the bound for the other edges of $\mathcal{A}_{\le k}(\mathcal{F})$, and observing that each edge-crossing between any two edges of $\mathcal{A}_{\le k}(\mathcal{F})$ with crossing number at most q is counted in this manner exactly once, we obtain

$$\varphi_0(\mathcal{F}, k) = \sum_{e \in \mathcal{A}_{\le k}(\mathcal{F})} \omega_0(e, \mathcal{F}) \le \frac{\beta(k)}{q} \varphi_{\le q}(\mathcal{F}, k) + \lambda_s(q)\psi(n, k),$$

which implies

$$\varphi_0(n, k) \le \frac{\beta(k)}{q} \varphi_{\le q}(n, k) + \lambda_s(q)\psi(n, k) \; . \tag{2.7}$$

In Lemma 2.3 below, we obtain the following upper bound on $\varphi_{\le q}(n, k)$:

$$\varphi_{\le q}(n, k) \le A(q + 1)^4 \varphi_0 \left(\left\lceil \frac{2n}{q+1} \right\rceil, \left\lceil \frac{2k}{q+1} \right\rceil \right) \tag{2.8}$$

for a constant $A > 0$. Substituting (2.1) and (2.8) in (2.7) and using the fact that $q > 4$, we obtain that

$$\varphi_0(n, k) \le B \left[(q + 1)^3 \beta(k) \varphi_0 \left(\left\lceil \frac{2n}{q+1} \right\rceil, \left\lceil \frac{2k}{q+1} \right\rceil \right) + \lambda_s(q) k^3 \psi\left(\frac{n}{k}\right) \right], \tag{2.9}$$

where $B > 0$ is another constant. The solution of (2.9) is

$$\varphi_0(n,k) \le Dk^{3+\varepsilon}\psi\left(\frac{n}{k}\right), \tag{2.10}$$

where $D = D(\varepsilon)$ is a sufficiently large constant depending on ε. The proof is by double induction on k and n. First, as already noted, the bound holds for $k \le k_0$ with an appropriate choice of D. For $k > k_0$ and for $n = k$, we have

$$\varphi_0(n,k) = O(n^3\beta(n)) \;=\; O(k^3\beta(k)) \;\le\; Dk^{3+\varepsilon}\psi(n/k),$$

provided D is chosen sufficiently large.

For $k > k_0$ and $n > k$, we have, by the induction hypothesis,

$$\varphi_0(n,k) \le B\left[(q+1)^3\beta(k)D\left\lceil\frac{2k}{q+1}\right\rceil^{3+\varepsilon}\psi\left(\left\lceil\frac{2n}{q+1}\right\rceil \Big/ \left\lceil\frac{2k}{q+1}\right\rceil\right) + \lambda_s(q)k^3\psi\left(\frac{n}{k}\right)\right].$$

Since

$$\left\lceil\frac{2n}{q+1}\right\rceil \Big/ \left\lceil\frac{2k}{q+1}\right\rceil \le \frac{2n}{k} \tag{2.11}$$

and, by condition (F3), $\psi(2n/k) \le c'\psi(n/k)$ for a constant $c' > 0$, we obtain

$$\varphi_0(n,k) \le Dk^{3+\varepsilon}\psi\left(\frac{n}{k}\right)\left[Bc'\frac{\beta(k)}{(q+1)^\varepsilon} + \frac{B}{D}\frac{\lambda_s(q)}{k^\varepsilon}\right].$$

Since

$$q = \left\lceil\left(\frac{c\lambda_s(k)}{k}\right)^{1/\varepsilon}\right\rceil$$

and $k^{1+\varepsilon^2/2} > c\lambda_s(k)$ (see (2.5)), we have

$$(q+1)^\varepsilon > c\frac{\lambda_s(k)}{k} = \Omega(\beta(k)) \quad \text{and} \quad q \le \left\lceil k^{\varepsilon/2}\right\rceil.$$

Hence, $\lambda_s(q) = o(k^\varepsilon)$, and we thus obtain

$$Bc'\frac{\beta(k)}{(q+1)^\varepsilon} + \frac{B}{D}\frac{\lambda_s(q)}{k^\varepsilon} < 1,$$

provided that c and k_0 are chosen sufficiently large. Hence,

$$\varphi_0(n,k) \le Dk^{3+\varepsilon}\psi\left(\frac{n}{k}\right).$$

This establishes Theorem 2.1. □

To complete the above proof, we establish the promised lemma.

LEMMA 2.3. *Let $n > k > q$ be three natural numbers. There exists a constant $A > 0$ such that*

$$\varphi_{\le q}(n,k) \le A(q+1)^4\varphi_0\left(\left\lceil\frac{2n}{q+1}\right\rceil, \left\lceil\frac{2k}{q+1}\right\rceil\right).$$

Proof. We prove the lemma using a probabilistic argument that is similar to, though slightly different from, the one used by Clarkson and Shor [25]; see also [64].

Set $p = 1/(q+1)$. We choose a subset $R \subseteq \mathcal{F}$ by selecting each function of $\mathcal{F}$ independently with probability p, so the expected size of R is np. Set $k' = \lceil 2kp \rceil$, and let us bound from below the expected value of $\varphi_0(R, k')$. For an edge-crossing (e, e', σ), let $I(e, e', \sigma)$ denote the indicator function whose value is 1 if the level of $e \cap \ell_\sigma$ in $\mathcal{A}(R)$ is at most k' and $(e, e', \sigma) \in \mathcal{C}_0(e, R)$. Otherwise, $I(e, e', \sigma) = 0$. Then

$$\begin{aligned} \mathbf{E}[\varphi_0(R, k')] &= \sum_{e \in \mathcal{A}(\mathcal{F})} \sum_{j=0}^{\mu(e)-2} \sum_{(e,e',\sigma) \in \mathcal{C}_j(e,\mathcal{F})} \Pr[\, I(e, e', \sigma) = 1 \,] \\ &\geq \sum_{e \in \mathcal{A}_{\leq k}(\mathcal{F})} \sum_{j=0}^{q} \sum_{(e,e',\sigma) \in \mathcal{C}_j(e,\mathcal{F})} \Pr[\, I(e, e', \sigma) = 1 \,]. \end{aligned} \tag{2.12}$$

Fix an edge-crossing $(e, e', \sigma) \in \mathcal{C}_j(e, \mathcal{F})$ with $e \in \mathcal{A}_{\leq k}(\mathcal{F})$, and $j \leq q$. Let f_1, f_2 (resp., f_3, f_4) be the functions of $\mathcal{F}$ whose intersection curve contains e (resp., e'). Notice that $f_1, \ldots, f_4$ are distinct. It is easily seen that $I(e, e', \sigma) = 1$ if and only if the following three events occur simultaneously:

(E_1) $\{f_1, f_2, f_3, f_4\} \subseteq R$.

(E_2) None of the j functions of $\mathcal{F}$ whose graphs intersect ℓ_σ between e and e' is chosen in R.

(E_3) Among the $\mu(e) - j - 2$ functions whose graphs intersect ℓ_σ below e', at most $k' - 2$ are chosen in R. That is, $\mu_R(e' \cap \ell_\sigma) \leq k' - 2$.

Since the subevents E_1, E_2, E_3 are independent, we have

$$\Pr[\, I(e, e', \sigma) = 1 \,] = \Pr[\, E_1 \,] \cdot \Pr[\, E_2 \,] \cdot \Pr[\, E_3 \,].$$

We have $\Pr[\, E_1 \,] = p^4$ and $\Pr[\, E_2 \,] = (1-p)^j$. Therefore, it suffices to obtain a lower bound on $\Pr[E_3]$.

First consider the case where $q \geq k/6$. $\mu_R(e' \cap \ell_\sigma)$ is obviously at most $k' - 2$ if none of the functions whose graphs intersect ℓ_σ below e' are chosen in R. Therefore

$$\Pr[\, E_3 \,] \geq (1-p)^{\mu(e)-j-2} > (1-p)^{k-j} \geq (1-p)^{6q-j}.$$

Consequently,

$$\begin{aligned} \Pr[\, I(e, e', \sigma) = 1 \,] &\geq p^4 (1-p)^j (1-p)^{6q-j} \\ &= \frac{1}{(q+1)^4} \left(1 - \frac{1}{q+1}\right)^{6q} \\ &> \frac{1}{e^6} \cdot \frac{1}{(q+1)^4}. \end{aligned} \tag{2.13}$$

The last inequality follows from the fact that $(1 - 1/(q+1))^q > 1/e$.

Next, assume that $q < k/6$. First, observe that $\Pr[E_3] = 1$ for $\mu(e) \leq k'$, so we may assume that $\mu(e) > k'$. If we let $g_1, \ldots, g_m$, for $m = \mu(e) - j - 2 < k$, denote the functions whose graphs intersect ℓ_σ below e', and define X_i to be the random variable whose value is 1 if $g_i \in R$ and 0 otherwise, then

$$\mu_R(e' \cap \ell_\sigma) = \sum_{i=1}^{m} X_i.$$

By definition, X_i is 1 with probability p and 0 with probability $1-p$. Therefore

$$\mathbf{E}[\,\mu_R(e' \cap \ell_\sigma)\,] \;=\; mp < kp$$

and, by Markov's inequality,

$$\Pr[\,\mu_R(e' \cap \ell_\sigma) > k'-2\,] \;<\; \frac{kp}{k'-2} \quad\le\; \frac{kp}{2kp-2}.$$

Hence,

$$\begin{aligned}\Pr[\,E_3\,] &= 1 - \Pr[\,\mu_R(e' \cap \ell_\sigma) > k'-2\,] \\ &> 1 - \frac{1}{2(1-1/kp)}.\end{aligned}$$

Since $q < k/6$, we have

$$kp \;=\; \frac{k}{q+1} \;\ge\; \frac{k}{2q} \;>\; 3,$$

which implies

$$\Pr[\,E_3\,] \;>\; 1 - \frac{1}{2(1-1/3)} \;=\; \frac{1}{4}.$$

Therefore

$$\Pr[\,I(e,e',\sigma)=1\,] \;\ge\; p^4(1-p)^q \cdot \frac{1}{4}.$$

Since $p = 1/(q+1)$ and $(1-1/(q+1))^q > 1/e$, we have

$$\Pr[\,I(e,e',\sigma)=1\,] \;\ge\; \frac{1}{4e}\cdot\frac{1}{(q+1)^4} \;>\; \frac{1}{e^6}\cdot\frac{1}{(q+1)^4}. \tag{2.14}$$

Substituting (2.13) and (2.14) into (2.12), we get

$$\begin{aligned}\mathbf{E}[\,\varphi_0(R,k')\,] &\ge \sum_{e\in\mathcal{A}_{\le k}(\mathcal{F})}\sum_{j=0}^{q}\frac{1}{e^6(q+1)^4}\cdot\varphi_j(e,\mathcal{F}) \\ &= \frac{1}{e^6(q+1)^4}\varphi_{\le q}(\mathcal{F},k).\end{aligned} \tag{2.15}$$

On the other hand,

$$\begin{aligned}\mathbf{E}[\,\varphi_0(R,k')\,] &\le \sum_{i\ge 0}\Pr[2ipn \le |R| < 2(i+1)pn]\cdot\varphi_0\left(\lceil 2(i+1)pn\rceil, k'\right) \\ &\le \varphi_0\left(\lceil 6pn\rceil, k'\right) + \sum_{i\ge 3}\Pr[|R|\ge 2ipn]\cdot\varphi_0(\lceil 2(i+1)pn\rceil, k').\end{aligned} \tag{2.16}$$

The weak bound on $\varphi_0(m,k)$ in (2.4) and the fact that $\psi(m) = O(m^{2+\varepsilon})$ imply that for any $\rho \ge 1$,

$$\varphi_0(\rho m, k') \le \rho^3\varphi_0(m,k'). \tag{2.17}$$

We use the following form of Chernoff's bound to estimate the second term in (2.16); see [57, p. 72]. Let n be an integer, and for $1 \le i \le n$, let X_i be a random variable which is 1 with probability p and 0 with probability $1-p$. Assume that the X_i's are independent, and put $X = \sum_{i=1}^{n} X_i$. Then for any $\delta > 2e$,

$$\Pr[\, X > \delta np \,] \; < \; 2^{-\delta np}. \tag{2.18}$$

If we let X_i be a random variable which is 1 if the ith function in $\mathcal{F}$ is in R and 0 otherwise, then $|R| = X$. Since $p = 1/(q+1) \ge 1/n$, (2.18) implies that for $i \ge 3$,

$$\Pr[\, |R| > 2inp \,] < 2^{-2i}. \tag{2.19}$$

Substituting (2.17) and (2.19) into (2.16), we obtain

$$\begin{aligned}\mathbf{E}[\, \varphi_0(R,k') \,] &\le 3^3\varphi_0(\lceil 2pn\rceil, k') + \sum_{i\ge 3} 2^{-2i}(i+1)^3 \cdot \varphi_0(\lceil 2pn\rceil, k') \\ &\le A'\varphi_0\left(\left\lceil \frac{2n}{q+1}\right\rceil, \left\lceil \frac{2k}{q+1}\right\rceil\right), \qquad (2.20)\end{aligned}$$

where

$$A' = 3^3 + \sum_{i\ge 3}(i+1)^3 2^{-2i} > 0$$

is a constant. Combining (2.15) and (2.20), we obtain

$$\begin{aligned}\varphi_{\le q}(n,k) &\le e^6(q+1)^4 \cdot A'\varphi_0\left(\left\lceil \frac{2n}{q+1}\right\rceil, \left\lceil \frac{2k}{q+1}\right\rceil\right) \\ &\le A(q+1)^4\varphi_0\left(\left\lceil \frac{2n}{q+1}\right\rceil, \left\lceil \frac{2k}{q+1}\right\rceil\right) \qquad (2.21)\end{aligned}$$

for an appropriate constant A. This completes the proof of the lemma and thus also of Theorem 2.1. $\square$

As promised in the beginning of the section, an inspection of the proof of Theorem 2.1 shows that it continues to apply, with minor and straightforward modifications, to the cases of partially defined or piecewise-algebraic functions. We therefore obtain the following result.

COROLLARY 2.4. *Let $\mathcal{F}$ be a collection of n partially defined, or piecewise-algebraic, bivariate functions, each of constant description complexity and satisfying conditions* (F2)–(F3). *For any integer $k \le n$, the combinatorial complexity of $\mathcal{A}^{\star}_{\le k}(\mathcal{F})$ is $O(k^{3+\varepsilon}\psi(n/k))$ for any $\varepsilon > 0$, where the constant of proportionality depends on ε and on the maximum complexity of any of the functions in $\mathcal{F}$.*

3. Shallow cuttings. In this section we first define $(1/r)$-cuttings of $\mathcal{A}_{\le k}(\mathcal{F})$ and then prove the existence of such cuttings with small size.

Let $\mathcal{F}$ be a collection of bivariate functions satisfying conditions (F1) and (F3). We call a region (or cell) $\Delta \subseteq \mathbb{R}^3$ *primitive* if there exists a set $D = D(\Delta)$ of at most six functions in $\mathcal{F}$ such that $\Delta \in \mathcal{A}^{\star}(D)$ and *weakly primitive* if it is the intersection of two primitive cells. For a primitive or a weakly primitive cell Δ, let $\mathcal{F}_\Delta \subseteq \mathcal{F}$ be the set of functions whose graphs intersect the interior of Δ and put $n_\Delta = |\mathcal{F}_\Delta|$. A set Ξ of pairwise openly disjoint weakly primitive cells is called a $(1/r)$-*cutting of* $\mathcal{A}_{\le k}(\mathcal{F})$

if the union of Ξ contains $\mathcal{A}_{\leq k}(\mathcal{F})$ and if $n_\Delta \leq n/r$ for every $\Delta \in \Xi$. We also refer to such a cutting as a *shallow cutting* in the arrangement $\mathcal{A}(\mathcal{F})$.

By the ε-net theory [40] and the result of [18], there exists a $(1/r)$-cutting of the entire $\mathcal{A}(\mathcal{F})$ that consists of $O(r^3\beta(r)\log^3 r)$ *primitive* cells, where $\beta(r)$ is the extremely slowly growing function defined in the previous section. Moreover, if r is a constant, such a cutting can be constructed in $O(n)$ deterministic time, using the technique of [50]. We now prove the existence of small-size $(1/r)$-cuttings of $\mathcal{A}_{\leq k}(\mathcal{F})$.

THEOREM 3.1. *Let $\mathcal{F}$ be a collection of n bivariate functions, as above, and let $k, r < n$ be integers. Set $q = k(r/n) + 1$. Then there exists a $(1/r)$-cutting Ξ of $\mathcal{A}_{\leq k}(\mathcal{F})$ whose size is at most $C_1 q^{3+\varepsilon}\psi(r/q)$, where $C_1 = C_1(\varepsilon)$ is an appropriate constant. Moreover, if $r = O(1)$, a $(1/r)$-cutting of this size can be computed in $O(n)$ time.*

Matoušek [52] proved a similar result for the case of linear functions. We follow his proof, but we need to enhance it with additional machinery.

Proof. For a subset $\mathcal{G} \subseteq \mathcal{F}$ and an integer $j < n$, let $\mathcal{T}_j(\mathcal{G})$ be the set of primitive cells Δ in $\mathcal{A}^\star(\mathcal{G})$ that lie completely in $\mathcal{A}_{\leq j}(\mathcal{F})$ (i.e., the level of all points in Δ with respect to $\mathcal{F}$ is at most j). In other words,

$$\mathcal{T}_j(\mathcal{G}) = \{\Delta \in \mathcal{A}^\star(\mathcal{G}) \mid \mu_{\mathcal{F}}(p) \leq j \text{ for all } p \in \Delta\}.$$

Let $\mathcal{T}_j = \bigcup_{\mathcal{G} \subseteq \mathcal{F}} \mathcal{T}_j(\mathcal{G})$.

Fix $p = r/n$. Choose a subset $R \subseteq \mathcal{F}$ by selecting each function of $\mathcal{F}$ independently with probability p. For each cell $\Delta \in \mathcal{A}^\star(R)$ that has a point whose level with respect to $\mathcal{F}$ is at most k, we do the following. If $|n_\Delta| \leq n/r$, we add Δ to Ξ. Otherwise, suppose that $(t-1)n/r < n_\Delta \leq tn/r$ for some integer $t > 1$. (We then say that the *excess* of such a Δ is t.) We compute a $(1/t)$-cutting Ξ_Δ of $\mathcal{A}(\mathcal{F}_\Delta)$, which consists of $O(t^3\beta(t)\log^3 t)$ primitive cells, clip each cell of Ξ_Δ within Δ, and add the resulting (weakly primitive) cells to Ξ. This yields a $(1/r)$-cutting Ξ of $\mathcal{A}_{\leq k}(\mathcal{F})$. We now show that the expected size of Ξ is as asserted in the theorem.

Let $\eta(p, t, j)$ denote the expected number of those cells Δ in $\mathcal{T}_j(R)$ whose excess is at least t. If any cell $\Delta \in \mathcal{A}^\star(R)$ with excess t intersects $\mathcal{A}_{\leq k}(\mathcal{F})$, then the level of all points of Δ with respect to $\mathcal{F}$ is at most $k + tn/r$. This is easily seen to imply

$$\mathbf{E}[|\Xi|] \ \leq \ \eta(p, 0, k) + \sum_{t \geq 1} O(t^3\beta(t)\log^3 t) \cdot \eta\left(p, t, k + \left\lceil \frac{tn}{r} \right\rceil\right).$$

A primitive cell $\Delta \in \mathcal{T}_j$ appears in $\mathcal{T}_j(R)$ if and only if $D(\Delta) \subseteq R$ and $\mathcal{F}_\Delta \cap R = \emptyset$. Using the same argument as in [52] (see also [7, 20]), one can show that for $t \geq 1$,

$$\eta(p, t, j) \leq c2^{-t} \cdot \eta\left(\frac{p}{t}, 0, j\right) \tag{3.1}$$

for a constant $c > 0$. We prove in Lemma 3.2 below that for any $0 < p < 1$ and for any integer $l < n$,

$$\eta(p, 0, k) = O\left((1 + kp)^\varepsilon (kp)^3 \psi\left(\frac{n}{k}\right) + (kp)^2 \min\left\{\psi\left(\frac{n}{k}\right), \left(\frac{n}{k}\right)^2\right\}\right). \tag{3.2}$$

By (3.1) and (3.2), for $t \geq 1$, we obtain

(3.3)
$$\eta\left(p, t, k + \left\lceil \frac{tn}{r} \right\rceil\right) = O\left(2^{-t}\left(1 + \frac{p(k + \lceil tn/r \rceil)}{t}\right)^{\varepsilon}\left(\frac{p(k + \lceil tn/r \rceil)}{t}\right)^{3}\psi\left(\frac{n}{k + tn/r}\right)\right)$$
$$= O\left(2^{-t}q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).$$

The last two inequalities follow because

$$\frac{p(k + \lceil tn/r \rceil)}{t} = \left(\frac{pk}{t} + \frac{\lceil tn/r \rceil}{tn/r}\right) \leq 2 + kp = 2q$$

and

$$\frac{n}{k + tn/r} = \frac{r}{t + kp} \leq \frac{r}{1 + kp} = \frac{r}{q};$$

in particular, the second term in (3.2) is dominated by the first term.

Next, we prove that

$$\eta(p, 0, k) = O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).$$

If $kp \geq 1$, then by (3.2),

$$\eta(p, 0, k) = O\left((1 + kp)^{\varepsilon}(kp)^{3}\psi\left(\frac{n}{k}\right)\right).$$

Condition (F3) allows us to write

$$\psi\left(\frac{n}{k}\right) = \left(\frac{n}{k}\right)^{\alpha}\beta\left(\frac{n}{k}\right)$$

for some constant $\alpha \leq 2$ and an appropriate small function β. Therefore,

$$(1 + kp)^{\varepsilon}(kp)^{3}\psi\left(\frac{n}{k}\right) = (1 + kp)^{3+\varepsilon}\left(\frac{kp}{1 + kp}\right)^{3}\left(\frac{np}{kp}\right)^{\alpha}\beta\left(\frac{np}{kp}\right)$$
$$= O\left(q^{3+\varepsilon}\left(\frac{kp}{1 + kp}\right)^{3-\alpha}\psi\left(\frac{np}{1 + kp}\right)\right)$$
$$= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).$$

The first inequality follows because (F3) is easily seen to imply that $\beta(\rho m) = O(\beta(m))$ for any $\rho \leq 2$, say.

On the other hand, if $kp < 1$ and $\alpha = 2 - \delta$, for some $\delta > 0$, then $q \leq 2$ and (3.2) implies that

$$\eta(p, 0, k) = O\left((1 + kp)^{\varepsilon}(kp)^{2}\psi\left(\frac{n}{k}\right)\right)$$
$$= O\left(q^{2+\varepsilon}\left(\frac{kp}{1 + kp}\right)^{2}\left(\frac{np}{kp}\right)^{2-\delta}\beta\left(\frac{np}{kp}\right)\right)$$
$$= O\left(q^{2+\varepsilon}\left(\frac{kp}{1 + kp}\right)^{\delta}\left(\frac{r}{q}\right)^{\alpha}\beta\left(\frac{r}{q}\cdot\frac{1 + kp}{kp}\right)\right)$$
$$= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).$$

The last inequality follows because, by (F3), $\beta(\rho m) \leq \rho^{\delta}\beta(m)$ for any $\rho \geq 1$ and for any $\delta > 0$, provided that m is sufficiently large.

Finally, if $kp < 1$ and $\alpha = 2$, then again $q \leq 2$ and (3.2) implies that

$$\begin{aligned}
\eta(p,0,k) &= O\left((1+kp)^{\varepsilon}(kp)^3\psi\left(\frac{n}{k}\right) + (kp)^2\left(\frac{n}{k}\right)^2\right)\\
&= O\left(q^{3+\varepsilon}\left(\frac{kp}{1+kp}\right)\left(\frac{np}{1+kp}\right)^2\beta\left(\frac{np}{kp}\right) + r^2\right)\\
&= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right) + q^2\left(\frac{r}{q}\right)^2\right)\\
&= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).
\end{aligned}$$

Hence,

$$\begin{aligned}
\mathbf{E}[\,|\Xi|\,] &= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right) + \sum_{t\geq 1} O(t^3 2^{-t}\beta(t)\log^3 t)\cdot O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right)\\
&= O\left(q^{3+\varepsilon}\psi\left(\frac{r}{q}\right)\right).
\end{aligned}$$

Finally, observe that the proof of this theorem is constructive, so we immediately obtain a randomized algorithm for computing Ξ. If $r = O(1)$, then the expected running time is $O(n)$. The algorithm can be made deterministic using the method of conditional probabilities, as described in [16, 50]. ◻

To complete the analysis, we establish the promised lemma.

LEMMA 3.2. *For any real number $0 < p < 1$ and for any integer $k < n$,*

$$\eta(p,0,k) = O\left((1+kp)^{\varepsilon}(kp)^3\psi\left(\frac{n}{k}\right) + (kp)^2\min\left\{\psi\left(\frac{n}{k}\right), \left(\frac{n}{k}\right)^2\right\}\right).$$

Proof. Let R be, as above, a random subset of $\mathcal{F}$ obtained by choosing each element of $\mathcal{F}$ independently with probability p. We want to bound the expected number of those primitive cells Δ in $\mathcal{A}^{\star}(R)$ that belong to $\mathcal{T}_k(R)$, i.e., the level of all points in Δ with respect to $\mathcal{F}$ is at most k. Each cell $\Delta \in \mathcal{A}^{\star}(R)$ is of one of the following three types (these types are not mutually exclusive):

(C1) Δ contains a vertex v of $\mathcal{A}(R)$; if $\Delta \in \mathcal{T}_k(R)$, then $\mu_{\mathcal{F}}(v) \leq k$.

(C2) Δ has a vertical face parallel to the yz-plane and tangent to an edge e of $\mathcal{A}(R)$ at some point w; if $\Delta \in \mathcal{T}_k(R)$, then $\mu_{\mathcal{F}}(w) \leq k$.

(C3) Δ has a vertical edge induced by an edge-crossing $(e, e', \sigma) \in \mathcal{C}_0(e, R)$ for some $e \in \mathcal{A}(R)$. If $\Delta \in \mathcal{T}_k(R)$, then $\mu_{\mathcal{F}}(e \cap \ell_{\sigma}) \leq k$.

A vertex $v \in \mathcal{A}_{\leq k}(\mathcal{F})$ appears as a vertex of $\mathcal{A}(R)$ if and only if the three functions whose graphs contain v are chosen in R. Moreover, under the general position assumption, each vertex of $\mathcal{A}(R)$ appears in $O(1)$ cells of $\mathcal{A}^{\star}(R)$. Therefore, the expected number of cells in $\mathcal{T}_k(R)$ of type (C1) is at most

$$\begin{aligned}
O\Bigg(\sum_{v\in\mathcal{A}_{\leq k}(\mathcal{F})} \Pr\left[\, v \text{ is a vertex of } \mathcal{A}(R)\,\right]\Bigg) &= O(p^3\psi(n,k)) = O((kp)^3)\psi\left(\frac{n}{k}\right)\\
&= O((1+kp)^{\varepsilon}(kp)^3)\left(\frac{n}{k}\right).
\end{aligned}$$

Similarly, the number of cells in $\mathcal{T}_k(R)$ of type (C2) can be bounded by

$$O\Bigl(\sum_w \Pr\left[\, w \text{ appears in } \mathcal{A}(R)\,\right]\Bigr),$$

where the sum is taken over all locally x-extremal points on the edges of $\mathcal{A}_{\leq k}(\mathcal{F})$. The probability of such a point appearing in $\mathcal{A}_{\leq k}(\mathcal{F})$ is p^2. By conditions (F1) and (F2), the number of x-extremal points in the relative interior of the edges of $\mathcal{A}_{\leq k}(\mathcal{F})$ is $O(n^2)$. On the other hand, another application of the Clarkson–Shor technique [25] implies that the number of such x-extremal points is $O(k^2\psi'(n/k))$, where $\psi'(n/k)$ is the number of x-extremal points in the relative interior of the edges of the lower envelope of at most n/k functions of $\mathcal{F}$. Since $\psi'(n/k) \leq \psi(n/k)$, we can bound the number of type (C2) cells by

$$O\left((kp)^2 \min\left\{\psi\left(\frac{n}{k}\right), \left(\frac{n}{k}\right)^2\right\}\right).$$

Next, consider the cells of type (C3). Let

$$\Phi(R,k) = \bigcup_{e\in\mathcal{A}(R)} \left\{(e,e',\sigma) \in \mathcal{C}_0(e,R) \mid \mu_{\mathcal{F}}(e\cap \ell_\sigma) \leq k\right\}.$$

In order to bound the expected number of cells of type (C3), it suffices to obtain an upper bound on the expected value of $|\Phi(R,k)|$.

We proceed as in the proof of Lemma 2.3. Let e be an edge of $\mathcal{A}_{\leq k}(\mathcal{F})$, and let $(e,e',\sigma) \in \mathcal{C}_j(e,\mathcal{F})$ for some $0 \leq j \leq k-2$. The edge-crossing (e,e',σ) contributes (exactly) one edge-crossing to $\Phi(R,k)$ if and only if the four functions of $\mathcal{F}$ defining e and e' are chosen in R and none of the j functions whose graphs intersect the line ℓ_σ between e and e' is chosen in R. The probability that (e,e',σ) gives rise to one edge-crossing in $\Phi(R,k)$ is thus $p^4(1-p)^j$. Since every edge-crossing of $\Phi(R,k)$ is of this form, we obtain

$$\begin{aligned}
\mathbf{E}\bigl[|\Phi(R,k)|\bigr] &= \sum_{j=0}^{k-2} \sum_{e\in\mathcal{A}_{\leq k}(\mathcal{F})} \sum_{(e,e',\sigma)\in\mathcal{C}_j(e,\mathcal{F})} p^4(1-p)^j \\
&= \sum_{j=0}^{k-2} p^4(1-p)^j \varphi_j(\mathcal{F},k) \\
&= p^4\varphi_0(\mathcal{F},k) + \sum_{j=1}^{k-2} p^4(1-p)^j \left(\varphi_{\leq j}(\mathcal{F},k) - \varphi_{\leq j-1}(\mathcal{F},k)\right) \\
&\leq p^4(1-p)^{k-2}\varphi_{\leq k-2}(n,k) + \sum_{j=0}^{k-3} p^5(1-p)^j \varphi_{\leq j}(n,k) \\
&\leq p^4(1-p)^{k-2}\cdot A(k-1)^4\varphi_0\left(\left\lceil\frac{2n}{k-1}\right\rceil, \left\lceil\frac{2k}{k-1}\right\rceil\right) \\
&\quad + \sum_{j=0}^{k-3} p^5(1-p)^j \cdot A(j+1)^4\varphi_0\left(\left\lceil\frac{2n}{j+1}\right\rceil, \left\lceil\frac{2k}{j+1}\right\rceil\right) \\
&\qquad \text{(applying Lemma 2.3)}
\end{aligned}$$

$$
\begin{aligned}
&\le Ap^4(1-p)^{k-2}(k-1)^4 \cdot D\left\lceil \frac{2k}{k-1}\right\rceil^{3+\varepsilon} \psi\left(\frac{2n}{k}\right) \\
&\quad + \sum_{j=0}^{k-3} Ap^5(1-p)^j(j+1)^4 \cdot D\left\lceil \frac{2k}{j+1}\right\rceil^{3+\varepsilon} \psi\left(\frac{2n}{k}\right) \\
&\qquad \text{(applying (2.10))} \\
&= O\left((kp)^{3+\varepsilon}\psi\left(\frac{n}{k}\right)\right)\left[p(k-1)(1-p)^{k-2} + \sum_{j=0}^{k-3} p^2(j+1)(1-p)^j\right] \\
&= O\left((1+kp)^{\varepsilon}(kp)^3\psi\left(\frac{n}{k}\right)\right).
\end{aligned}
$$

The last inequality follows from the fact that the sum in the square brackets is $1-(1-p)^{k-1} \le 1$. This completes the proof of the lemma and thus the proof of Theorem 3.1. $\square$

Remark 3.3. We note that if $\mathcal{F}$ is a collection of n univariate functions, then $\mathcal{A}^\star(\mathcal{F})$ has only $O(n^2)$ cells and $\psi(n,k) = O(k^2\lambda_s(n/k))$, where s is the maximum number of intersections between the graphs of any pair of functions in $\mathcal{F}$. Following the preceding analysis, one can obtain the following result.

THEOREM 3.4. *Let $\mathcal{F}$ be a collection of n univariate algebraic functions, each of constant maximum degree, and let $k, r < n$ be integers. Set $q = k(r/n)+1$. Then there exists a $(1/r)$-cutting Ξ of $\mathcal{A}_{\le k}(\mathcal{F})$ whose size is at most $O(q^2\lambda_s(r/q))$. Moreover, if $r = O(1)$, a $(1/r)$-cutting of this size can be computed in $O(n)$ time.*

4. Range searching.

An immediate consequence of Theorem 3.1 is an efficient solution of the following problem: Given a set $\mathcal{F} \subseteq \mathbf{F}$ of n bivariate functions satisfying the conditions (F1) and (F3), preprocess it into a (static) data structure, so that for a query point $p = (x_p, y_p, z_p)$, the subset $\{f \in \mathcal{F} \mid f(x_p, y_p) < z_p\}$ can be reported efficiently. We also want to update $\mathcal{F}$ dynamically by inserting and deleting functions of $\mathbf{F}$ into/from $\mathcal{F}$.

Chazelle et al. [18] have shown that $\mathcal{F}$ can be preprocessed in time $O(n^{3+\varepsilon})$ into a (static) data structure of size $O(n^{3+\varepsilon})$ so that a query can be answered in $O(\log n + \xi)$ time, where ξ is the output size. In this section we present a data structure that uses $O(\psi(n) \cdot n^{\varepsilon})$ storage and preprocessing time, answers a query in $O(\log n + \xi)$ time, and can be updated dynamically at a small (amortized) cost. We first present a static data structure and then explain how to dynamize it.

4.1. A static data structure.

We construct a tree data structure T to answer queries of the above form. Each node v of T is associated with a subset $\mathcal{F}^v \subseteq \mathcal{F}$ and a weakly primitive cell $\Delta_v \subseteq \mathbb{R}^3$. The root is associated with $\mathcal{F}$ and with the entire $\mathbb{R}^3$. Fix a sufficiently large constant r and set $k = \lfloor 2n/r \rfloor$. If $|\mathcal{F}|$ is less than some prespecified constant n_0, then T consists of just the root. Otherwise, we compute a $(1/r)$-cutting Ξ of $\mathcal{A}_{\le k}(\mathcal{F})$, whose size, by Theorem 3.1 and (F3), is $O(\psi(r))$ (where the constant of proportionality is independent of r). For each cell $\Delta \in \Xi$, let $\mathcal{F}_\Delta^- \subseteq \mathcal{F}$ be the set of functions whose graphs either intersect Δ or lie below Δ. If $|\mathcal{F}_\Delta^-| > 3n/r$, we discard Δ because Δ lies completely above $\mathcal{A}_{\le k}(\mathcal{F})$. Otherwise, we create a child $v = v_\Delta$ of the root, corresponding to Δ, put $\Delta_v = \Delta$ and $\mathcal{F}^v = \mathcal{F}_\Delta^-$, and recursively preprocess $\mathcal{F}^v$ (in this recursive processing, we use the same parameter r and set $k = \lfloor 2|\mathcal{F}^v|/r \rfloor$). The recursion stops when we reach nodes v with $|\mathcal{F}^v| \le n_0$. If $S(n)$ denotes the maximum space used by the data structure, on any set $\mathcal{F}$ of n functions of $\mathbf{F}$, then we obtain the recurrence

$$S(n) \leq \begin{cases} c_0, & n \leq n_0, \\ c_1\psi(r) \cdot S(3n/r) + c_2 n, & n > n_0, \end{cases}$$

where c_0, c_1, c_2 are appropriate constants (independent of r). Recall that, by (F3), the bound $\psi(n)$ can be written as $n^\alpha\beta(n)$, where $1 \leq \alpha \leq 2$ is some constant and where $\limsup_{n\to\infty} \beta(n)/n^\delta = 0$ for any $\delta > 0$. The solution of the recurrence is

$$S(n) \leq B\psi(n) \cdot n^\varepsilon,$$

for any $\varepsilon > 0$, where B depends on ε. The proof is by induction on n. The claim holds for all $n \leq n_0$, provided that B is chosen sufficiently large. For larger values of n we have, by the induction hypothesis,

$$\begin{aligned} S(n) &\leq c_1\psi(r) \cdot B\psi\left(\frac{3n}{r}\right) \cdot \left(\frac{3n}{r}\right)^\varepsilon + c_2 n \\ &\leq c_1 B r^\alpha \beta(r) \left(\frac{3n}{r}\right)^\alpha \beta\left(\frac{3n}{r}\right) \cdot \left(\frac{3n}{r}\right)^\varepsilon + c_2 n \\ &\leq B \cdot n^\alpha \beta(n) \cdot n^\varepsilon \left[\frac{3^{\alpha+\varepsilon} c_1 \beta(r)}{r^\varepsilon} + \frac{c_2 n^{1-\alpha-\varepsilon}}{B\beta(n)}\right] \\ &\leq B\psi(n) n^\varepsilon, \end{aligned}$$

provided that r and B are chosen sufficiently large (as functions of ε). The preprocessing time of the data structure is also easily seen to be $O(\psi(n) \cdot n^\varepsilon)$.

Let p be a query point. To answer the query for p, we trace a path in T, starting from the root. Suppose we are at a node v. If v is a leaf or if there is no child w of v for which $p \in \Delta_w$, we explicitly check for each $f \in \mathcal{F}^v$ (the set of functions associated with v) whether p lies above the graph of f, and we report those functions which satisfy this condition. The time spent at v is then $O(|\mathcal{F}^v|)$, but then either $|\mathcal{F}^v| \leq n_0$ or p lies above the $\lfloor 2|\mathcal{F}^v|/r \rfloor$-level in $\mathcal{A}(\mathcal{F}^v)$, so we report at least $\lfloor 2|\mathcal{F}^v|/r \rfloor$ functions. If there is a child w of v such that $p \in \Delta_w$, we recursively visit that child. The overall query time is $O(\log n + \xi)$, where ξ is the output size of the query. Hence, we obtain the following result.

THEOREM 4.1. *Given a family* $\mathbf{F}$ *of bivariate functions satisfying* (F1) *and* (F3) *and a set* $\mathcal{F} \subseteq \mathbf{F}$ *of size* n*, we can preprocess* $\mathcal{F}$ *(in an appropriate model of computation) in time* $O(\psi(n) \cdot n^\varepsilon)$ *into a data structure* $\Pi(\mathcal{F})$ *of size* $O(\psi(n) \cdot n^\varepsilon)$*, so that all* ξ *functions whose graphs lie below a query point* p *can be reported in time* $O(\log n + \xi)$.

Remark 4.2. If we are interested only in determining whether a query point lies below the graphs of all the functions of $\mathcal{F}$, we can construct the minimization diagram, $M(\mathcal{F})$, of $\mathcal{F}$ and preprocess $M(\mathcal{F})$ for planar point-location queries. The size of the data structure is $O(\psi(n))$, and a query can be answered in time $O(\log n)$; see, for example, [66].

4.2. A dynamic data structure. We can dynamize the above data structure using the technique of Agarwal and Matoušek [6]. Since the basic idea is the same, we give only a brief description of the structure. For the sake of simplicity, we describe a data structure that handles only deletions; insertions can then be handled using standard decomposition techniques [6, 15].

We need the following lemma, which is an easy consequence of a result by Chazelle et al. [18], but we give a somewhat simpler proof.

LEMMA 4.3. *Let Ξ be a cutting in $\mathbb{R}^3$ consisting of t cells. Ξ can be preprocessed in time $O(t^3)$ into a data structure of size $O(t^3)$, so that we can determine in $O(\log t)$ time the cell of Ξ containing a query point p, or we can conclude that there is no such cell.*

Proof. Assume that the faces of Ξ are xy-monotone, which will hold in our applications. (However, the proof also holds when this is not the case, by appropriately decomposing each face of the cutting cells (each of which is assumed to have constant description complexity) into a constant number of xy-monotone subfaces.) Project each face onto the xy-plane. Let Ξ^* be the set of the resulting planar patches. For a point $p \in \mathbb{R}^2$, let $\Lambda(p)$ be the list of faces of Ξ, sorted in the $+z$-direction, that intersect the z-vertical line passing through p. We compute the arrangement $\mathcal{A}(\Xi^*)$. Since the interiors of all cells of Ξ are pairwise disjoint, $\Lambda(p)$ is same for all points within a face of $\mathcal{A}(\Xi^*)$. For each face $\phi \in \mathcal{A}(\Xi^*)$, we store this sorted list, denoted by $\Lambda(\phi)$. This requires a total of $O(t^3)$ space and can be computed in time $O(t^3)$. (In fact, the storage and preprocessing time can be improved to $O(t^2)$ and $O(t^2 \log t)$, respectively, using persistent data structures, but the weaker bounds suffice for our purpose.) To locate a point $w \in \mathbb{R}^3$ in Ξ, we first find the face of $\mathcal{A}(\Xi^*)$ that contains the xy-projection of w and then search with w in the appropriate list $\Lambda(\phi)$ to determine the cell of Ξ containing w. If Ξ does not cover the entire $\mathbb{R}^3$, then the search in $\Lambda(\phi)$ can also determine whether w lies outside Ξ. For example, in the case of shallow cuttings, if w lies above all faces in $\Lambda(\phi)$, we can conclude that w does not lie in any cell of Ξ. □

We are now in position to describe the data structure. The overall data structure, denoted as $\Psi(\mathcal{F})$, is a tree of constant depth. Each node v of $\Psi(\mathcal{F})$ is associated with a subset $\mathcal{F}^v \subseteq \mathcal{F}$. The subtree rooted at any node of $\Psi(\mathcal{F})$ is reconstructed periodically after performing some deletions. Let m_v (resp., m) be the size of $\mathcal{F}^v$ (resp., $\mathcal{F}$) when the structure was last reconstructed, and let n_v (resp., n) denote the size of the current set $\mathcal{F}^v$ (resp., $\mathcal{F}$). Set $r = m^\delta$ for some sufficiently small $\delta > 0$. (The value of r is the same for all nodes of the tree, and it is updated only when the entire structure $\Psi(\mathcal{F})$ is reconstructed.) We reconstruct the subtree rooted at v after performing $m_v/2r$ deletions from that subtree. Hence, $n_v \geq m_v(1 - 1/2r)$. A function $f \in \mathcal{F}$ is said to be *relevant* for a cell Δ if the graph of f either intersects Δ or lies below Δ.

If $|\mathcal{F}| \leq r$, $\Psi(\mathcal{F})$ consists of a single node at which we preprocess $\mathcal{F}$ into the range-searching data structure, $\Pi(\mathcal{F})$, provided in Theorem 4.1. Otherwise, the root u of $\Psi(\mathcal{F})$ stores the following information:

(i) a partition of $\mathcal{F}^u$ into pairwise-disjoint sets $\mathcal{F}_1, \ldots, \mathcal{F}_t$, where $t \leq \lceil 1/\delta^2 \rceil$;

(ii) a cutting Ξ_i for each $1 \leq i \leq t$. Initially, Ξ_i is a $(1/r)$-cutting of $\mathcal{A}_{\leq k}(\bigcup_{j \geq i} \mathcal{F}_j)$, where $k = \lfloor m/r \rfloor$ and each function of $\mathcal{F}_i$ is relevant for at most

$$\kappa = 2C_1 \frac{\psi(r)}{r^{1-\delta}} \tag{4.1}$$

cells of Ξ_i, where C_1 is the constant arising in (4.2) below;

(iii) a data structure for point-location queries for each Ξ_i, as in Lemma 4.3;

(iv) a pointer to the subtree $\Psi(\mathcal{F}_{i,\Delta})$ for every i and for every $\Delta \in \Xi_i$, where $\mathcal{F}_{i,\Delta} \subseteq \mathcal{F}_i$ is the set of functions relevant for Δ;

(v) for each $1 \leq i \leq t$ and for every $f \in \mathcal{F}_i$, we store the set $L(f)$ of cells $\Delta \in \Xi_i$ for which f is relevant;

(vi) a counter $dcount_u$, which is initially set to $m/2r$;

(vii) the range-searching data structure $\Pi_u = \Pi(\mathcal{F}^u)$ on $\mathcal{F}^u$, as provided in Theorem 4.1.

This information is computed as follows. We first construct Π_u in $O(\psi(m)m^\varepsilon)$ time, using Theorem 4.1. Next, we construct the $\mathcal{F}_i$'s and Ξ_i's. Suppose we have already computed $\mathcal{F}_1, \ldots, \mathcal{F}_{i-1}$. Let $\overline{\mathcal{F}}_i = \mathcal{F}^u - (\mathcal{F}_1 \cup \cdots \cup \mathcal{F}_{i-1})$, and let $m_i = |\overline{\mathcal{F}}_i|$. If $m_i \leq m/r$, then Ξ_i consists of just one sufficiently large primitive cell and $\mathcal{F}_i = \overline{\mathcal{F}}_i$. Otherwise, set

$$r_i = r\frac{m_i}{m} \quad \text{and} \quad k = \left\lfloor \frac{m}{r} \right\rfloor = \left\lfloor \frac{m_i}{r_i} \right\rfloor.$$

Using Theorem 3.1, we compute a $(1/r_i)$-cutting Ξ_i for $\mathcal{A}_{\leq k}(\overline{\mathcal{F}}_i)$ whose size is at most $C_1\psi(r_i)$. For each cell $\Delta \in \Xi_i$, let $\overline{\mathcal{F}}_{i,\Delta} \subseteq \overline{\mathcal{F}}_i$ be the set of functions relevant for Δ. We have $|\overline{\mathcal{F}}_{i,\Delta}| \leq k + m_i/r_i \leq 2m_i/r_i$. Therefore

$$\sum_{\Delta \in \Xi_i} |\overline{\mathcal{F}}_{i,\Delta}| \leq C_1 \frac{2m_i}{r_i}\psi(r_i) \; = \; 2C_1 m_i \frac{\psi(r_i)}{r_i}. \tag{4.2}$$

We call a function "good" if it is relevant for at most κ cells. Then (4.2) implies that the number of "bad" functions is at most

$$2C_1 m_i \frac{\psi(r_i)}{r_i} \bigg/ 2C_1 \frac{\psi(r)}{r^{1-\delta}} \; = \; \frac{m_i}{r^\delta} \cdot \frac{\psi(r_i)}{r_i} \bigg/ \frac{\psi(r)}{r} \; \leq \; \frac{m_i}{m^{\delta^2}}.$$

Let $\mathcal{F}_i$ be the set of good functions of $\overline{\mathcal{F}}_i$. We now repeat the construction for $\overline{\mathcal{F}}_{i+1} = \overline{\mathcal{F}}_i - \mathcal{F}_i$. The above analysis implies that $|\overline{\mathcal{F}}_i| \leq m^{1-i\delta^2}$, so the process terminates after at most $\lceil 1/\delta^2 \rceil$ steps. The construction of the structure is now completed by recursively building the data structure $\Psi(\mathcal{F}_{i,\Delta})$ for each $\Delta \in \Xi_i$. (The remaining pointers, lists, and counters are trivial to produce.) The size of the data structure and the preprocessing time are both $O(\psi(n)n^\varepsilon)$, arguing as in [6], and the choice of r ensures that the depth of $\Psi(\mathcal{F})$ is $O(1)$.

Deleting a function. We first describe how to delete a function f from $\mathcal{F}$. We traverse $\Psi(\mathcal{F})$ in a top-down fashion. Suppose we are at a node v. If v is a leaf or $dcount_v = 1$, we reconstruct $\Psi(\mathcal{F}^v)$, including Π_v, with the current set of functions in $\mathcal{F}^v$. Otherwise, we decrease the variable $dcount_v$ by 1, and for each cell $\Delta \in L(f)$, we delete f from $\Psi(\mathcal{F}_{i,\Delta})$ recursively. In this case, we do not modify Π_v, so it may contain some of the functions that have been deleted. Following the same analysis as in [6], we can show that the (amortized) deletion time is $O(\psi(n)/n^{1-\varepsilon})$.

Answering a query. Let p be a query point. We again traverse $\Psi(\mathcal{F})$ in a top-down fashion. Suppose we are at a node v. If v is a leaf, we report in $O(\log n + \xi)$ time all ξ functions of $\mathcal{F}^v$ whose graphs lie below p, using the secondary range-searching data structure Π_v. (Note that if v is a leaf, then Π_v does not store any function that has already been deleted.) If v is an internal node, then for each i, we find in $O(\log n)$ time the cell $\Delta_i \in \Xi_i$ that contains p, using the point-location data structure of Lemma 4.3. If Δ_i is defined for every i, we recursively search in $\Psi(\mathcal{F}_{i,\Delta})$ for all $i \leq t$. Otherwise, there is an $i \leq t$ such that p does not lie in any cell of Ξ_i. Let ξ_v (resp., ξ'_v) be the number of functions in $\mathcal{F}^v$ at present (resp., when $\Psi(\mathcal{F}^v)$ was last constructed) whose graphs lie below p. Then $\xi'_v > m_v/r$. At most $m_v/2r$ functions have been deleted since $\Psi(\mathcal{F}^v)$ was last constructed; therefore

$$\xi_v \geq \xi'_v - m_v/2r > \xi'_v - \xi'_v/2 = \xi'_v/2.$$

We query Π_v and report those functions of the query output that have not been deleted so far. The time spent at v is $O(\log n + \xi'_v) = O(\log n + \xi_v)$. The total query time is thus $O(\log n + \xi)$, where ξ is the total output size. We have thus shown the following.

THEOREM 4.4. *Given a family* $\mathbf{F}$ *of* n *bivariate functions satisfying* (F1) *and* (F3) *and a subset* $\mathcal{F} \subseteq \mathbf{F}$ *of size* n*, we can preprocess* $\mathcal{F}$ *(in an appropriate model of computation) in time* $O(\psi(n)\cdot n^{\varepsilon})$ *into a data structure of size* $O(\psi(n)\cdot n^{\varepsilon})$*, so that all* ξ *functions whose graphs lie below a query point can be reported in time* $O(\log n+\xi)$ *and a function* $f \in \mathbf{F}$ *can be inserted into or deleted from* $\mathcal{F}$ *in* $O(\psi(n)/n^{1-\varepsilon})$ *(amortized) time. The same dynamic data structure can be used to determine in* $O(\log n)$ *time whether a query point lies below the graphs of all the functions in the current set* $\mathcal{F}$*.*

5. Computing the ($\le k$)-level. Next, consider the problem of computing $\mathcal{A}_{\le k}(\mathcal{F})$, where $\mathcal{F} \subseteq \mathbf{F}$ is a set of n bivariate functions satisfying (F1) and (F3) and k is any integer $< n$. If $\mathcal{F}$ is a set of linear functions in $\mathbb{R}^3$, this can be done by the randomized algorithm of [1] (see also [27, 58, 59]), whose expected time is $O(nk^2 + n\log^3 n)$, but this algorithm does not extend to nonlinear functions. The algorithm that we present below computes all 0-, 1-, and 2-dimensional faces of $\mathcal{A}_{\le k}(\mathcal{F})$, along with their incidence relations. A slight enhancement of the algorithm can store $\mathcal{A}_{\le k}(\mathcal{F})$ into a data structure, so that for a query point p, we can determine in $O(\log n)$ time whether $\mu_{\mathcal{F}}(p) \le k$, and, if so, we return the value of $\mu_{\mathcal{F}}(p)$. We construct $\mathcal{A}_{\le k}(\mathcal{F})$ using a divide-and-conquer algorithm, based on our shallow cutting theorem, as follows. A similar algorithm was developed by Clarkson [23] for computing the ($\le k$)-level in an arrangement of hyperplanes.

Choose a sufficiently large constant integer parameter r. If $k \ge n/r$, construct the entire arrangement $\mathcal{A}(\mathcal{F})$ and output the first k levels. The time and storage required are $O(n^3 \log n)$ and $O(n^3)$, respectively. This can be done using one of several known techniques; e.g., for each function $f \in \mathcal{F}$, we compute the intersection curves of f with all the other functions of $\mathcal{F}$ and then compute, by a line-sweep algorithm that takes $O(n^2 \log n)$ time, all the faces of $\mathcal{A}_{\le k}(\mathcal{F})$ that lie on the graph of f.

Otherwise, the parameter $q = 1 + kr/n$ is at most 2. Using Theorem 3.1, we compute in $O(n)$ time a $(1/r)$-cutting Ξ of size $O(\psi(r))$ for $\mathcal{A}_{\le k}(\mathcal{F})$. For each cell $\Delta \in \Xi$, compute the set $\mathcal{F}_\Delta$ of the functions whose graphs cross Δ and the set $\mathcal{F}^-_\Delta$ of the functions whose graphs pass fully below Δ. If $|\mathcal{F}^-_\Delta| > k$, then discard Δ. For the remaining cells, we have $|\mathcal{F}_\Delta| \le n/r$ and $k' = |\mathcal{F}^-_\Delta| \le k$. We now construct recursively $\mathcal{A}_{\le k-k'}(\mathcal{F}_\Delta)$, repeat this construction over all cells of Ξ, and glue together the resulting outputs. The gluing step involves merging those faces which are computed by separate subproblems and which are portions of the same face of $\mathcal{A}(\mathcal{F})$. This can be accomplished in time proportional to the total output size by a rather straightforward procedure, operating on each surface separately, whose details are omitted.

Let $T(n,k)$ denote the maximum time needed to construct $\mathcal{A}_{\le k}(\mathcal{F})$, where $|\mathcal{F}| = n$. Then we obtain the following recurrence:

$$T(n,k) \le \begin{cases} c_1\psi(r)\cdot T\left(\dfrac{n}{r}, k\right) + c_2 n, & \text{for } k < n/r, \\ c_3 n^3 \log n, & \text{for } k \ge n/r, \end{cases}$$

where c_1, c_2, c_3 are appropriate constants. It is easily seen, arguing as in the analysis leading to Theorem 4.1, that the solution of this recurrence is

$$T(n,k) = O(k^3 n^{\varepsilon} \psi(n/k)),$$

which is close to optimal in the worst case. Hence, we obtain the following result.

THEOREM 5.1. *Let* $\mathbf{F}$ *be a family of bivariate functions satisfying* (F1) *and* (F3). *Given a set* $\mathcal{F} \subseteq \mathbf{F}$ *of size* n *and an integer parameter* $k \leq n$, *we can construct* $\mathcal{A}_{\leq k}(\mathcal{F})$ *in time* $O(k^3 n^\varepsilon \psi(n/k))$ *in an appropriate model of computation.*

If we want to store $\mathcal{A}_{\leq k}(\mathcal{F})$ into a data structure for answering point-location queries of the form described above, we keep the cuttings obtained during the above recursive construction in a tree-like structure T. This yields a data structure similar to the one described in the previous section.

For a point p, we answer a query as follows. We trace a path in T from the root down, maintaining a global count of the number of functions that are known to pass below p. Initially we set this count to 0. When we visit a node v, we locate p (by brute force) in the cutting Ξ_v associated with v. If no cell of the cutting contains p, we report that p lies above the k-level and stop. Otherwise, let Δ be the cell of Ξ_v containing p. We add $k' = |\mathcal{F}_\Delta^-|$ to the global count and continue the search at the child of v associated with Δ. When we reach a leaf of T, we explicitly test each of the constant number of functions stored at that leaf whether it passes below p, update the global count accordingly, and output its final value. We thus obtain the following result.

THEOREM 5.2. *Given a collection* $\mathcal{F}$ *of* n *bivariate functions satisfying* (F1)–(F3) *and an integer parameter* $k < n$, *we can preprocess* $\mathcal{A}_{\leq k}(\mathcal{F})$ *into a data structure of size* $O(k^3 n^\varepsilon \psi(n/k))$, *in time* $O(k^3 n^\varepsilon \psi(n/k))$, *so that for any query point* p, *we can determine in* $O(\log n)$ *time whether* $\mu_{\mathcal{F}}(p) \leq k$ *and, if so, we compute the value of* $\mu_{\mathcal{F}}(p)$.

6. Nearest-neighbor searching.

Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in the plane, and let $\delta(\cdot, \cdot)$ be a given "distance function" defined on $\mathbb{R}^2 \times \mathbb{R}^2$. Normally, we would take δ to be a metric or a convex distance function, but many of the results obtained below apply for more general "reasonable" functions δ. The function δ is called *reasonable* if the family

$$\mathbf{F} = \{\delta((x,y),(a_1,a_2)) \mid (a_1,a_2) \in \mathbb{R}^2\}$$

satisfies conditions (F1) and (F3) of section 2. For example, all L_p metrics, for integer $1 \leq p \leq \infty$, are reasonable. Here we do not assume any metric-like properties of δ, so δ can be fairly arbitrary. We want to store S into a data structure so that points can be inserted into or deleted from S and for any query point q, we can efficiently compute a *nearest neighbor* of q in S under the function δ (i.e., we want to return a point $p_i \in S$ so that $\delta(q, p_i) = \min_{1 \leq j \leq n} \delta(q, p_j)$). We may also want to return k nearest neighbors of a query point for some $k \geq 1$. Let

$$\mathcal{F} = \{f_i(x,y) = \delta((x,y), p_i) \mid 1 \leq i \leq n\}.$$

For a given point q, finding a nearest neighbor of q in S is equivalent to computing a function of $\mathcal{F}$ that attains the lower envelope $E_{\mathcal{F}}$ of $\mathcal{F}$ at q. Notice that the complexity of the lower envelope $E_{\mathcal{F}}$ is the same as the complexity of the Voronoi diagram of S under the distance function δ [31] if δ is indeed a metric or a convex distance function.

The nearest-neighbor searching problem, also known as the post-office problem, has been widely studied because of its numerous applications [4, 6, 23, 53], but most of the work to date deals with the case where δ is the L_1, L_2, or L_∞ metric and where S does not change dynamically. If δ is a reasonable function, then we can construct the minimization diagram of the resulting envelope $E_{\mathcal{F}}$ and preprocess it for planar point-location queries. Therefore a nearest-neighbor query can be answered in $O(\log n)$ time using $O(n^{2+\varepsilon})$ storage, but this data structure is difficult to dynamize. On the other hand, using the range-searching data structures of Agarwal and Matoušek [5], S can be preprocessed in $O(n \log n)$ time into a linear-size data structure so that a query can be answered in $O(n^{1/2+\varepsilon})$ time. This data structure can be updated in $O(\log^2 n)$ time per insertion or deletion. Since the Agarwal–Matoušek technique will be useful for our applications, we describe it briefly. Let

$$\Gamma = \{\delta((x,y),(\alpha,\beta)) \le r \mid \alpha,\beta,r \in \mathbb{R}\}$$

be the set of all "disks" under the distance function δ. We preprocess S, in $O(n \log n)$ time, into a linear size data structure for the counting version of Γ-range searching (in which, for any given $\gamma \in \Gamma$, we want to compute $|S \cap \gamma|$), as described in [5]. Using this data structure for a query point q and a real parameter r, we can determine in $O(n^{1/2+\varepsilon})$ time whether the distance between q and its nearest neighbor in S is less than, greater than, or equal to r. Plugging this procedure into the parametric-searching technique of Megiddo [55] (in the same manner as in [4]), we can answer a nearest-neighbor query, under the distance function δ, in $O(n^{1/2+\varepsilon})$ time. This structure, however, fails to answer a query in $O(\log n)$ time even for the Euclidean metric.

In this section we present a data structure, based on Theorem 4.4, that uses only $O(\psi(n) \cdot n^{\varepsilon})$ space and preprocessing time, answers a query in $O(\log n)$ time, and inserts or deletes a point in $O(\psi(n)/n^{1-\varepsilon})$ time. Moreover, it can also report k nearest neighbors of a query point in time $O(\log n + k)$. These bounds are significantly better than the previously mentioned bounds if we allow insertions/deletions or if we want to answer k-nearest-neighbor queries. As an example, for the case of L_p metrics, we can obtain a nearest-neighbor searching data structure of size $O(n^{1+\varepsilon})$ that can answer a query in $O(\log n)$ time (or a k-nearest-neighbor query in time $O(\log n + k)$) and that can insert/delete a point in $O(n^{\varepsilon})$ time. Compared with the data structure described in [6], we improve the query time significantly at a slight increase in the update time and in the size of the data structure.

6.1. A static data structure. As in section 4, we first describe a static data structure. This data structure is more complicated than simply storing the Voronoi diagram of S (or, more generally, the minimization diagram of the envelope $E_{\mathcal{F}}$), but it allows us to answer efficiently k-nearest-neighbor queries. We preprocess S, or, rather, the associated collection $\mathcal{F}$, into the range-searching data structure $\Pi(\mathcal{F})$ of Theorem 4.1. For a point $q \in \mathbb{R}^2$, we find a function of $\mathcal{F}$ that attains $E_{\mathcal{F}}$ at q, and thus find a nearest neighbor of q in S, as follows. Let $\vec{\ell}$ be the vertical line in $\mathbb{R}^3$ passing through q and oriented in the $(+z)$-direction. We trace a path in the tree-structure $T = \Pi(\mathcal{F})$, starting from the root. Suppose we are at a node $v \in T$; v is associated with a subset $\mathcal{F}^v \subseteq \mathcal{F}$ and a weakly primitive cell Δ_v. Inductively, assume that $E_{\mathcal{F}}(q) = E_{\mathcal{F}^v}(q)$. If v is a leaf, we explicitly search for a function of $\mathcal{F}^v$ that attains $E_{\mathcal{F}^v}$ at q, report that function, and terminate the query. Recall that if v is not a leaf, then each child of v is associated with a cell of a $(1/r)$-cutting Ξ_v of $\mathcal{A}_{\le k}(\mathcal{F}^v)$ (for appropriate parameters r and k). We determine the highest cell Δ of Ξ_v (in the $(+z)$-direction) intersected by $\vec{\ell}$. Since Ξ_v covers $\mathcal{A}_{\le k}(\mathcal{F}^v)$, Δ is the last cell in Ξ_v intersecting $\vec{\ell}$, and $\mathcal{F}_{\Delta}^-$ contains all functions of $\mathcal{F}^v$ whose graphs either intersect Δ or lie below Δ, it easily follows that $E_{\mathcal{F}_{\Delta}^-}(q) = E_{\mathcal{F}^v}(q) = E_{\mathcal{F}}(q)$. Hence, we can recursively search at the child w corresponding to Δ (where $\mathcal{F}^w = \mathcal{F}_{\Delta}^-$). The overall query time is obviously $O(\log n)$.

6.2. A dynamic data structure. If S is allowed to change dynamically, we use the dynamic data structure described in section 4 with the following modification. For a $(1/r)$-cutting Ξ, we now need a data structure that, for a given query point q in the xy-plane, can determine in $O(\log|\Xi|)$ time the highest cell of Ξ intersected by the vertical line $\vec{\ell}$ passing through q. As in Lemma 4.3, we project the facets of cells in Ξ onto the xy-plane, let Ξ^* be the set of resulting regions in the plane, and compute the arrangement $\mathcal{A}(\Xi^*)$. Since the (interiors of) cells in Ξ are pairwise disjoint, the highest cell of Ξ intersected by a vertical line erected from any point within a face $\phi \in \mathcal{A}(\Gamma)$ is the same. By storing this cell for each face ϕ and by preprocessing $\mathcal{A}(\Xi^*)$ for planar point location queries, we can compute in $O(\log|\Xi|)$ time the highest cell of Ξ intersected by a vertical line.

A query is answered as follows: Suppose we are at a node v. If v is a leaf, we compute $E_{\mathcal{F}^v}(q)$, using the procedure just described (or the simpler one that stores explicitly the lower envelope). Otherwise, for each Ξ_i, we find in $O(\log n)$ time the highest cell Δ intersected by $\vec{\ell}$ by locating q in $\mathcal{A}(\Xi_i^*)$ (as in Lemma 4.3). For each $1 \leq i \leq t$, we recursively compute the function f_i that attains the lower envelope $E_{\mathcal{F}_{i,\Delta}}$ at q. Finally, we return the function that attains the envelope of $\{f_1, \ldots, f_t\}$ at q. Since the procedure visits $O(1)$ nodes and spends $O(\log n)$ time at each node, the overall query time is $O(\log n)$. Updating the structure is done as in section 4.

THEOREM 6.1. *Let S be a set of n points in the plane and let δ be any reasonable distance function, as above. We can preprocess S in time $O(\psi(n)n^\varepsilon)$ into a data structure of size $O(\psi(n)n^\varepsilon)$ so that a nearest-neighbor query in S can be answered in $O(\log n)$ time. Moreover, we can insert and delete points into/from S in $O(\psi(n)/n^{1-\varepsilon})$ time per update operation. Here $\psi(n)$ is an upper bound, satisfying* (F3)*, for the maximum complexity of the envelope $E_{\mathcal{F}}$, as defined above (if δ is a metric or a convex distance function, then $\psi(n)$ is also (an upper bound for) the maximum complexity of the Voronoi diagram of a set of n points in the plane, under the distance function δ).*

Remark 6.2. For a point q, finding a farthest neighbor of q in S is equivalent to finding a function of $\mathcal{F}$ that attains the upper envelope of $\mathcal{F}$ at q. By reversing the direction of the z-axis, we can use the same data structure for answering farthest-neighbor queries.

6.3. Reporting the κ nearest (farthest) neighbors. The above query procedures can be modified, so that for a query point q and an integer $\kappa \leq n$, we can compute the κ nearest (or farthest) neighbors of q in S in time $O(\log n + \kappa)$. Consider the static data structure, for instance. We again follow a path of the tree, starting from the root. If we are at a leaf v or at a node v with $|\mathcal{F}^v| \leq 3\kappa r$, we explicitly check all functions of $\mathcal{F}^v$ to determine the κ nearest neighbors of q in S and terminate the query. Otherwise, we find the highest cell of Ξ_v intersected by the corresponding vertical line $\vec{\ell}$ and recursively visit that child. For the dynamic version, the query procedure is somewhat more involved, but a query can still be answered in $O(\log n + \kappa)$ time. Hence, we have the following theorem.

THEOREM 6.3. *Let S be a set of n points in the plane, and let δ be any reasonable distance function, as above. We can preprocess S in time $O(\psi(n)n^\varepsilon)$ into a data structure of size $O(\psi(n)n^\varepsilon)$, so that for a given point p and an integer $\kappa \leq n$, the κ nearest (or farthest) neighbors of p can be computed in $O(\log n + \kappa)$ time. Moreover, a point can be inserted into or deleted from S in $O(\psi(n)/n^{1-\varepsilon})$ time.*

6.4. Some special cases. If δ is an L_p-metric, for any integer $1 \le p \le \infty$, then $\psi(n) = O(n)$ [45]. Similarly, if δ is an *(additive) weighted Euclidean* metric, i.e., each point $p_i \in S$ has a weight w_i and $\delta(q, p_i) = d(q, p_i) + w_i$, where $d(\cdot, \cdot)$ is the Euclidean distance, then also $\psi(n) = O(n)$ [46]. A linear bound also holds when the sites are pairwise-disjoint line segments [47]. Hence, we obtain the following result.

COROLLARY 6.4. *Let S be a set of n points in the plane, and let δ be any L_p-metric or any weighted Euclidean metric. We can preprocess S in $O(n^{1+\varepsilon})$ time into a data structure of size $O(n^{1+\varepsilon})$ so that points can be inserted into or deleted from S in time $O(n^{\varepsilon})$ per update and a nearest-neighbor query can be answered in $O(\log n)$ time. We can also construct a data structure with the same performance bounds for the case when S is a set of n pairwise disjoint segments (or other simply shaped objects) in the plane and δ is the Euclidean metric. In all these cases, the k nearest neighbors of a query point in the plane can be reported in time $O(\log n + k)$.*

Space/query-time trade-off. If the bound $\psi(n)$ is rather large, say, $O(n^{2+\varepsilon})$, we can obtain a space/query-time trade-off by combining Theorem 6.1 with the linear-size data structure of [5] for nearest-neighbor searching, mentioned at the beginning of this section, in a rather standard manner [22]. For example, if $\psi(n) = O(n^{2+\varepsilon})$, then for any parameter $n \le m \le n^2$, we can store S into a data structure of size $O(m^{1+\varepsilon})$ so that a nearest-neighbor query can be answered in time $O(n^{1+\varepsilon}/\sqrt{m})$. Insertions and deletions can be performed in $O(m^{1+\varepsilon}/n)$ time, per update. Hence, we can conclude the following.

THEOREM 6.5. *Let S be a set of n points in the plane, $n \le m \le n^2$ be a real parameter, and δ be a reasonable distance function, as above. We can preprocess S in time $O(m^{1+\varepsilon})$ into a data structure of size $O(m^{1+\varepsilon})$ so that a nearest-neighbor (or a farthest-neighbor) query can be answered in $O(n^{1+\varepsilon}/\sqrt{m})$ time. Moreover, we can insert and delete points in $O(m^{1+\varepsilon}/n)$ time per update.*

Remark 6.6. If the number of queries and the number of updates are both $O(n)$, then an optimal choice for m is $n^{4/3}$, for which the entire sequence of operations can be performed in $O(n^{4/3+\varepsilon})$ time.

The dynamic bichromatic closest-pair problem. Next, we consider the following dynamic bichromatic closest-pair problem: Let R and B be two sets of points in the plane, of a total of at most n points. We wish to update R and B dynamically and to return the closest pair between R and B after each update operation.[4] We use the following result of Eppstein [35].

LEMMA 6.7. *Let R and B be two sets of points in the plane, with a total of at most n points. Let δ be a distance function for which we can store a set of n points into a data structure that supports insertions, deletions, and nearest-neighbor queries in $O(T(n))$ time per operation. Then a closest pair between R and B can be maintained in $O(T(n)\log n)$ time per insertion and in $O(T(n)\log^2 n)$ time per deletion.*

If δ is an L_p metric or any weighted Euclidean metric, then by Corollary 6.4, $T(n) = O(n^{\varepsilon})$. On the other hand, if δ is any reasonable distance function, then by choosing $m = n^{4/3}$ in Theorem 6.5, $T(n) = O(n^{1/3+\varepsilon})$. Plugging these values into the above lemma, we obtain the following result.

THEOREM 6.8. *Let R and B be two sets of points in the plane, with a total of n points. We can store $R \cup B$ in a dynamic data structure of size $O(n^{1+\varepsilon})$ that maintains a closest pair in $R \times B$, under any L_p-metric or any weighted Euclidean metric, in $O(n^{\varepsilon})$ time per insertion or deletion. If we use an arbitrary, reasonable*

[4]A *closest pair* between R and B under a distance function δ is a pair of points $p \in R$, $q \in B$ such that $\delta(p, q) = \min_{p' \in R, q' \in B} \delta(p', q')$.

distance function, then the storage increases to $O(n^{4/3+\varepsilon})$ *and the update time to* $O(n^{1/3+\varepsilon})$.

Using this theorem and extending the analysis of Agarwal et al. [2], as done in [35], we can obtain an efficient procedure for maintaining a minimum spanning tree T of a set of points under any L_p metric, as follows. For an angle $\alpha \leq \pi/2$ and a unit vector d, let $Cone(d, \alpha) = \{\mathbf{x} \mid \angle(\mathbf{x}, d) \leq \alpha\}$. We call a pair of point sets P, Q *α-separated* if there exist a point z and a direction d such that $P \subseteq z + Cone(-d, \alpha)$ and $Q \subseteq z + Cone(d, \alpha)$. Let α_0 be a sufficiently small angle, and let

$$\mathcal{C} = \{(P_1, Q_1), \ldots, (P_u, Q_u)\}$$

be a family of α_0-separated pairs such that for every pair $p, q \in S$, there is an i with $p \in P_i$ and $q \in Q_i$; we will refer to $\mathcal{C}$ as an *α_0-cover* of S. Consider the graph G whose vertices are the points of S and whose edges are pairs of points (p_i, q_i), $1 \leq i \leq u$, so that (p_i, q_i) is a δ-closest pair between P_i and Q_i, where δ is an L_p-metric. The weight of an edge (p_i, q_i) is $\delta(p_i, q_i)$. Following the same argument used by Agarwal et al. [2, Lemmas 1–4], but extending their Lemma 3 to the case of an L_p-metric, one can show that G contains a minimum spanning tree of S as a subgraph.

Using a standard divide-and-conquer approach, an α_0-cover $\mathcal{C}$ of S, with the property that each point of S is contained in $O(\log^2 n)$ pairs, can be constructed in $O(n \log^2 n)$ time; see, e.g., [2]. Moreover, $\mathcal{C}$ can be maintained in $O(\log^2 n)$ time per insertion or deletion. Insertion or deletion of a point changes $O(\log^2 n)$ edges of the graph G, and Theorem 6.8 implies that the new edges can be found in $O(n^\varepsilon \log^2 n)$ time. The problem of maintaining a minimum spanning tree of S now reduces to maintaining the family $\mathcal{C}$, a closest pair for each pair $(P_i, Q_i) \in \mathcal{C}$, and a minimum spanning tree of the resulting graph G. Holm, de Lichtenberg, and Thorup [41] have shown that a minimum spanning tree of a graph with n vertices can be updated in $O(\log^4 n)$ time for each insertion or deletion of an edge into/from the graph. Hence we obtain the following result.

THEOREM 6.9. *A minimum spanning tree of a set of n points in the plane, under any L_p-metric, can be maintained in $O(n^\varepsilon)$ time for each insertion or deletion operation using $O(n^{1+\varepsilon})$ storage.*

Remark 6.10. We believe that Theorem 6.9 also holds for more general metrics, but we did not attempt such a generalization in this paper.

7. Minimum-weight bipartite Euclidean matching.

Let $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_n\}$ be two sets of points in the plane. We wish to compute a *minimum-weight bipartite Euclidean matching* M of P and Q. A matching of P and Q is a set of n pairs $(p, q) \in P \times Q$ such that each point of $P \cup Q$ appears in exactly one pair. The weight of a pair is the Euclidean distance between its points, and the weight of a matching is the sum of the weights of its pairs. The standard *Hungarian method* [43, 44] yields an $O(n^3)$-time algorithm for computing M. Exploiting the fact that the weights are Euclidean distances, Vaidya obtained an $O(n^{2.5} \log n)$-time algorithm for computing M [70]. A number of efficient algorithms have been developed for computing a minimum-weight bipartite Euclidean matching when P and Q have some special structure [10, 17, 49], but no progress has been made when P and Q are arbitrary sets of points in the plane. Recently, Efrat and Itai [33] have proposed an $O(n^{1.5+\varepsilon})$-time algorithm for computing a bottleneck Euclidean bipartite matching, in which one wishes to minimize the maximum weight of an edge in the matching.

In this section, we show that Vaidya's algorithm, for arbitrary sets P, Q, can be modified so that its running time improves to $O(n^{2+\varepsilon})$. One can equally consider the nonbipartite version of the problem, where we are given just one set P of $2n$ points in the plane and wish to compute a minimum-weight Euclidean matching in P, that is, a partition of P into n pairs of points so that the sum of the intrapair Euclidean distances is minimized. Vaidya has also given in [70] an $O(n^{2.5} \log n)$-time algorithm for the nonbipartite case. Recently, the bound was improved by Varadarajan to $O(n^{1.5} \log n)$ [72].

For the sake of completeness, we first give a brief sketch of Vaidya's algorithm for the bipartite case and then show how it can be improved using the new machinery of this paper.

The bipartite matching problem can be formulated as a linear program:

$$\begin{array}{lll} \min & \sum_{i,j} d(p_i, q_j) x_{ij} \quad \text{subject to} & \\ & \sum_{j=1}^{n} x_{ij} = 1, & i = 1, \ldots, n, \\ & \sum_{i=1}^{n} x_{ij} = 1, & j = 1, \ldots, n, \\ & x_{ij} \geq 0, & i, j = 1, \ldots, n, \end{array}$$

where (p_i, q_j) is an edge of M if and only if $x_{ij} = 1$. The dual linear program is

$$\begin{array}{lll} \max & \sum_{i} \alpha_i + \sum_{j} \beta_j \quad \text{subject to} & \\ & \alpha_i + \beta_j \leq d(p_i, q_j), & i, j = 1, \ldots, n, \end{array}$$

where α_i (resp., β_j) is the dual variable associated with the point p_i (resp., q_j).

The Hungarian method computes a matching in n phases, each of which augments the matching by one edge and updates the dual variables. Let X be the current matching computed so far by the algorithm. Initially, we set $X = \emptyset$, $\alpha_i = 0$, and $\beta_j = \min_i \ d(p_i, q_j)$ for all i, j. An edge (p_i, q_j) is called *admissible* if $d(p_i, q_j) = \alpha_i + \beta_j$. A vertex of $P \cup Q$ is called *exposed* if it is not incident to any edge of X. An *alternating path* is one that alternately traverses edges of X and edges not in X, starting with an edge not in X. An alternating path between two exposed vertices is called an *augmenting path*.

During each phase, we search for an augmenting path consisting only of admissible edges, as follows. For each exposed point $q \in Q$, we grow (in an implicit manner) an "alternating tree" whose paths are alternating paths starting at q. More precisely, each point of $P \cup Q$ in an alternating tree is reachable from its root by an alternating path that consists only of admissible edges. For a point w of P (resp., Q), the path leading to w ends at an edge not in X (resp., in X). Let S (resp., T) denote the set of points of Q (resp., P) that lie in any alternating tree. In the beginning of each phase, S is the set of exposed vertices of Q and $T = \emptyset$. Let

$$\delta = \min_{p_i \in P - T, q_j \in S} \{d(p_i, q_j) - \alpha_i - \beta_j\}.$$

At each step, the algorithm takes one of the following actions, depending on whether $\delta = 0$ or $\delta > 0$:

Case 1. $\delta = 0$. Let (p_i, q_j), for $p_i \in P - T$ and $q_j \in S$ be an admissible edge ($\delta = 0$ implies that such an edge must exist). If p_i is an exposed vertex, an augmenting path has been found, so the algorithm moves to the next phase (see below for more details).

Otherwise, let q_k be the vertex such that $(p_i, q_k) \in X$. The algorithm adds the edges (p_i, q_j) and (p_i, q_k) to all the alternating trees that contain the vertex q_j. It also adds the point p_i to T and the point q_k to S.

Case 2. $\delta > 0$. The algorithm updates the dual variables, as follows. For each vertex $p_i \in T$, it sets $\alpha_i = \alpha_i - \delta$, and for each $q_j \in S$, it sets $\beta_j = \beta_j + \delta$.

The algorithm repeats these steps until it reaches an exposed vertex of P, thereby obtaining an augmenting path. If an augmenting path Π is found, we delete the edges of $\Pi \cap X$ from the current matching X, add the other edges of Π to X (thereby increasing the size of the current matching by 1), and move to the next phase. This completes the description of the algorithm. Further details of the algorithm and the proof of its correctness can be found in [44, 70].

For arbitrary graphs, each step can be implemented in $O(n)$ time. Since there are at most n phases and each phase consists of $O(n)$ steps, the total running time of the above procedure is $O(n^3)$. Vaidya suggested the following approach to expedite the running time of each step. Maintain a variable Δ and associate a weight $w(\cdot)$ with each point in $P \cup Q$. In the beginning of each phase, $\Delta = 0$ and $w(p_i) = \alpha_i$, $w(q_j) = \beta_j$ for each $1 \le i, j \le n$. During each step, the weights and Δ are updated, but the values of the dual variables remain unchanged. This is done as follows. If Case 1 occurs, then we set $w(p_i) = \alpha_i + \Delta$ and $w(q_k) = \beta_k - \Delta$ and do not change the value of Δ. If Case 2 occurs, then we set $\Delta = \Delta + \delta$ and do not change the weights. Notice that for each $q_j \in S$, the current value of β_j is equal to $w(q_j) + \Delta$, and, similarly, for each $p_i \in T$, the current value of α_i is equal to $w(p_i) - \Delta$. Also, the current value of the dual variables for other points is equal to their values at the beginning of the phase. At the end of each phase, the value of the dual variables can be computed from Δ and from the weights of the corresponding points. The weight of each point changes only once during a phase, namely, when it is added either to S or to T. Moreover, at any time during a phase,

$$\delta = \min_{p_i \in P-T, q_j \in S} \{d(p_i, q_j) - w(p_i) - w(q_j)\} - \Delta.$$

Hence, δ can be computed during each step by maintaining the weighted closest pair between S and $P - T$. Since each step requires at most two update operations (inserting a point into S and deleting a point from $P - T$), each step can be performed in $O(n^\varepsilon)$ time by the algorithm provided in Theorem 6.8. (The current setup is slightly different than that of section 6, because here the weighted distance between two points subtracts weights associated with both points. Nevertheless, the analysis of section 3 leading to Theorem 4.4 and the way in which this theorem is used in section 6 still apply.) If the distance between two points is measured by an arbitrary reasonable distance function, each step of the above algorithm will require $O(n^{1/3+\varepsilon})$ time with an appropriately increased storage (the distance function continues to be reasonable when weights are added to the points, as above). We can thus conclude the following.

THEOREM 7.1. *Given two sets P, Q, each consisting of n points in the plane, a minimum-weight bipartite Euclidean matching between P and Q can be computed in $O(n^{2+\varepsilon})$ time, using $O(n^{1+\varepsilon})$ space. If we use an arbitrary, reasonable distance function for computing distances between points of P and Q, a minimum-weight bipartite matching between P and Q can be computed in time $O(n^{7/3+\varepsilon})$, using $O(n^{4/3+\varepsilon})$ space.*

The above algorithm can be extended to obtain faster algorithms for several other problems related to the bipartite matching. For example, consider the following *Euclidean transportation* problem, which is a generalization of the minimum-weight bipartite Euclidean matching problem: Let $P = \{p_1, \ldots, p_u\}$ be a set of u "supply" points in the plane, each with a supply value a_i, and let $Q = \{q_1, \ldots, q_v\}$ be a set of v "demand" points in the plane, each with a demand value b_j, such that $\sum_{i=1}^{u} a_i = \sum_{j=1}^{v} b_j$. Assume that the cost of transporting unit commodity from p_i to q_j is $d(p_i, q_j)$. We wish to find out how to satisfy all the demands at the minimum cost. Atkinson and Vaidya [13] gave an $O((u+v)^{2.5}\log(u+v)\log N)$-time algorithm, where $N = \max\{\max_i a_i, \max_j b_j\}$. Plugging Theorem 6.8 into their algorithm, we can improve the running time to $O((u+v)^{2+\varepsilon}\log N)$. See [14, 44, 70] for some other applications.

8. Maintaining the intersection of congruent balls in three dimensions.

The next application of our techniques is an efficient algorithm for dynamically maintaining the intersection of congruent balls in $\mathbb{R}^3$, under insertions and deletions of balls. With no loss of generality, we assume that the balls have radius 1. For a point $p \in \mathbb{R}^3$, let $B(p)$ denote the ball of radius 1 centered at p. Let S be a set of n points in $\mathbb{R}^3$. Let $\mathcal{B} = \mathcal{B}(S) = \{B(p) \mid p \in S\}$, and let $K(S) = \bigcap_{p\in S} B(p)$ be the common intersection of $\mathcal{B}$. We wish to maintain $K(S)$ as we update S dynamically by inserting and deleting points. Although the complexity of $K(S)$ is $O(n)$ [36], a sequence of m updates in S may cause $\Omega(mn + m^2)$ changes in the structure of $K(S)$, even if we perform only insertions. Consequently, we cannot hope to maintain $K(S)$ explicitly if we seek fast update time. Instead, we store $K(S)$ into a data structure so that a number of different types of queries can be answered efficiently. The simplest query is whether a query point p lies in $K(S)$. A stronger type of query is to determine efficiently, after each update, whether $K(S)$ is empty. In this section, we present a data structure that answers these two queries efficiently.

Let $B^+(p)$ (resp., $B^-(p)$) denote the region consisting of all points that lie in or above (resp., in or below) $B(p)$. Let $K^+(S) = \bigcap_{p\in S} B^+(p)$ and $K^-(S) = \bigcap_{p\in S} B^-(p)$. If we regard the boundary of each region $B^+(p)$ for $p \in S$ as the graph of a partially defined bivariate function, then (the nonvertical portion of) $\partial K^+(S)$ is the graph of the upper envelope of these functions restricted to the domain, $D(S)$, of intersection of the xy-projections of the balls $B(p)$, $p \in S$. We denote this upper envelope by $z = F^+(\mathbf{x})$. The same argument as in [36] implies that $K^+(S)$ also has linear complexity. In fact, the surfaces $\partial B^+(p)$ behave similar to a family of pseudoplanes—any pair of them intersects in a single connected unbounded curve (or does not intersect at all), and, assuming general position, any triple of them intersects at a single point (or does not intersect at all). Similar and symmetric properties hold for the region $K^-(S)$, and we use the notation $z = F^-(\mathbf{x})$ to denote the graph of its (nonvertical) boundary.

A query point $p = (p_x, p_y, p_z)$ lies in $K(S)$ if and only if $F^+(p_x, p_y) \le p_z \le F^-(p_x, p_y)$. Therefore, the problem of determining whether $p \in K(S)$ reduces to evaluating $F^+(p_x, p_y)$ and $F^-(p_x, p_y)$. By Theorem 6.1, we can store $\{B^+(p) \mid p \in S\}$ and $\{B^-(p) \mid p \in S\}$ into two data structures, each of size $O(n^{1+\varepsilon})$, so that a point can be inserted into or deleted from S in $O(n^\varepsilon)$ time and so that for a query point $p = (p_x, p_y, p_z)$, the functions $F^+(p_x, p_y)$ and $F^-(p_x, p_y)$ can be evaluated in $O(\log n)$ time.[5]

The same data structure can be used to determine whether $K(S)$ is empty, although this is a considerably more involved operation. Observe that the boundaries

[5]This is an example where the given bivariate functions are only partially defined, so we use Corollary 2.4 instead of Theorem 2.1.

of the regions $K^+(S)$ and $K^-(S)$ are both (weakly) xy-monotone—one of them is a convex surface and the other is concave. These properties can be exploited to obtain a fast procedure for detecting (after each update) whether $K^+(S)$ and $K^-(S)$ intersect by applying a variant of the parametric searching technique [55], as proposed by Toledo [69]. For the sake of completeness, we include here a brief description of this technique in the context of the problem under consideration.

We wish to find a point $\mathbf{x}^* \in \mathbb{R}^2$ such that $F^+(\mathbf{x}^*) \leq F^-(\mathbf{x}^*)$; this point will serve as a witness to the fact that $K^+(S)$ and $K^-(S)$ intersect. In fact, we will seek a point $\mathbf{x}^*$ at which

$$F^{\Delta}(\mathbf{x}^*) \stackrel{\text{def}}{=} F^+(\mathbf{x}^*) - F^-(\mathbf{x}^*)$$

is minimized; the minimum is negative if and only if the regions $K^+(S)$ and $K^-(S)$ intersect. Let A_s denote the query procedure that computes $F^{\Delta}(\mathbf{x})$ for a given $\mathbf{x}$ in $O(\log n)$ time.

We first consider the 1-dimensional problem, where for a given line $\ell : x = c$, we wish to find a point $\mathbf{y}^* = (c, y^*) \in \ell$ that minimizes F^{Δ}. We execute the query procedure A_s without knowing the value $\mathbf{y}^*$. This procedure makes $O(\log n)$ comparisons, each of which involves computing the sign of a univariate, constant-degree polynomial $g(y)$ in the y-coordinate y^* of $\mathbf{y}^*$. We compute the roots $y_1, \ldots, y_p$ of g, and evaluate $F^{\Delta}(c, y_i)$ for $1 \leq i \leq p$. Set $y_0 = -\infty$ and $y_{p+1} = +\infty$. Since F^{Δ} is convex, by examining the local behavior of $F^{\Delta}(c, \cdot)$ at each y_i (using an appropriate straightforward extension of the preceding procedure), we can determine whether $\mathbf{y}^* = (c, y_i)$ for some $0 \leq i \leq p+1$; if not, we can determine the open interval (y_i, y_{i+1}) that contains y^*. We can thus compute the sign of g at $\mathbf{y}^*$. Proceeding in this manner, we can compute the value of $\mathbf{y}^*$. Since F^{Δ} is convex over the entire xy-plane (or rather over the convex 2-dimensional domain $D(S)$), the above procedure can be extended to determine whether the global minimum $\mathbf{x}^*$ of F^{Δ} lies to the left of ℓ, to the right of ℓ, or on ℓ, simply by examining the local behavior of F^{Δ} in the neighborhood of $\mathbf{y}^*$. In the third case, the procedure can return the value of $\mathbf{x}^* = \mathbf{y}^*$ and terminate. The total running time of this procedure is $O(\log^2 n)$, because we spend $O(\log n)$ time at each comparison and the algorithm makes $O(\log n)$ comparisons.

Using the above 1-dimensional procedure as a subroutine, we can compute $\mathbf{x}^*$ as follows. We now run the query procedure of Theorem 6.1 in a generic manner at $\mathbf{x}^*$. Each comparison now involves computing the sign of a bivariate, constant-degree polynomial $\pi(\mathbf{x})$. Let $\vee\pi$ denote the set of roots of π. We compute Collins's cylindrical algebraic decomposition Π of $\mathbb{R}^2$ so that the sign of π is invariant within each cell of Π [12, 26, 61]. Our aim is to determine the cell $\tau \in \Pi$ that contains $\mathbf{x}^*$, thereby determining the sign of π at $\mathbf{x}^*$.

The cells of Π are delimited by $O(1)$ y-vertical lines—each passing through a self-intersection point of $\vee\pi$ or through a point of vertical tangency of $\vee\pi$; see Figure 8.1. For each vertical line ℓ, we run the standard 1-dimensional parametric-searching procedure to determine which side of ℓ contains $\mathbf{x}^*$. If any of these substeps returns $\mathbf{x}^*$, we are done. Otherwise, we obtain a vertical strip σ that contains $\mathbf{x}^*$. We still have to search through the cells of Π within σ, which are stacked one above the other in the y-direction, to determine which of them contains $\mathbf{x}^*$. We note that the number of roots of π along any vertical line $\ell : x = x_0$ within σ is the same, that each root varies continuously with x_0, and that their relative y-order is the same for each vertical line. In other words, the roots of $\vee\pi$ in σ constitute a collection of disjoint, x-monotone arcs $\gamma_1, \ldots, \gamma_t$ whose endpoints lie on the boundary lines of σ. We can regard each γ_i

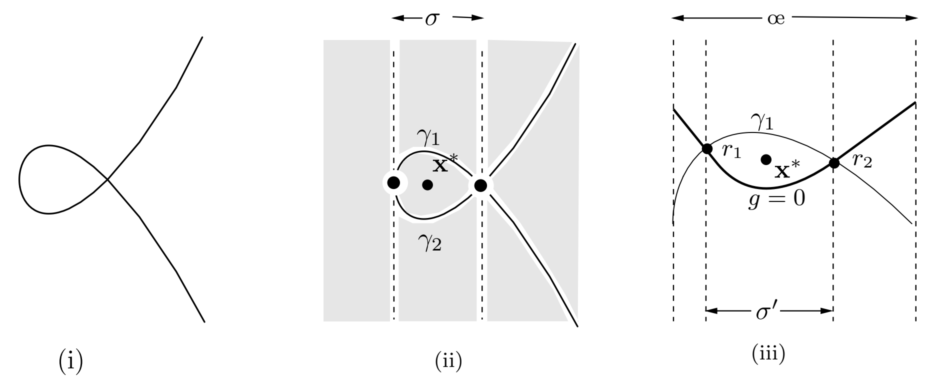


FIG. 8.1. (i) *The set* $\vee\pi$ *of roots of* π. (ii) *The cylindrical algebraic decomposition of* π. (iii) *The curves* $g = 0$ *and* γ_1.

as the graph of a univariate function $\gamma_i(x)$.

Next, for each γ_i we determine whether $\mathbf{x}^*$ lies below, above, or on γ_i. Let x^* be the (unknown) x-coordinate of $\mathbf{x}^*$, and let ℓ be the vertical line $x = x^*$. If we knew x^*, we could have run A_s at each $\gamma_i \cap \ell$ and could have located $\mathbf{x}^*$ with respect to γ_i, as desired. Since we do not know x^*, we execute the 1-dimensional procedure, described above, generically, on the line ℓ, with the intention of simulating it at the unknown point $\mathbf{x}_i = \gamma_i \cap \ell$. This time, performing a comparison involves computing the sign of some bivariate, constant-degree polynomial g at $\mathbf{x}_i$ (we prefer to treat g as a bivariate polynomial, although we could have eliminated one variable by restricting $\mathbf{x}$ to lie on γ_i). We compute the roots $r_1, \ldots, r_u$ of g that lie on γ_i, and set r_0 and r_{u+1} to be the left and right endpoints of γ_i, respectively. As above, we compute the index j so that $\mathbf{x}^*$ lies in the vertical strip σ' bounded between r_j and r_{j+1}. Notice that the sign of g is the same for all points on γ_i within the strip σ', so we can now compute the sign of g at $\mathbf{x}_i$.

When the generic algorithm being simulated at $\mathbf{x}_i$ terminates, it returns a constant-degree polynomial $F_i(x, y)$, corresponding to the value of F^Δ at $\mathbf{x}_i$ (i.e., $F_i(\mathbf{x}_i) = F^\Delta(\mathbf{x}_i)$), and a vertical strip $\sigma_i \subseteq \sigma$ that contains $\mathbf{x}^*$. Let $\rho_i(x) = F_i(x, \gamma_i(x))$. Let γ_i^+ (resp., γ_i^-) be the copy of γ_i translated by an infinitesimally small amount in the $(+y)$-direction (resp., $(-y)$-direction), i.e., $\gamma_i^+(x) = \gamma_i(x) + \varepsilon$ (resp., $\gamma_i^-(x) = \gamma_i(x) - \varepsilon$), where $\varepsilon > 0$ is an infinitesimal. We next simulate the algorithm at $\mathbf{x}_i^+ = \gamma_i^+ \cap \ell$ and $\mathbf{x}_i^- = \gamma_i^- \cap \ell$. We thus obtain two functions $\rho_i^+(x)$, $\rho_i^-(x)$ and two vertical strips σ_i^+, σ_i^-. Let $\hat{\sigma}_i = \sigma_i \cap \sigma_i^+ \cap \sigma_i^-$. We need to evaluate the signs of $\rho_i(x^*) - \rho_i^+(x^*)$ and $\rho_i(x^*) - \rho_i^-(x^*)$ to determine the location of $\mathbf{x}^*$ with respect to γ_i (this is justified by the convexity of F^Δ). We compute the x-coordinates of the intersection points of (the graphs of) $\rho_i, \rho_i^+, \rho_i^-$ that lie inside $\hat{\sigma}_i$. Let $x_1 \le x_2 \le \cdots \le x_s$ be these x-coordinates, and let x_0, x_{s+1} be the x-coordinates of the left and right boundaries of $\hat{\sigma}_i$, respectively. By running A_s on the vertical lines $x = x_j$, for $1 \le j \le s$, we determine the index $0 \le j \le s$ so that the vertical strip $\sigma_i = [x_j, x_{j+1}] \times \mathbb{R}$ contains $\mathbf{x}^*$. Notice that the signs of polynomials $\rho_i(x) - \rho_i^+(x), \rho_i(x) - \rho_i^-(x)$ are fixed for all $x \in [x_j, x_{j+1}]$. By evaluating $\rho_i, \rho_i^+, \rho_i^-$ for any $x_0 \in [x_j, x_{j+1}]$, we can compute the signs of $\rho_i(x^*) - \rho_i^+(x^*)$ and of $\rho_i(x^*) - \rho_i^-(x^*)$.

Repeating this procedure for all γ_i's, we can determine the cell of Π that contains $\mathbf{x}^*$ and thus resolve the comparison involving π. We then resume the execution of the generic algorithm.

The execution of the 1-dimensional procedure takes $O(\log^2 n)$ steps, which implies that its generic simulation requires $O(\log^3 n)$ time. The total time spent in resolving the sign of π at $\mathbf{x}^*$ is therefore $O(\log^3 n)$. Hence, the total running time of the 2-dimensional algorithm is $O(\log^4 n)$.

THEOREM 8.1. *The intersection of a set of congruent balls in $\mathbb{R}^3$ can be maintained dynamically, by a data structure of size $O(n^{1+\varepsilon})$, so that each insertion or deletion of a ball takes $O(n^\varepsilon)$ time and the following queries can be answered:* (a) *For any query point p, we can determine in $O(\log n)$ time whether p lies in the current intersection, and* (b) *after performing each update, we can determine in $O(\log^4 n)$ time whether the current intersection is nonempty.*

Remark 8.2. (i) The same data structure can be used to answer other queries, such as determining whether a line intersects $K(S)$, computing the highest (or the lowest) vertex of $K(S)$, etc.

(ii) The same technique can also be used to maintain the intersection of other simply-shaped, convex objects in $\mathbb{R}^3$. The technique will be most effective in cases where the complexity of such an intersection can be shown to be small.

(iii) The problem studied in this section arises in the 3-dimensional 2-*center* problem, where we are given a set S of n points in $\mathbb{R}^3$ and wish to cover them by the union of two congruent balls whose radius is as small as possible. Extending to three dimensions the standard approach to this problem [29], we face a main subproblem in which we need to maintain dynamically the intersection of a set of congruent balls and to determine after each update whether this intersection is nonempty. The main difference, though, is that in the 2-center problem the sequence of updates is known in advance, which makes the problem easier to solve. Still, using the method of this section, we obtain an $O(n^{3+\varepsilon})$-time algorithm for the 3-dimensional 2-center problem.

9. Smallest stabbing disk.

Let $\mathbf{C}$ be a (possibly infinite) family of simply shaped compact strictly-convex sets (called *objects*) in the plane. By "simply shaped" we mean that each object is described by a Boolean combination of a constant number of polynomial equalities and inequalities of constant maximum degree. Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be a finite subset of $\mathbf{C}$. We wish to update $\mathcal{C}$ dynamically, by inserting objects $c \in \mathbf{C}$ into $\mathcal{C}$ or by deleting such objects from $\mathcal{C}$, and maintain a smallest disk or, more generally, a smallest homothetic copy of some given simply shaped compact convex set P that intersects all sets of $\mathcal{C}$. This study extends recent work by Agarwal and Matoušek [6], who have obtained an algorithm that takes $O(n^\varepsilon)$ time per update operation for the case where $\mathcal{C}$ is a set of points and P is a disk.

9.1. Farthest-neighbor Voronoi diagrams.

The set P induces a *convex distance function* defined by

$$d_P(x, y) = \min \{\lambda \mid y \in x + \lambda P\}.$$

We assume here that P contains the origin in its interior. The function d_P is a metric if and only if P is centrally symmetric with respect to the origin. We define the farthest-neighbor Voronoi diagram $\mathit{Vor}_P(\mathcal{C})$ of $\mathcal{C}$, under the distance function d_P, in a standard manner (see [48, 56, 63] for details). In what follows, we assume that P is strictly convex; the results can be extended to more general sets with some extra care. We first prove the following theorem, which is of independent interest. (We are not aware of any previous proof of this result.)

THEOREM 9.1. *Let $\mathcal{C}$ be a set of n (possibly intersecting) simply shaped compact convex objects in the plane. Then the complexity of the farthest-neighbor Voronoi*

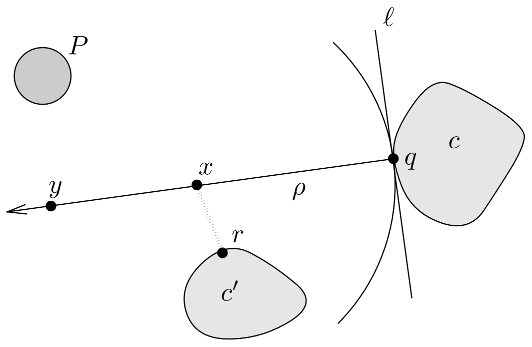


FIG. 9.1. *Anti-star-shape property of Voronoi cells.*

diagram of $\mathcal{C}$, under a convex distance function d_P induced by any simply shaped compact strictly convex set P, is $O(\lambda_s(n))$. Here s is a constant depending on the shape of P and of the objects in $\mathcal{C}$. If $\mathcal{C}$ is a set of n line segments and d_P is the Euclidean distance function (i.e., P is a disk), the complexity of the farthest-neighbor Voronoi diagram is $O(n)$.

Proof. Let R be one of the Voronoi cells of $Vor_P(\mathcal{C})$, and let $c \in \mathcal{C}$ be the farthest neighbor of all points of R. We show that R has the following "anti-star-shape" property: Let $x \in R$, and let $q \in c$ be the nearest point to x (i.e., $d_P(x,q) = d_P(x,c)$); the convexity of c and the strict convexity of P imply that q is unique. Let ρ be the ray emanating from q towards x, and let y be any point on ρ past x (i.e., x lies in the segment qy). We claim that c is the farthest neighbor of y; see Figure 9.1. Indeed, suppose to the contrary that the farthest neighbor of y in $\mathcal{C}$ is $c' \neq c$ (so that $d_P(y,c') > d_P(y,c)$), and let $r \in c'$ be the nearest point to x. Let ℓ be the (unique) line passing through q and tangent to $x + d_P(x,c)P$ there. The convexity of c implies that c is contained in the (closed) halfplane bounded by ℓ and not containing x. Since ℓ is also tangent to $y + d_P(y,q)P$, it easily follows that $d_P(y,c) = d_P(y,q)$. Now, by the triangle inequality we have

$$\begin{aligned} d_P(y,c') &\leq d_P(y,r) \\ &\leq d_P(y,x) + d_P(x,r) \\ &= d_P(y,x) + d_P(x,c') \\ &\leq d_P(y,x) + d_P(x,c) \\ &= d_P(y,x) + d_P(x,q) \\ &= d_P(y,q) \\ &= d_P(y,c), \end{aligned}$$

a contradiction that establishes the claim.

This implies that R is unbounded, and so $Vor_P(\mathcal{C})$ is an outerplanar map. It is easy to verify that every vertex of $Vor_P(\mathcal{C})$ has degree at least three. Hence, by Euler's formula for planar graphs, the complexity of $Vor_P(\mathcal{C})$ is proportional to the number of its faces. We bound the number of faces in $Vor_P(\mathcal{C})$, as follows. By a compactness argument, there exists a sufficiently large r such that the circle σ_r of radius r about the origin cuts the cells of $Vor_P(\mathcal{C})$ in the same sequence as does the circle at infinity. For each $c \in \mathcal{C}$, let $f_c(\theta) = d_P((r,\theta),c)$, and let $F(\theta) = \max_{c\in\mathcal{C}} f_c(\theta)$. Clearly, the number of edges of $Vor_P(\mathcal{C})$ crossed by σ_r, which is an upper bound on the number of faces

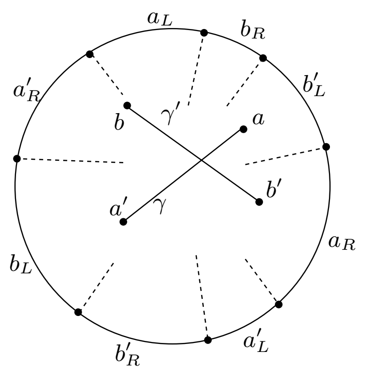


FIG. 9.2. *Partition of the circle σ_r into maximal arcs.*

in $Vor_P(\mathcal{C})$, is equal to the number of breakpoints in the upper envelope F. Since we have assumed P and each set $c \in \mathcal{C}$ to be simply shaped, it follows that for each pair $c, c' \in \mathcal{C}$, the functions f_c and $f_{c'}$ have at most some constant number, s, of intersection points. Hence, the number of breakpoints of F is, by standard Davenport–Schinzel theory [39, 66], at most $\lambda_s(n)$. This establishes the first part of the claim.

For the case of line segments and Euclidean distance, the bound improves to $O(n)$. To see this, let $\mathcal{C}$ be a set of n line segments in the plane. Traverse the circle σ_r, as defined above, and decompose it into maximal arcs so that the following conditions are satisfied for each arc ζ:

(i) The farthest segment from all points $q \in \zeta$ is the same segment e,
(ii) the nearest endpoint of e from q is fixed, and
(iii) ζ lies fully in one of the halfplanes bounded by the line containing e.

We label each arc ζ by the endpoint of the corresponding segment e, but we use different labels for arcs that lie on different "sides" of e (as in (iii) above); see Figure 9.1. We call an arc ζ, labeled by endpoint a, a *left arc* (and label it by a_L) if it lies on the left side of the segment incident to a when this segment is directed towards a; otherwise, ζ is a *right arc* (and we label it a_R). Let S (resp., S_R, S_L) denote the cyclic sequence consisting of the labels of all (resp., all the right, all the left) arcs in the clockwise order along σ_r. Note that S is obtained by merging S_L and S_R and it does not contain any pair of equal adjacent elements (S_L and S_R may contain such pairs). Each of S_L and S_R is composed of at most $2n$ symbols. We claim that S_L cannot contain an alternating subcycle of the form $\langle a_L \cdots b_L \cdots a_L \cdots b_L \rangle$, where a and b are two segment endpoints.

In order to prove this claim, it suffices to rule out the existence of such a subcycle in the partition of σ_r induced by the one or two segments incident to a and b. First, such a subcycle is impossible if a and b are endpoints of the same segment e because all arcs labeled by a are separated from all arcs labeled by b by the perpendicular bisector of e.

Suppose then that a and b are endpoints of two distinct respective segments e_1, e_2. Let a' be the other endpoint of e_1, and let b' be the other endpoint of e_2. Since r is sufficiently large, the perpendicular bisector of any two endpoints of segments in $\mathcal{C}$ partitions σ_r into two arcs, each of which is larger than, say, $3\pi r/4$. Both appearances of a (resp., of b) in the assumed subcycle occur within an arc γ_1 (resp., γ_2), bounded by the perpendicular bisector e_1 (resp., e_2) and by the line containing e_1 (resp., e_2). See Figure 9.3. An easy calculation shows that the lengths of γ_1, γ_2 are both less than

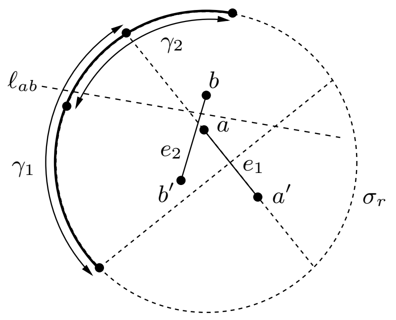


FIG. 9.3. *Segments e_1, e_2 and arcs γ_1, γ_2.*

$3\pi r/4$ (each of the arcs is approximately a quarter circle, so the above property will hold if r is chosen sufficiently large), and, by definition, a is the nearest endpoint of e_1 within γ_1 and b is the nearest endpoint of e_2 within γ_2. By assumption, γ_1 contains three disjoint subarcs labeled a_L, b_L, a_L in the clockwise order.

This, however,+ implies that the perpendicular bisector, ℓ_{ab}, of a and b intersects γ_1 twice, which is impossible because the length of γ_1 is less than $3\pi r/4$ and the length of each of the two arcs into which σ_r is partitioned by ℓ_{ab} is more than $3\pi r/4$. This completes the proof of the claim.

A symmetric argument shows that S_R too cannot contain such a subcycle. Hence, if we erase from S_L and from S_R every element equal to its predecessor, we obtain two (cyclic) Davenport–Schinzel sequences of order 2, each composed of at most $2n$ symbols, which are thus of length at most $4n-2$ each. It is now a fairly standard exercise to show that the total length of S is also linear in n (see, e.g., [66]). This completes the proof. $\square$

For each object $c_i \in \mathcal{C}$, define a bivariate function

$$f_i(\mathbf{x}) = -\min_{q \in c_i} d_P(\mathbf{x}, q)$$

for $\mathbf{x} \in \mathbb{R}^2$. Let $\mathcal{F} = \{f_i \mid 1 \le i \le n\}$. The minimization diagram of $\mathcal{F}$ is the farthest-neighbor Voronoi diagram of $\mathcal{C}$ under the distance function d_P. Following the same argument as in Theorem 6.1 and using Theorem 9.1, we obtain the following corollary.

COROLLARY 9.2. *Let $\mathcal{C}$ be a set of n (possibly intersecting) simply shaped compact convex sets in the plane and d_P be a convex distance function as above. $\mathcal{C}$ can be preprocessed into a farthest-neighbor-searching data structure of size $O(n^{1+\varepsilon})$ so that a query can be answered in $O(\log n)$ time and an object (of the same form) can be inserted into or deleted from $\mathcal{C}$ in $O(n^\varepsilon)$ time.*

9.2. Maintaining the smallest stabbing disk. Returning to the smallest stabbing disk problem, we need to compute the highest point w^* on the lower envelope $E_{\mathcal{F}}$ of $\mathcal{F}$. In view of Corollary 9.2 and the easily established fact that the cell of $\mathcal{A}(\mathcal{F})$ lying below $E_{\mathcal{F}}$ is convex, a natural approach to computing w^* is to use the multidimensional parametric-searching technique of Toledo [69] in a manner similar to that described in the preceding section. Omitting further details, which can be worked out by the reader, we have the next theorem.

THEOREM 9.3. *A set $\mathcal{C}$ of n (possibly intersecting) simply shaped compact convex sets in the plane can be stored in a data structure of size $O(n^{1+\varepsilon})$ so that a smallest* *homothetic placement of some given simply shaped compact strictly convex set P that intersects all objects of $\mathcal{C}$ can be computed in $O(n^{\varepsilon})$ time, after each insertion or deletion of a set into/from $\mathcal{C}$.*

Remark 9.4. A related problem is that of maintaining the smallest disk (or homothetic copy of some given P) *enclosing* all but at most k objects of a dynamically changing collection $\mathcal{C}$. The same technique as above can be used, except that now we need to replace $Vor_P(\mathcal{C})$ by a different farthest-neighbor diagram, where the d_P-distance to an object c is the maximum distance to a point of c. See [68] for a recent study of such Voronoi diagrams.

10. Other applications.

In this section, we list some other applications of the techniques developed in this paper. To keep the length of the paper under control, we omit most of the details concerning these applications; some of these details are fairly routine, but some are more technical and problem-specific. Some of the details can be found in [32]. We are confident that the new techniques will also find many additional applications.

10.1. Smallest stabbing disk with at most *k* violations.

Let $\mathcal{C}$ and P be as in section 9, and let $1 \leq k \leq n$ be an integer parameter. We consider the problem of finding a smallest homothetic placement of P that stabs at least $n-k$ of the objects of $\mathcal{C}$. This problem extends the problem studied by Matoušek [51], in which one seeks the smallest disk stabbing all but at most k points of a given set of points in the plane.

Let $\mathcal{F} = \{f_i \mid 1 \leq i \leq n\}$ be the set of bivariate functions as defined in section 9. As easily seen, our goal is to find a point in $\mathcal{A}_{\leq k}(\mathcal{F})$ with the maximum z-coordinate. In view of Corollary 9.2, this can be done in time $O(n^{1+\varepsilon}k^2)$ by explicitly computing $\mathcal{A}_{\leq k}(\mathcal{F})$, as in Theorem 5.1. However, one can do much better. Note that the problem

$$\max \quad z \quad \text{subject to}$$
$$z \leq f_i(x,y), \qquad i = 1,\ldots,n,$$

is an LP-type problem (see [54, 67]). Matoušek [51, Theorem 2.2] showed that the number of local maxima in the first k levels of $\mathcal{A}(\mathcal{F})$ is $O((k+1)^3)$. Note that each local maxima w of the jth level lies on at most three function graphs and is the unique maximum of the lower envelope of a subcollection $\mathcal{F}'$ of $n-j$ functions of $\mathcal{F}$. Moreover, there exists a local maximum w' of the $(j-1)$st level that "hides" w in the sense that w' is the unique maximum of the lower envelope of $\mathcal{F}' \cup \{f\}$, where f is one of the functions whose graph contains w'.

Matoušek also proposed a method for computing all these maxima, which proceeds in a depth-first-search fashion, starting from the unique maximum p in $\mathcal{A}_{\leq 0}(\mathcal{F})$ (i.e., the lower envelope of $\mathcal{F}$). Suppose the algorithm is currently at a local maxima q of the jth level for some $j \leq k$; q is the unique maximum of the lower envelope of a subset $\mathcal{F}' \subseteq \mathcal{F}$ of $n-j$ functions. If $j < k$, the algorithm removes one of the (at most) three constraints defining q, say, f, and finds the maximum q' of the LP-type problem defined by the constraints in $\mathcal{F}' \setminus \{f\}$, using an appropriate dynamic data structure. Applying this step repeatedly, we find all local maxima violating at most k constraints. The cost of this algorithm is proportional to the preprocessing cost for constructing the data structure plus the number of maxima (i.e., $O((k+1)^3)$) times the cost of a query and an update of the structure.

In our setting, a point q in $\mathbb{R}^3$ represents a homothetic copy of P that misses exactly k objects of $\mathcal{C}$ if and only if it lies at the kth level of $\mathcal{A}(\mathcal{F})$. Hence, to implement Matoušek's technique for our problem, we first construct the data structure of Theorem 9.3 in time $O(n^{1+\varepsilon})$. This also yields a smallest homothetic copy of P that stabs all the objects of $\mathcal{C}$. Then we compute each of the $O((k+1)^3)$ maxima of $\mathcal{A}_{\le k}(\mathcal{F})$, using the above technique. Each step of the algorithm starts at some local minimum q on one of the first k levels so that the data structure currently represents all the objects of $\mathcal{C}$ not violating q. It then deletes one of the (at most 3) objects defining q and queries the structure to obtain the minimum for the resulting set of remaining constraints (as in Theorem 9.3). The cost of such a query is $O(n^{\varepsilon})$. We thus obtain the following theorem.

THEOREM 10.1. *Let $\mathcal{C}$ be a family of n (possibly intersecting) simply shaped compact convex sets in the plane, and let $0 \le k \le n$ be a parameter. We can find a smallest homothetic placement of some given simply shaped, compact, strictly convex set P, which stabs all except at most k of the sets of $\mathcal{C}$, in time $O(n^{1+\varepsilon} + k^3 n^{\varepsilon})$.*

Remark 10.2. Note that this approach can be generalized to many other optimization problems, where the optimizing object has three degrees of freedom and we seek an optimum solution that violates at most k of the given constraints. The machinery developed in this paper is powerful enough to allow such extensions, requiring only that the lower envelope of any subcollection of objects is a convex surface and that the relevant constraints satisfy some rather mild conditions.

10.2. Segment center with at most k violations. Let S be a set of n points in the plane, and let e be a fixed-length segment. A placement e^* of e is called a *segment center* of S if the maximum distance between the points of S and e^* is minimized. We call e^* a *segment center* of S *with k violations* if the $(k+1)$st largest distance between e^* and the points of S is minimized. The problem of computing the segment center (without violations) has been studied in [3, 34, 42]; the best algorithm, given in [34], computes the segment center in $O(n^{1+\varepsilon})$ time.

Applying the machinery developed in section 5 and extending the result of [34], one can obtain the following result.

THEOREM 10.3. *The segment center with k violations of a set S of n points in the plane can be computed in time $O(n^{1+\varepsilon}k^2)$.*

Here is a brief sketch of the proof. We first solve the *fixed-size problem*: Given a real $d > 0$, determine whether there exists a placement of e for which the $(k+1)$st largest distance to the points of S is $\le d$. As follows from the analysis of [34], this is equivalent to the problem of determining whether there exists a (translated and rotated) placement Z of the *hippodrome* $H(e,d)$, defined as the Minkowski sum of e and a (closed) disk of radius d, that contains all but at most k points of S.

Extending the technique of [34], one can reduce this latter problem to the following one: We are given two collections $\mathcal{F}$ and $\mathcal{G}$ of n partially defined fixed-degree algebraic bivariate functions, and we wish to determine whether there exists an intersection between the first k lower levels of $\mathcal{A}(\mathcal{F})$ and the first k upper levels of $\mathcal{A}(\mathcal{G})$ so that the sum of the levels of such an intersection does not exceed k. This can be solved efficiently using the machinery developed in section 5. As shown in [34], we have $\psi(n) = O(n \log n)$ for the collections $\mathcal{F}$ and $\mathcal{G}$, which then implies that the asymptotic running time of the algorithm is $O(n^{1+\varepsilon}k^2)$.

Once the fixed-size problem is solved, the segment center problem can be solved by parallelizing the decision algorithm under Valiant's model [71] and by applying the parametric searching technique of [55]. The bound on the running time remains asymptotically the same.

11. Conclusions. In this paper we have extended known range-searching techniques to arrangements of low-degree algebraic bivariate functions in $\mathbb{R}^3$. By establishing sharp bounds on the vertical decomposition of the first k levels of such an arrangement, we were able to extend the construction of [52] of shallow cuttings to such arrangements. This in turn has yielded a collection of static and dynamic range-searching techniques for problems that involve such arrangements, from which we have obtained new and efficient solutions to several geometric optimization problems, including minimum weight Euclidean bipartite matching.

The main open problem that the paper raises is to extend our results to higher dimensions. The first obstacle that we face here is the lack of really sharp bounds on the complexity of vertical decompositions in arrangements of surfaces, even for the whole arrangement, which prevents our technique from "taking off" at all. Recently Schwarzkopf and Sharir [62] have shown that the complexity of the vertical decomposition of a single cell in an arrangement of surfaces in $\mathbb{R}^3$ is $O(n^{2+\varepsilon})$. It is an open problem whether these techniques can be extended to bound the complexity of the vertical decomposition of the lower envelope of trivariate functions.

Another open problem is to extend our results to other cases involving shallow levels in 3-dimensional arrangements. For example, we may want to maintain dynamically a single cell in an arrangement of surfaces in $\mathbb{R}^3$ or the union or intersection of a collection of 3-dimensional objects, etc. Extending the analysis of section 2 to these situations seems considerably more difficult.

REFERENCES

[1] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf, *Constructing levels in arrangements and higher order Voronoi diagrams*, SIAM J. Comput., 27 (1998), pp. 654–667.

[2] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl, *Euclidean minimum spanning trees and bichromatic closest pairs*, Discrete Comput. Geom., 6 (1991), pp. 407–422.

[3] P. K. Agarwal, A. Efrat, M. Sharir, and S. Toledo, *Computing a segment-center for a planar point set*, J. Algorithms, 15 (1993), pp. 314–323.

[4] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.

[5] P. K. Agarwal and J. Matoušek, *Range searching with semialgebraic sets*, Discrete Comput. Geom., 11 (1994), pp. 393–418.

[6] P. K. Agarwal and J. Matoušek, *Dynamic half-space range searching and its applications*, Algorithmica, 14 (1995), pp. 325–345.

[7] P. K. Agarwal, J. Matoušek, and O. Schwarzkopf, *Computing many faces in arrangements of lines and segments*, SIAM J. Comput., 27 (1998), pp. 491–505.

[8] P. K. Agarwal, O. Schwarzkopf, and M. Sharir, *Overlay of lower envelopes in three dimensions and its applications*, Discrete Comput. Geom., 15 (1996), pp. 1–13.

[9] P. K. Agarwal, M. Sharir, and P. Shor, *Sharp upper and lower bounds for the length of general Davenport Schinzel sequences*, J. Combin. Theory Ser. A, 52 (1989), pp. 228–274.

[10] A. Aggarwal, A. Bar-Noy, S. Khuller, D. Kravets, and B. Schieber, *Efficient minimum cost matching using quadrangle inequality*, J. Algorithms, 19 (1995), pp. 116–143.

[11] A. Aggarwal, M. Hansen, and T. Leighton, *Solving query-retrieval problems by compacting Voronoi diagrams*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 331–340.

[12] D. S. Arnon, G. E. Collins, and S. McCallum, *Cylindrical algebraic decomposition* I: *The basic algorithm*, SIAM J. Comput., 13 (1984), pp. 865–877.

[13] D. Atkinson and P. Vaidya, *Using geometry to solve the transportation problem in the plane*, Algorithmica, 13 (1995), pp. 442–461.

[14] F. Aurenhammer, F. Hoffmann, and B. Aronov, *Minkowski-type theorems and least square partitioning*, in Proc. 8th Annual Symposium on Computational Geometry, 1992, pp. 350–357.

[15] J. Bentley and J. Saxe, *Decomposable searching problems* I: *Static-to-dynamic transformation*, J. Algorithms, 1 (1980), pp. 301–358.

[16] H. Brönnimann, B. Chazelle, and J. Matoušek, *Product range spaces, sensitive sampling, and derandomization*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, 1993, pp. 400–409.

[17] S. P. Buss and P. N. Yianilos, *Linear and $O(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours*, SIAM J. Comput., 27 (1998), pp. 170–201.

[18] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *A singly exponential stratification scheme for real semialgebraic varieties and its applications*, Theoret. Comput. Sci., 84 (1991), pp. 77–105.

[19] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *Diameter, width, closest line pair, and parametric searching*, Discrete Comput. Geom., 10 (1993), pp. 183–196.

[20] B. Chazelle and J. Friedman, *A deterministic view of random sampling and its use in geometry*, Combinatorica, 10 (1990), pp. 229–249.

[21] B. Chazelle and F. P. Preparata, *Halfspace range searching: An algorithmic application of k-sets*, Discrete Comput. Geom., 1 (1986), pp. 83–93.

[22] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Algorithmica, 8 (1992), pp. 407–430.

[23] K. Clarkson, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[24] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl, *Combinatorial complexity bounds for arrangements of curves and spheres*, Discrete Comput. Geom., 5 (1990), pp. 99–160.

[25] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry*, II, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[26] G. Collins, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, in Second GI Conf. on Automata Theory and Formal Languages, Lecture Notes in Comput. Sci. 33, H. Barkhage, ed., Springer-Verlag, New York, Berlin, Heidelberg, 1975, pp. 134–183.

[27] M. de Berg, K. Dobrindt, and O. Schwarzkopf, *On lazy randomized incremental construction*, Discrete Comput. Geom., 14 (1995), pp. 261–286.

[28] M. de Berg, L. Guibas, and D. Halperin, *Vertical decomposition for triangles in* 3*-space*, Discrete Comput. Geom., 15 (1996), pp. 35–61.

[29] Z. Drezner, *The planar two-center and two-median problems*, Transportation Science, 18 (1984), pp. 351–361.

[30] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[31] H. Edelsbrunner and R. Seidel, *Voronoi diagrams and arrangements*, Discrete Comput. Geom., 1 (1986), pp. 25-44.

[32] A. Efrat, *Geometric Location Optimization Problems*, Ph.D. dissertation, Tel Aviv University, Tel Aviv, Israel, 1998.

[33] A. Efrat and A. Itai, *Improvements on bottleneck matching and related problems using geometry*, in Proc. 12th Annual Symposium on Computational Geometry, 1996, pp. 301–310.

[34] A. Efrat and M. Sharir, *A near-linear algorithm for the planar segment center problem*, Discrete Comput. Geom., 16 (1996), pp. 239–257.

[35] D. Eppstein, *Dynamic Euclidean minimum spanning trees and extrema of binary functions*, Discrete Comput. Geom., 13 (1995), pp. 111–122.

[36] B. Grünbaum, *A proof of Vázsonyi's conjecture*, Bull. Research Council Israel Sec. A, 6 (1956), pp. 77–78.

[37] L. Guibas, D. Halperin, J. Matoušek, and M. Sharir, *Vertical decomposition of arrangements of hyperplanes in four dimensions*, Discrete Comput. Geom., 14 (1995), pp. 113-122.

[38] D. Halperin and M. Sharir, *New bounds for lower envelopes in three dimensions, with applications to visibility of terrains*, Discrete Comput. Geom., 12 (1994), pp. 313–326.

[39] S. Hart and M. Sharir, *Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151–177.

[40] D. Haussler and E. Welzl, *ϵ-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–151.

[41] J. Holm, K. de Lichtenberg, and M. Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree,* 2*-edge, and biconnectivity*, in Proc. 30th Annual ACM Symposium on Theory of Computing, 1998, pp. 79–89.

[42] H. Imai, D. T. Lee, and C. Yang, 1*-segment center covering problems*, ORSA J. Comput., 4 (1992), pp. 426–434.

[43] H. Kuhn, *The Hungarian method for the assignment problem*, Naval Research Logistics Quar-

terly, 2 (1955), pp. 83–97.
[44] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Saunders College Publishing, Fort Worth, TX, 1976.
[45] D. T. Lee, *Two-dimensional Voronoi diagrams in the L_p-metric*, J. Assoc. Comput. Mach., 27 (1980), pp. 604–618.
[46] D. T. Lee and R. L. Drysdale, III, *Generalization of Voronoi diagrams in the plane*, SIAM J. Comput., 10 (1981), pp. 73–87.
[47] D. Leven and M. Sharir, *Intersection and proximity problems and Voronoi diagrams*, in Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics, J. T. Schwartz and C.-K. Yap, eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 187–228.
[48] D. Leven and M. Sharir, *Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams*, Discrete Comput. Geom., 2 (1987), pp. 9–31.
[49] O. Marcotte and S. Suri, *Fast matching algorithms for points on a polygon*, SIAM J. Comput., 20 (1991), pp. 405–422.
[50] J. Matoušek, *Approximations and optimal geometric divide-and-conquer*, J. Comput. Systems Sci., 50 (1995), pp. 203–208.
[51] J. Matoušek, *On geometric optimization with few violated constraints*, Discrete Comput. Geom., 14 (1995), pp. 365–384.
[52] J. Matoušek, *Reporting points in halfspaces*, Comput. Geom., 2 (1992), pp. 169–186.
[53] J. Matoušek and O. Schwarzkopf, *On ray shooting in convex polytopes*, Discrete Comput. Geom., 10 (1993), pp. 215–232.
[54] J. Matoušek, M. Sharir, and E. Welzl, *A subexponential bound for linear programming and related problems*, Algorithmica, 16 (1996), pp. 498–516.
[55] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.
[56] K. Mehlhorn, S. Meiser, and R. Rasch, *Furthest Site Abstract Voronoi Diagrams*, Tech. report, Max-Planck Institut für Informatik, Saarbrücken, 1992.
[57] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, 1995.
[58] K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice–Hall, Englewood Cliffs, NJ, 1993.
[59] K. Mulmuley, *On levels in arrangements and Voronoi diagrams*, Discrete Comput. Geom., 6 (1991), pp. 307–338.
[60] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Comput. Sci. 156, Springer-Verlag, Berlin, 1983.
[61] J. Schwartz and M. Sharir, *On the "piano movers" problem* II: *General technique for computing topological properties of real algebraic manifolds*, Adv. in Appl. Math., 4 (1983), pp. 298–351.
[62] O. Schwarzkopf and M. Sharir, *Vertical decomposition of a single cell in a three-dimensional arrangement of surfaces and its applications*, Discrete Comput. Geom., 18 (1997), pp. 269–288.
[63] M. Sharir, *Intersection and closest-pair problems for a set of planar discs*, SIAM J. Comput., 14 (1985), pp. 448–468.
[64] M. Sharir, *On k-sets in arrangements of curves and surfaces*, Discrete Comput. Geom., 6 (1991), pp. 593–613.
[65] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions*, Discrete Comput. Geom., 12 (1994), pp. 327–345.
[66] M. Sharir and P. K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.
[67] M. Sharir and E. Welzl, *A combinatorial bound for linear programming and related problems*, in Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science, 1992, pp. 569–579.
[68] M. Sharir and E. Welzl, *Circular and Spherical Separability*, manuscript.
[69] S. Toledo, *Maximizing non-linear concave functions in fixed dimensions*, in Complexity in Numerical Computations, P. M. Pardalos, ed., World Scientific, Singapore, 1993, pp. 429–447.
[70] P. M. Vaidya, *Geometry helps in matching*, SIAM J. Comput., 18 (1989), pp. 1201–1225.
[71] L. G. Valiant, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.
[72] K. R. Varadarajan, *A divide-and-conquer algorithm for min-cost perfect matching in the plane*, in Proc. 39th Annual IEEE Symposium on Foundations of Computer Science, 1998, pp. 320–329.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 954–986

A $2\frac{1}{2}$-APPROXIMATION ALGORITHM FOR SHORTEST SUPERSTRING*

Z. SWEEDYK†

Abstract. Given a set of strings $S = \{s_1, s_2, \ldots, s_n\}$ over a finite alphabet Σ, a *superstring* of S is a string that contains each s_i as a contiguous substring. The *shortest superstring* (SS) problem is to find a superstring of minimum length.

This problem has important applications in computational biology and in data compression (see, respectively, [A. Lesk, ed., *Computational Molecular Biology, Sources and Methods for Sequence Analysis*, Oxford University Press, Oxford, 1988]; [J. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988]). SS is MAX SNP-hard [A. Blum et al., *Proc.* 23*rd Annual ACM Symposium on Theory of Computing*, ACM, New York, 1991, pp. 328–336] so it is unlikely that the length of a shortest superstring can be approximated to within an arbitrary constant. Several heuristics have been suggested and it is conjectured that GREEDY achieves an approximation factor of 2. This, unfortunately, remains an open question.

Several linear approximation algorithms for SS have been proposed. The first, by Blum et al. [*Proc.* 23*rd Annual ACM Symposium on Theory of Computing*, ACM, New York, 1991, pp. 328–336], guarantees a performance factor of 3. The factor has been successively improved to $2\frac{8}{9}$, $2\frac{5}{6}$, $2\frac{50}{63}$, $2\frac{3}{4}$, $2\frac{2}{3}$, and 2.596 (see, respectively, [S. Teng and F. Yao, *Proc.* 34*th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Piscataway, NJ, 1993, pp. 158–165]; [A. Czumaj et al., *Proc. First Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Comput. Sci. 824, Springer-Verlag, Berlin, 1994, pp. 95–106]; [R. Kosaraju, J. Park, and C. Stein, *Proc.* 35*th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Piscataway, NJ, 1994, pp. 166–177]; [C. Armen and C. Stein, *Proc.* 5*th Internat. Workshop on Algorithms and Data Structures*, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 494–505]; [C. Armen and C. Stein, *Proc. Combinatorial Pattern Matching*, Lecture Notes in Comput. Sci. 1075, Springer-Verlag, Berlin, 1996, pp. 87–101]; and [D. Breslauer, T. Jiang, and Z. Jiang, *J. Algorithms*, 24 (1997), pp. 340–353]). In this paper we give an algorithm that guarantees a $2\frac{1}{2}$-approximation factor.

Key words. algorithm, shortest superstring, shotgun sequencing, data compression

AMS subject classifications. 11Y16

PII. S0097539796324661

1. Introduction.

Let $S = \{s_1, \ldots, s_n\}$ be a set of strings. A *superstring* of S is a string containing each $s_i \in S$ as a contiguous substring. The *shortest superstring* (SS) problem is to find a minimum length superstring of the input set S. We use OPT_S to denote a shortest superstring of S and $|OPT_S|$ to denote the string's length.

SS has important applications in computational biology [6], [8]. For example, DNA is a (double-stranded) sequence of four types of nucleotides: adenine, cytosine, guanine, and thymine. The sequence has an orientation and can be viewed as a string over the alphabet $\{A, C, G, T\}$. The *sequencing* problem in molecular biology is to "read" a string of DNA. There are laboratory methods to read fairly short pieces of DNA. To read a longer piece, many copies are made that are then cut into smaller, overlapping pieces that can be read. A typical approach to reassembling them is finding a short superstring; intuitively, short superstrings preserve important

*Received by the editors July 22, 1996; accepted for publication (in revised form) June 29, 1998; published electronically December 15, 1999. This research was funded by Department of Education grant P200A40206 and was initiated while the author was visiting Sandia National Labs, Albuquerque, NM.

http://www.siam.org/journals/sicomp/29-3/32466.html

†CIS Division, University of Pennsylvania, Philadelphia, PA 19103 (zzz@saul.cis.upenn.edu).

biological structure and are good models of the original DNA sequence. SS also has applications in data compression [9], as data may be stored efficiently as a superstring.

In practice, biologists tend to use heuristics to find short superstrings. Probably the most widely used heuristic is GREEDY. It merges two strings in the input set with largest overlap, repeating until only one string remains. There are examples in which GREEDY produces a string of length $(2 - o(1))|OPT_S|$ and it is conjectured that this bound is tight.

SS is MAX SNP-hard [3], so it is unlikely that $|OPT_S|$ can be approximated to within an arbitrary constant. A p-approximation algorithm for SS produces a superstring of length at most $p \cdot |OPT_S|$; p is called the approximation factor of the algorithm. Blum et al. [3] gave the first linear approximation bound for SS; they showed that GREEDY is a 4-approximation algorithm and also gave a GREEDY variant that gives a 3-approximation. The 3-approximation factor was improved in a series of papers yielding approximation ratios of $2\frac{8}{9}$ by Teng and Yao [11]; $2\frac{5}{6}$ by Czumaj et al. [5]; $2\frac{50}{63}$ by Kosaraju, Park, and Stein [7]; $2\frac{3}{4}$ by Armen and Stein [1]; $2\frac{2}{3}$ by Armen and Stein [2]; and 2.596 by Breslauer, Jiang, and Jiang [4]. Here we give a $2\frac{1}{2}$-approximation algorithm.

The general approach of these algorithms is a two-stage process. They first partition the input set S into sets $S_1, S_2, \ldots, S_k$ and construct a superstring α_i for each S_i. In the second stage they construct a superstring α on the set $S' = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$; α is also a superstring of S. The strings in S' enjoy some nice properties that generally don't hold for S; Blum et al. are able to show that $|\alpha| \leq |OPT_{S'}| + |OPT_S|$. The 3-approximation proof is completed by showing that $|OPT_{S'}| \leq 2|OPT_S|$.

The partition of S described above is defined by a *cycle cover* on the *distance/overlap graph*. This graph, denoted G_S, is a complete digraph with vertex set S; each edge of the graph is assigned a positive *length*. A *cycle cover* of G_S is a set of simple cycles that partition the vertices of the graph. A minimum-length cycle cover of G_S, denoted ${\mathcal{C}_S}^*$, can be found in polynomial time, and the length of such a cover is no more than $|OPT_S|$. Each set S_i described above corresponds to a distinct cycle $c \in {\mathcal{C}_S}^*$.

Subsequent improvements in the approximation factor for SS have been achieved primarily by improving the $|\alpha| \leq |OPT_{S'}| + |OPT_S|$ bound; by more careful construction of the superstrings α_i in S', [2] and [4] show that $|\alpha| \leq |OPT_{S'}| + \frac{2}{3}|OPT_S|$. This general approach has obvious limitations: one needs to give an optimal superstring algorithm for the set S' in order to achieve a 2-approximation for the input set S, and the superstring problem remains MAX-SNP hard when restricted to input sets with the properties of S'.[1]

Our algorithm begins by constructing ${\mathcal{C}_S}^*$; then it *merges* cycles in ${\mathcal{C}_S}^*$ to produce a new cycle cover $\mathcal{C}$, and finally it *opens* each cycle of $\mathcal{C}$ to produce a set of strings. The concatenation of these strings yields a superstring of S; we denote this approximately optimal superstring as $AOPT_S$. Unlike the superstrings produced by the previous algorithms, $AOPT_S$ is not necessarily a superstring over some S' as described earlier. As a consequence we are able to avoid the problematic bound; there is no obvious reason our approach can't be improved to give a 2-approximation.

To analyze the length of $AOPT_S$ we define a generalized distance/overlap multigraph, $\mathcal{G}_S$. We also define an integer-valued function f. Our result is given in the following theorem; we call $\mathcal{C}_L$ the *lower bound cycle cover* and $\mathcal{C}_U$ the *upper bound cycle cover*.

[1]The property is that the self-loops of $G_{S'}$ are a minimum-length cycle cover of the graph.

Theorem 1.1. *Let S be a set of strings. There exist cycle covers $\mathcal{C}_U$ and $\mathcal{C}_L$ of $\mathcal{G}_S$ such that*

$$|AOPT_S| \leq |\mathcal{C}_U| \leq \frac{5}{2} f(\mathcal{C}_L) \leq \frac{5}{2}|OPT_S|.$$

Our algorithm runs in time $O(n^3 + L_S)$, where $L_S = \sum_{s \in S}|s|$, which is the same as the other approximation algorithms for this problem. We conjecture that our algorithm can be modified slightly and the analysis improved to show a $2\frac{1}{3}$ or better approximation bound. In addition, our techniques seem to shed light on GREEDY and may lead to better bounds for that algorithm.

In section 2 we give some basic definitions and lemmas related to strings. We also define the distance/overlap graph and give an algorithm that finds a minimum-length cycle cover of the graph. In section 3 we define the generalized distance/overlap multigraph and discuss properties of its cycles and cycle covers. In section 4 we describe our approximation algorithm and prove Theorem 1.1, subject to two lemmas which are proved in sections 5 and 6.

2. Background. In this section we give some basic definitions and well-known lemmas relating string overlap to periodic structure. We also define the distance/overlap graph and describe some properties of its cycles and cycle covers. In the following, s, t, u, v, and w refer to strings and S refers to a set of strings.

The string s^q (where $q|s|$ is integral) is defined recursively as the prefix of s of length $q|s|$ when $q < 1$ and the string ss^{q-1} otherwise. For example, $(ba)^0$ is the empty string, which we denote as ϵ, $(ba)^{1/2} = b$, $(ba)^{3/2} = bab$, and $(ba)^2 = baba$.

We say the string v is an *overlap of s with respect to t* if v is a proper suffix of s and a proper prefix of t. $OV(s,t)$ is the set of overlaps of s with respect to t, and $ov_\ell(s,t)$ denotes the string in $OV(s,t)$ of length ℓ, provided such a string exists. $OV(s,t)$ always contains the empty string ϵ and thus is nonempty for any s, t. When $ov_\ell(s,t)$ exists we denote the strings u and w, where $s = u \cdot ov_\ell(s,t)$ and $t = ov_\ell(s,t) \cdot w$, as $pref_\ell(s,t)$ and $suff_\ell(s,t)$. We use $ov(s,t)$ to denote the longest string in $OV(s,t)$; $pref(s,t)$ and $suff(s,t)$ denote the corresponding prefix of s and suffix of t. Finally, we define $ov_{max}(s,t) = \max(|ov(s,t)|, |ov(t,s)|)$.

Example 2.1. *Consider the strings $u_1 = c(ab)^j$ and $u_2 = (ba)^k$ for integers $j, k \geq 1$. Then $OV(u_1,u_2) = \{\epsilon, (ba)^{i-\frac{1}{2}}, i = 1, \ldots, \min(j,k)\}$, $OV(u_2,u_1) = \{\epsilon\}$, $OV(u_1,u_1) = \{\epsilon\}$, and $OV(u_2,u_2) = \{(ba)^i, i = 0, \ldots, k-1\}$.*

Note that in this example u_i, $i = 1, 2$, is not in $OV(u_i,u_i)$ since the self-overlap of u_i must be a proper prefix and proper suffix of u_i. Lemma 2.2 is due to [12] and [13]. The proof follows from Figure 1. Lemma 2.3 follows by a similar argument.

Lemma 2.2. *Let s, t, u, and v be strings. If $ov_k(s,t)$, $ov_\ell(s,u)$, and $ov_j(v,t)$ exist for $k \geq \max(j,\ell)$, then $ov_m(v,u)$ exists for $m = \max(0, j + \ell - k)$.*

Lemma 2.3. *Let s, t, and u be strings. If $ov_k(s,t)$ and $ov_j(t,u)$ exist, then so does $ov_\ell(s,u)$, where $\ell = \max(0, j + k - |t|)$.*

If $s = uv$, then vu is a *cyclic rotation* of s. If all the cyclic rotations of s are distinct, then s is *irreducible*. If not, then s is *reducible*. We say that $s \equiv t$ if s is a cyclic rotation of t; $\equiv$ is an equivalence relation and $[s]$ denotes the equivalence class of cyclic rotations of s. Thus s is irreducible if and only if $||[s]|| = |s|$. We say that s is *periodic* in t if $s = t^q$ for some q. We also say that s is periodic in $[t]$ to mean that s is periodic in some cyclic rotation of t. Any string s is periodic in $pref(s,s)$. We call $pref(s,s)$ the *period of s* and use the shorthand δ_s to denote this string. It is easy

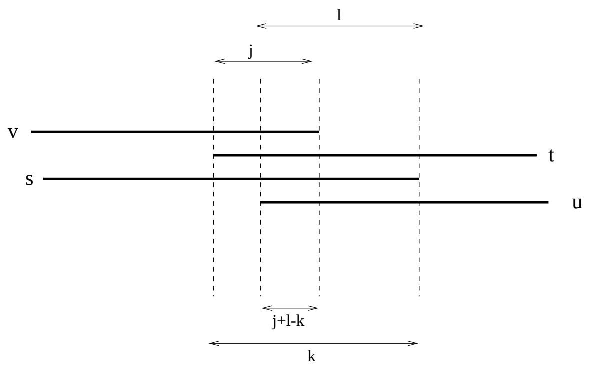


FIG. 1. *Overlap relationships.*

to see that δ_s is irreducible. In Example 2.1 $\delta_{u_1} = u_1$ and $\delta_{u_2} = ba$. The following lemmas and corollary are explicit or implicit in [3].

LEMMA 2.4. *For any string s, δ_s is the shortest string in which s is periodic. Furthermore, for $v \in OV(s,s)$, either $|v| \equiv |s| \bmod |\delta_s|$ or $|v| < |\delta_s|$.*

LEMMA 2.5. *Let δ_1 and δ_2 be irreducible strings such that $[\delta_1] \neq [\delta_2]$. If s is periodic in $[\delta_1]$ and in $[\delta_2]$, then $|s| < |\delta_1| + |\delta_2|$.*

Since $ov(s,t)$ is a substring of s and t and thus periodic in $[\delta_s]$ and $[\delta_t]$, we get the following corollary.

COROLLARY 2.6. *If s and t are strings such that $[\delta_s] \neq [\delta_t]$, then $|ov(s,t)| < |\delta_s| + |\delta_t|$.*

We say the set of strings $S = \{u_0, \dots, u_{n-1}\}$ is *substring free* if s is not a substring of t for any distinct strings $s,t \in S$. We assume, without loss of generality, that our input set S is substring free. Then for some permutation π of $[0, n-1]$, the string

$$pref(u_{\pi_0}, u_{\pi_1}) \cdot pref(u_{\pi_1}, u_{\pi_2}) \cdots pref(u_{\pi_{n-2}}, u_{\pi_{n-1}}) \cdot u_{\pi_{n-1}}$$

is a shortest superstring in S. We let π^*_S denote some such permutation for a substring-free set of strings S.

The *distance/overlap graph for S* is a digraph with vertex set S, and for every $s,t \in S$ an edge e from s to t with *length* $|e| = |pref(s,t)|$ and *overlap* $ov(e) = |ov(s,t)|$. As is standard in weighted graphs, the weight, in this case either length or overlap, of a cycle or cycle cover in G_S is the sum of the weights of its edges. Any permutation π of $[0, n-1]$ defines a Hamiltonian cycle c_π in G_S, $c_\pi = (u_{\pi_0}, u_{\pi_1}, \dots, u_{\pi_{n-1}}, u_{\pi_0})$. The following theorem given by Blum et al. [3] follows from the facts that $c_{\pi^*_S}$ is a cycle cover of G_S and that $|c_{\pi^*_S}| \leq |OPT_S|$.

THEOREM 2.7. *Let $\mathcal{C}$ be a minimum-length cycle cover of G_S. Then $|\mathcal{C}| \leq |OPT_S|$.*

Note that a minimum-length cycle cover of G_S is also a maximum-overlap cover. Blum et al. [3] show that the following algorithm constructs a maximum-overlap cover of G_S.

```
GREEDY-COVER($G_S$)
 Let $\mathcal{C}_S{}^* = \phi$. Order the edges of $G_S$ as $e_1, \dots, e_{n^2}$ so that $ov(e_i) \geq ov(e_{i+1})$.
 For $i = 1, \dots, n^2$
         Add $e_i = \langle s,t \rangle$ to $\mathcal{C}_S{}^*$ if $s$ does not have an out-edge and $t$ does not have an
             in-edge in $\mathcal{C}_S{}^*$.
```

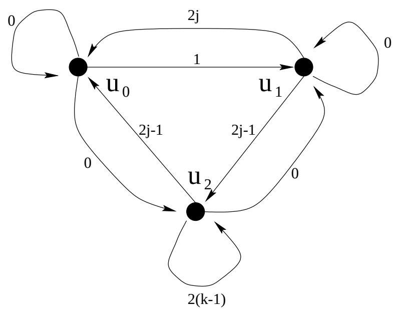


FIG. 2. $G_{S_{j,k}}$ *for* $1 \leq j \leq k$ *with edge overlaps shown.*

EXAMPLE 2.8. *Consider the class of sets*

$$S_{j,k} = \{u_0 = (ab)^j c, u_1 = c(ab)^j, u_2 = (ba)^k\}$$

for integers $k \geq j \geq 1$*. The graph* $G_{S_{j,k}}$ *is shown in Figure* 2*. For this example*

1. $OPT_S = c(ab)^{k+1}c$*,* $|OPT_S| = 2k + 4$*,*
2. $c_{\pi_S^*} = (u_0, u_1, u_2, u_0)$*,* $|c_{\pi_S^*}| = 2k + 3$*,*
3. $\mathcal{C}_S{}^* = \{c_1 = (u_0, u_1, u_0), c_2 = (u_2, u_2)\}$*,* $|\mathcal{C}_S{}^*| = 2j + 3$*.*

As the previous example illustrates, $|\mathcal{C}_S{}^*|$ may be a very good lower bound for $|OPT_S|$ or an arbitrarily bad lower bound; e.g., set $j = k$ or fix j and let k be arbitrarily large.

Throughout this paper we describe various features of the simple cycles of G_S (and later describe simple cycles of the generalized graph $\mathcal{G}_S$); we use *cycle* synonymously with *simple cycle*. Let $c = (s_0, s_1, \ldots, s_{j-1}, s_0)$ be a cycle of G_S. The *periods* of c, denoted $[c]$, are the equivalence class of cyclic rotations of the string

$$pref(s_0, s_1), \ldots, pref(s_{j-2}, s_{j-1}), pref(s_{j-1}, s_0).$$

For any ℓ, the string

$$x = pref(s_\ell, s_{\ell+1}) \cdot pref(s_{\ell+1}, s_{\ell+2}) \cdots pref(s_{\ell+j-2}, s_{\ell+j-1}) \cdot s_{\ell+j-1},$$

where the indices are taken modulo j, is called an *open* of c. The vertices s_ℓ and $s_{\ell+j-1}$ are called, respectively, x_{first} and x_{last}. The edge $\langle x_{last}, x_{first}\rangle$ is called the *opening edge* of x. We use $op(c)$ to denote a shortest open of c.

FACT 2.9. *Let c be a cycle in G_S and let x be an open of c with opening edge e. Then* $|x| = |c| + ov(e)$*.*

For $c \in \mathcal{C}_S{}^*$, we define $OP(c)$ to be the set of opens of c and $U_S^* = \cup_{c \in \mathcal{C}_S^*} OP(c)$. For $x \in U_S^*$, we use c_x to denote the cycle for which x is an open and e_x to denote the opening edge of x in c_x. Thus $\mathcal{C}_S{}^*$ consists of the edges $\{e_x \mid x \in U_S^*\}$. For $x, y \in U^*$, $c_x \neq c_y$, we define $e_{max}(x, y)$ to be the maximum-overlap edge in $\{\langle x_{last}, y_{first}\rangle, \langle y_{last}, x_{first}\rangle\}$.

EXAMPLE 2.8 (continued).

1. *The set* U_S^* *for this example is*

$$U_S^* = \{x_0 = (ab)^j c(ab)^j, x_1 = c(ab)^j c, x_2 = (ba)^k\}.$$

2. *The opening edges for* $x \in U_S^*$ *are* $e_{x_0} = \langle u_1, u_0\rangle, e_{x_1} = \langle u_0, u_1\rangle$*, and* $e_{x_2} = \langle u_2, u_2\rangle$*.*

3. *The shortest opens of the cycles in* ${\mathcal{C}_S}^*$ *are* $op(c_1) = x_1$, $op(c_2) = x_2$.

Blum et al. [3] proved the following lemmas. The second of these lemmas, together with Theorem 2.7, provides the bound described in the introduction, $OPT_{S'} \leq 2OPT_S$, when S' contains an open for each $c \in {\mathcal{C}_S}^*$.

LEMMA 2.10. *Let* c_x *and* c_y *be distinct cycles in* ${\mathcal{C}_S}^*$ *with opens, respectively,* x *and* y. *Then*

1. $[\delta_x] = [c_x] \neq [c_y] = [\delta_y]$,
2. $OV(x, x) = OV(x_{last}, x_{first})$ *and* $OV(x, y) = OV(x_{last}, y_{first})$,
3. $ov(e_{max}(x, y)) = ov_{max}(x, y) < |c_x| + |c_y|$.

LEMMA 2.11. *Let* X *be a subset of the cycles in* ${\mathcal{C}_S}^*$, *let* $T \subseteq S$ *be the strings corresponding to* $c \in X$, *and let* V *contain exactly one* $x \in OP(c)$ *for each* $c \in X$. *Then*

$$OPT_{S-T\cup V} \leq OPT_S + \sum_{c \in X} |c|.$$

Sweedyk [10] proved the following lemma. Since finding a shortest open of a cycle in G_S is easy, we can assume that ${\mathcal{C}_S}^*$ consists of more than one cycle.

LEMMA 2.12. *If* ${\mathcal{C}_S}^*$ *consists of a single cycle* c, *then* $op(c)$ *is a shortest superstring of* S.

We now define the edge exchange, which is a general purpose tool used throughout our proof.

DEFINITION 2.13. *Let* $\mathcal{C}$ *be a cycle cover and let* $e = \langle s, t\rangle$ *be an edge of* G_S. *Assume* $e_1 = \langle s, u\rangle$ *and* $e_2 = \langle v, t\rangle$ *are, respectively, the out-edge of* s *and the in-edge of* t *in* $\mathcal{C}$. *The edge exchange of* e *in* $\mathcal{C}$, *denoted* $\mathcal{X}(\mathcal{C}, e)$, *is the cycle cover* $\mathcal{C} - \{e_1, e_2\} \cup \{e, e_3\}$, *where* $e_3 = \langle v, u\rangle$.

Note that the in- and out-degrees of each vertex are unchanged by the edge exchange, so if $\mathcal{C}$ is a cycle cover and e an edge of G_S, then $\mathcal{X}(\mathcal{C}, e)$ is in fact a cycle cover of G_S. Also, we make no assumptions about distinctness of the strings u, v, s, and t in this definition.

DEFINITION 2.14. *Let* $\mathcal{C}$ *be a cycle cover and let* $e = \langle s, t\rangle$ *be an edge of* G_S. *Assume* $e_1 = \langle s, u\rangle$ *and* $e_2 = \langle v, t\rangle$ *are, respectively, the out-edge of* s *and the in-edge of* t *in* $\mathcal{C}$. *We say that* $e = \langle s, t\rangle$ *is a* winning edge *for* $\mathcal{C}$ *if* $ov(e) \geq \max(ov(e_1), ov(e_2))$.

LEMMA 2.15. *Let* $\mathcal{C}$ *be a cycle cover of* G_S *and let* e *be a winning edge for* $\mathcal{C}$. *Then* $|\mathcal{X}(\mathcal{C}, e)| \leq |\mathcal{C}|$.

This follows directly from Lemmas 2.2 and 2.10. In Example 2.8 the edge $e = \langle u_1, u_0\rangle$ is a winning edge for $c_{\pi_S^*} = (u_0, u_1, u_2, u_0)$ and $\mathcal{X}(\{c_{\pi_S^*}\}, e) = {\mathcal{C}_S}^*$. If $k > j$, then $\langle u_2, u_2\rangle$ is also a winning edge and the corresponding edge exchange also produces ${\mathcal{C}_S}^*$. The next fact follows from the construction in GREEDY-COVER.

FACT 2.16. *Let* $\langle s, t\rangle$ *be any edge in* G_S *and let* $\langle s, u\rangle$ *and* $\langle v, t\rangle$ *be, respectively, the out-edge of* s *and the in-edge of* t *in* ${\mathcal{C}_S}^*$. *Then* $\max(ov(s, u), ov(v, t)) \geq ov(s, t)$.

Lemma 2.15 and Fact 2.16 form the basis of the following optimality proof of ${\mathcal{C}_S}^*$. We generalize this proof in section 6.

1. Color the edges of ${\mathcal{C}_S}^*$ *white* and the remaining edges of G_S *black*. Let $\mathcal{C}_0$ be any cycle cover of G_S and set $i = 0$.
2. While $\mathcal{C}_i$ contains a *black* edge, choose the maximum overlap *white* edge e that is not in $\mathcal{C}_i$ and set $\mathcal{C}_{i+1} = \mathcal{X}(\mathcal{C}_i, e)$ and $i = i + 1$.

The construction produces a series of covers $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_m$. By Fact 2.16 each edge selected as the basis of an edge exchange is a winning edge for the current cover so $|\mathcal{C}_{i+1}| \leq |\mathcal{C}_i|$. The final cover consists entirely of *white* edges and thus is ${\mathcal{C}_S}^*$.

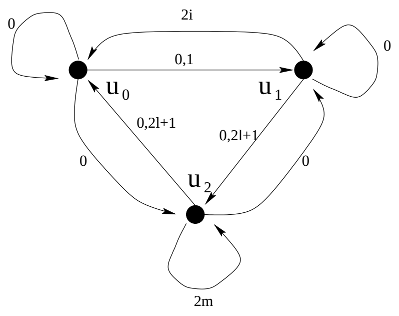


FIG. 3. $\mathcal{G}_{S_{j,k}}$ *with edge overlaps shown, where* $i \in [0, j]$, $\ell \in [0, j-1]$, *and* $m \in [0, k-1]$. *(Recall* $u_0 = (ab)^j c$, $u_1 = c(ab)^j$, $u_2 = (ba)^k$.*)*

3. The string functions and lemmas.

In this section we introduce several definitions that capture some of the structure of the overlaps of strings in U_S^*. To analyze this structure we generalize the distance/overlap graph as follows.

DEFINITION 3.1. *The distance/overlap multigraph* $\mathcal{G}_S$ *for a set of strings* S *is a digraph with vertex set* S *and, for every* $s, t \in S$, *and every* $v \in OV(s, t)$, *an edge* $\langle s, t, |v|\rangle$ *from* s *to* t *with length* $|s| - |v|$ *and overlap* $|v|$.

Figure 3 shows $\mathcal{G}_S$ for the set of strings in Example 2.8. An edge $\langle s, t, k\rangle$ of $\mathcal{G}_S$ is *tight* if $k = |ov(s, t)|$ and *loose* otherwise. The tight edge $\langle s, t, |ov(s, t)|\rangle$ may be denoted simply as $\langle s, t\rangle$. Edge $e = \langle s, t, k\rangle$ and $e' = \langle s, t, k'\rangle$ are *versions* of each other and if $|e| \geq |e'|$ we say that e is an expansion of e'. These definitions extend to cycles (as before, we mean *simple cycles*) and to cycle covers in the natural way; e.g., c is an expansion of c' if every edge of c is an expansion of an edge of c'. Any edge of a maximum-overlap cover of $\mathcal{G}_S$ must be tight, so $\mathcal{C}_S{}^*$ is an optimal cover of $\mathcal{G}_S$.

We call a cycle $\hat{c}$ of $\mathcal{G}_S$ a 1-*expansion* of $c \in \mathcal{C}_S{}^*$ if $\hat{c}$ is an expansion of c; furthermore, it has at most one loose edge. When we refer to a 1-expansion of c_x for $x \in U_S^*$ we mean that the only possible loose edge in the 1-expansion is from x_{last} to x_{first}. There is a 1-expansion of c_x for each expansion of e_x in $\mathcal{G}_S$. We could define 1-expansions for arbitrary cycles in G_S, but we never need this broader definition, so by 1-expansion we implicitly mean a 1-expansion of some cycle $c \in \mathcal{C}_S{}^*$. (We will use the more general notion of expansion for arbitrary simple cycles of $\mathcal{G}_S$.)

The definitions of period and open extend in the natural way for cycles in $\mathcal{G}_S$. For a cycle

$$c = \langle s_0, s_1, k_0\rangle, \langle s_1, s_2, k_1\rangle, \ldots, \langle s_{j-1}, s_0, k_{j-1}\rangle$$

in $\mathcal{G}_S$ $[c] = [\mathit{pref}_{k_0}(s_0, s_1) \cdot \mathit{pref}_{k_1}(s_1, s_2) \cdots \mathit{pref}_{k_{j-1}}(s_{j-1}, s_0)]$, and for any ℓ,

$$x = \mathit{pref}_{k_\ell}(s_\ell, s_{\ell+1}) \cdot \mathit{pref}_{k_{\ell+1}}(s_{\ell+1}, s_{\ell+2}) \cdots \mathit{pref}_{k_{\ell+j-2}}(s_{\ell+j-2}, s_{\ell+j-1}) \cdot s_{\ell+j-1}$$

is an open of c, where the indices are taken modulo j. Furthermore, $|x| = |c| + ov(e)$, where $e = \langle s_{\ell+j-1}, s_\ell, k_{\ell+j-1}\rangle$ is the opening edge of x. It follows that if a cycle c is an expansion of c' in $\mathcal{G}_S$, then $|op(c)| \geq |op(c')|$. Also, for $x \in U_S^*$, x is an open of any 1-expansion $\hat{c}_x$ of c_x and its opening edge in $\hat{c}_x$ is an expansion of e_x.

Definition 2.14 for a winning edge does not change for covers in $\mathcal{G}_S$; an edge $e = \langle s, t, k\rangle$ of $\mathcal{G}_S$ is a winning edge for a cycle cover $\mathcal{C}$ of $\mathcal{G}_S$ if k is at least as large as the overlap of the out-edge of s and the in-edge of t in $\mathcal{C}$. We continue to use the

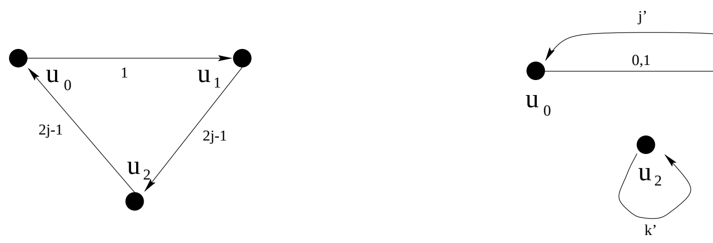


FIG. 4. *The cycle $c_{\pi_S^*}$ is shown on the left and the cycle cover $\tilde{\mathcal{X}}(c_{\pi_S^*}, e)$ on the right. If $e = e_{x_0}$, then $j' = 2j$ and $k' = 2j - 2$. If $e = \langle u_2, u_2, 2m\rangle$ for some $2k > 2m > 2j - 1$, then $j' = \max(0, 2(2j - 1 - m))$ and $k' = 2m$.*

edge exchange function $\mathcal{X}$ from Definition 2.13, which uses tight edges, but we define another edge exchange function, $\tilde{\mathcal{X}}$, which exploits the loose edges of $\mathcal{G}_S$.

DEFINITION 3.2. *Let $\mathcal{C}$ be a cycle cover and $e = \langle s, t, k\rangle$ be an edge of $\mathcal{G}_S$. Assume $e_1 = \langle s, u, j\rangle$ and $e_2 = \langle v, t, \ell\rangle$ are, respectively, the out-edge of s and the in-edge of t in $\mathcal{C}$. The* parsimonious *edge exchange of e in $\mathcal{C}$, denoted $\tilde{\mathcal{X}}(\mathcal{C}, e)$, is the cycle cover $\mathcal{C} - \{e_1, e_2\} \cup \{e, e_3\}$, where if $k \geq \max(j, \ell)$, then $e_3 = \langle v, u, \max(0, j + \ell - k)\rangle$; otherwise $e_3 = \langle v, u\rangle$.*

The existence of the edge e_3 in this definition is guaranteed by Lemma 2.2 when $k \geq j, \ell$. We call e_3 the *losing edge* of the edge exchange.

LEMMA 3.3. *Let $\mathcal{C}$ be a cycle cover of $\mathcal{G}_S$ and let e be a winning edge for $\mathcal{C}$. Then $|\tilde{\mathcal{X}}(\mathcal{C}, e)| \leq |\mathcal{C}|$. Also, the overlap of the losing edge of the exchange is no more than the minimum overlap of the edges eliminated by the exchange. Finally, if $|\tilde{\mathcal{X}}(\mathcal{C}, e)| < |\mathcal{C}|$, then the losing edge of the exchange has zero overlap.*

EXAMPLE 2.8 (continued). *The edge $e_{x_0} = \langle u_1, u_0\rangle$ is a winning edge for $c_{\pi_S^*}$. When $k > j$ and $2k > 2m \geq 2j - 1$, the edge $e_m = \langle u_2, u_2, 2m\rangle$ is also a winning edge for $c_{\pi_S^*}$. Figure 4 shows $c_{\pi_S^*}$ and $\tilde{\mathcal{X}}(\{c_{\pi_S^*}\}, e)$ for this example.*

To motivate the remaining definitions in this section we describe the general strategy of our proof relative to Example 2.8. We focus on two specific cases: case 1, where $j = 1$ and $k >> j$; and case 2, where $j = k > 1$. Our first objective is to establish the existence of a cover $\mathcal{C}_L$ whose length gives a *good* lower bound for $|OPT_S|$; for the moment *good* means that $|OPT_S|/|\mathcal{C}_L|$ is bounded. In case 2, setting $\mathcal{C}_L = \mathcal{C}_S{}^*$ suffices since $|\mathcal{C}_S{}^*| = |OPT_S| - 1$. In case 1, however, $|OPT_S|/|\mathcal{C}_S{}^*|$ is unbounded and the cover $\mathcal{C}_L$ must be a strict expansion of $\mathcal{C}_S{}^*$. We argue that, since $c_{\pi_S^*}$ is a Hamiltonian cycle, it does not contain $\langle u_2, u_2\rangle$ and thus must include either the edges $\langle u_1, u_2\rangle$ and $\langle u_2, u_0\rangle$ or the edges $\langle u_0, u_2\rangle$ and $\langle u_2, u_1\rangle$. In either case, $e = \langle u_2, u_2, 2\rangle$ is a winning edge for $c_{\pi_S^*}$. The cover $\mathcal{C}_L = \tilde{\mathcal{X}}(c_{\pi_S^*}, e)$ includes the 1-expansion $\hat{c}_1$ of c_1 containing the loose edge $\langle u_1, u_0, 0\rangle$ and the 1-expansion $\hat{c}_2$ of c_2 containing the loose edge $\langle u2, u_2, 2\rangle$. The length of $\mathcal{C}_L$ is exactly $|c_{\pi_S^*}| = |OPT_S| - 1$.

Our general argument is complicated by the fact that $\mathcal{C}_S{}^*$ typically contains more than two cycles, and the cycles may have many different opens. Nevertheless, our argument hinges on the fact that, for any $c \in \mathcal{C}_S{}^*$, there are vertices s and t in c whose in-edge and out-edge connect to vertices not in c; by analyzing the possibilities we can, when necessary, get a better bound for $|OPT_S|$ than that provided by $|\mathcal{C}_S{}^*|$. To this end we want to identify for x and y in U_S^*, $c_x \neq c_y$, the expansions of e_x and e_y that are winning edges for the cycle $c_{x,y} = \mathcal{X}(\{c_x, c_y\}, e_{max}(x, y))$; by this somewhat abusive notation we mean that $c_{x,y}$ is the tight cycle created by the edge exchange to the cover $\{c_x, c_y\}$ on the vertex-induced subgraph of $\mathcal{G}_S$ containing the vertices of c_x

and c_y. We are also interested in the forms of the 1-expansions resulting from edge exchanges to $c_{x,y}$.

Our second objective is to establish an appropriate upper bound on $AOPT_S$. In case 1 our algorithm would set $AOPT_S = x_1 \cdot x_2$. The cover $\mathcal{C}_L$ for this case has the *shallow* property—by that we mean[2] $|op(\hat{c}_i)| \leq 2|\hat{c}_i|$, $i = 1, 2$. Thus we get the following bound:

$$|AOPT_S| = |x_1| + |x_2| \leq |x_0| + |x_2| = \sum_{i=1,2} |op(\hat{c}_i)| \leq 2 \sum_{i=1,2} |\hat{c}_i| = 2|\mathcal{C}_L| \leq 2|OPT_S|.$$

The construction of $\mathcal{C}_U$ for this case is trivial and we omit the details. In case 2, however, the cover $\mathcal{C}_L$ does not have the shallow property and our bound is more complicated. For this case our algorithm would output the shortest open of the cycle $\mathcal{X}(\mathcal{C}_S^*, e_{max}(x_0, x_2))$, which just happens to be the string OPT_S. We cannot argue that strict expansions of e_{x_0} and e_{x_2} are winning edges for $c_{\pi_S^*}$ because of the relatively large overlap of $e_{max}(x, y)$. However, we can argue that there is an expansion $\mathcal{C}_U$ of of $\mathcal{C}_L$ such that $|\mathcal{C}_U| \leq \frac{5}{2}|\mathcal{C}_L|$, $e_{max}(x_0, x_2)$ is a winning edge for $\mathcal{C}_U$, and the cover $\tilde{\mathcal{X}}(\mathcal{C}_U, e_{max}(x_0, x_2))$ has a zero overlap edge. Because $\mathcal{C}_U$ is an expansion of $\mathcal{C}_L$ and thus of ${\mathcal{C}_S}^*$, $\tilde{\mathcal{X}}(\mathcal{C}_U, e_{max}(x_0, x_2))$ is an expansion of $\mathcal{X}({\mathcal{C}_S}^*, e_{max}(x_0, x_2))$ so we get

$$|op(\mathcal{X}({\mathcal{C}_S}^*, e_{max}(x_0, x_2)))| \leq |op(\tilde{\mathcal{X}}(\mathcal{C}_U, e_{max}(x_0, x_2)))| \leq \frac{5}{2}|\mathcal{C}_L| \leq \frac{5}{2}|OPT_S|.$$

To establish upper bounds for $AOPT_S$ we are interested in pairs $x, y \in U_S^*$, $c_x \neq c_y$, and the expansions of c_x and c_y for which the edge $e_{max}(x, y)$ is a winning edge.

For the remainder of this section let x and y be strings in U_S^*, $c_x \neq c_y$. Throughout this section we will use Lemma 2.10, which relates the self-overlaps and overlaps of x and y to the edges between $x_{first}, x_{last}, y_{first}, y_{last}$ in $\mathcal{G}_S$; i.e., $OV(x, x) = OV(x_{last}, x_{first})$ and $OV(x, y) = OV(x_{last}, y_{first})$. Next we develop some notation for describing the 1-expansion in $\mathcal{G}_S$.

DEFINITION 3.4. *Let x be a string in U_S^* and let $\hat{e}_x$ be an expansion of e_x. We denote the* 1*-expansion of c_x corresponding to $\hat{e}_x$ as c_x^d, where*

$$d = \frac{|x| - ov(\hat{e}_x)}{|c_x|}.$$

The quantity $d|c_x|$ is called the pseudolength *of the edge $\hat{e}$ and d is called the* normalized pseudolength *of the edge.*

FACT 3.5. *Let x be a string in U_S^*.*

1. *The* 1*-expansion c_x^d exists for some d if and only if there is an expansion of e_x with pseudo-length $d|c_x|$*
2. *If $\hat{e}_x$ is an expansion of e_x with pseudolength $d|c_x|$, then $d \geq 1$ with equality if and only if $\hat{e}_c = e_x$.*

Throughout our analysis we make the assertion that certain 1-expansions of c_x exist based on the definitions, lemmas, and corollaries of this section. In particular we define the *string functions* on U_S^* and $U_S^* \times U_S^*$ as *shallow*, *trade-off*, *stop*, and *squeeze.* We also prove various properties of the strings in U_S^* and their overlaps based on these functions: Lemmas 3.7, 3.10, 3.12, and 3.15. We call these the *string lemmas.*

DEFINITION 3.6. *Let x be a string in U_S^*. The* shallow *of x, denoted $sh(x)$, is the quantity $(|x| - |v|)/|\delta_x|$, where v is the longest string in $OV(x, x)$ such that $|v| \leq |\delta_x|$.*

[2]Note that the opens of $\hat{c}_1$ are x_0 and $c(ab)^{2j}c$.

Note that $sh(x)$ is well defined since $\epsilon \in OV(x,x)$ and $|\epsilon| = 0 < |\delta_x|$. The next lemma follows directly from Lemma 2.10.

LEMMA 3.7. *Let x be a string in U_S^*. Then $c_x^{sh(x)}$ exists and $|x| \leq (sh(x)+1)|c_x|$.*

By Lemma 2.4, the structure of the 1-expansions of c_x ordered by increasing length is

$$c_x = c_x^1, c_x^2, \ldots, c_x^k, c^{sh(x)}, c_x^{q_1}, c_x^{q_2}, \ldots, c_x^{q_j}, c_x^{|x|/|c_x|}$$

for some integer $k \geq 1, j \geq 0$ and some values $q_1 < q_2, \ldots, < q_{j-1} < q_j < |x|/|c_x|$, where possibly $c_x = c_x^{sh(x)}$ and/or $c_x^{sh(x)} = c_x^{|x|/|c_x|}$. Thus we get the following corollary to Lemma 2.4.

COROLLARY 3.8. *Let x be a string in U_S^*. Then c_x^k exists for any integer $1 \leq k \leq |x|/|c_x|$. If c_x^q exists for some nonintegral q, then $sh(x) \leq q \leq |x|/|c_x| \leq sh(x)+1$.*

The *trade-off* function allows us to identify the expansions of e_x and e_y that are winning edges for $c_{x,y}$.

DEFINITION 3.9. *Let x and y be strings in U_S^*, $c_x \neq c_y$. The* trade-off *of x with respect to y, denoted $tr(x,y)$, is defined as*

$$tr(x,y) = \frac{1}{|\delta_x|}(|x| - ov_{max}(x,y)).$$

LEMMA 3.10. *Let x and y be strings in U_S^*, $c_x \neq c_y$. Then*
1. *c_x^k exists for $k = \max(\lfloor tr(x,y) \rfloor, 1)$,*
2. *$|x| < (tr(x,y)+1)|c_x| + |c_y|$,*
3. *either $tr(x,y) \geq 1$ or $tr(y,x) \geq 1$,*
4. *any expansion of e_x with pseudolength at most $tr(x,y)|c_x|$ has overlap at least $ov(e_{max}(x,y))$, and*
5. *$e_{max}(x,y)$ is a winning edge for any 1-expansions c_x^q and c_y^r that exist, where $q \geq tr(x,y)$ and $r \geq tr(y,x)$.*

In the last point of the lemma, c_x^q and c_y^r exist for $q = |x|/|c_x| \geq tr(x,y)$ and $r = |y|/|c_y| \geq tr(y,x)$, so the claim is never vacuous. The proof of this lemma follows directly from point 3 of Lemma 2.10 and from Fact 2.16. If $tr(y,x) < 1$, then no expansion of e_y is a winning edge for $c_{x,y}$. In this case, by point 4 of Lemma 3.10, e_x is a winning edge for $c_{x,y}$. The following definition identifies the largest expansion of e_x that is guaranteed to be a winning edge for $c_{x,z}$ for any z, $c_z \neq c_x$, such that $tr(z,x) < 1$.

DEFINITION 3.11. *Let x be a string in U_S^*. The* stop *of x, denoted $st(x)$, is defined as*

$$st(x) = \min\{\lfloor tr(x,y) \rfloor \ : \ y \in U_S^*, c_y \neq c_x, tr(y,x) < 1\},$$

where min *over an empty set is ∞. When $st(x)$ is finite, $ST(x)$ denotes an arbitrary $y \in U_S^*$ that determines $st(x)$. If $st(x) = \infty$, then $ST(x)$ is undefined.*

LEMMA 3.12. *Let x be a string in U_S^* such that $st(x)$ is finite. Then $c_x^{st(x)}$ exists and, for $y = ST(x)$,*
1. *$|y| < 2|c_y| + |c_x|$,*
2. *$|x| < (st(x)+2)|c_x| + |c_y|$, and*
3. *$e_{max}(x,y)$ is a winning edge between the 1-expansions $c_x^{\min(st(x)+1, |x|/|c_x|)}$ and c_y.*

The proof follows from Lemma 3.10. If $tr(y,x) \geq 1$, then e_y is a winning edge for $c_{x,y}$, and $\tilde{\mathcal{X}}(\{c_{x,y}\}, e_y)$ consists of c_y and some 1-expansion of c_x. The following definition identifies the smallest 1-expansion of c_x that can be produced by such an edge exchange over all z, $c_z \neq c_x$, where $tr(z,x) \geq 1$.

Definition 3.13. *Let x be a string in U_S^*. The* squeeze *of x, denoted $sq(x)$, is defined to be*

$$sq(x) = \frac{1}{|\delta_x|} \min\{|c_{x,y}| - |c_y| \quad : \quad y \in U_S^*, c_y \neq c_x, |ov(y,y)| \geq ov_{max}(y,x)\},$$

where min *over an empty set is ∞. When $sq(x)$ is finite, $SQ(x)$ denotes an arbitrary $y \in U_S^*$ that determines $sq(x)$. If $sq(x) = \infty$, then $SQ(x)$ is undefined.*

Before we give the string lemma for the squeeze function, we give a technical lemma that will be useful here and later.

Lemma 3.14. *Let x and y be strings in U_S^* such that $c_x \neq c_y$. If $ov_k(y,y)$ exists for some $ov_{max}(x,y) \leq k < |ov(x,y)| + |ov(y,x)| - |\delta_x|$, then $ov_k(y,y)$ is periodic in $[\delta_x]$. Furthermore, $ov_{max}(x,y) > k - \min(|\delta_x|, |\delta_y|)$.*

Proof of Lemma 3.14. To see the proof of the first point refer to Figure 1 and let $s = t = y$, $u = v = x$, $j = |ov(y,x)|$, $\ell = |ov(x,y)|$, and let k be the self-overlap of y as defined in the lemma. Then $k \geq \max(j,\ell)$, so $ov_m(x,x)$ exists for $m = |ov(y,x)| + |ov(x,y)| - k$. By the upper bound on k in the lemma, $m > |\delta_x|$ and by Lemma 2.4 $pref_m(x,x) \cdot x$ is periodic in δ_x. Referring again to Figure 1, $ov_k(y,y)$ is a substring of $pref_m(x,x) \cdot x$ and thus periodic in $[\delta_x]$.

Now consider the second point of the lemma and suppose that $|\delta_x| \leq |\delta_y|$. By the previous point and the fact that x is periodic in $[\delta_x]$, if $ov_{\ell'}(x,y)$ exists for some $\ell' \leq k - |\delta_x|$, then $ov_{\ell'+|\delta_x|}(x,y)$ also exists. Thus $ov_{max}(x,y) \geq |ov(x,y)| > k - |\delta_x|$.

Now suppose that $|\delta_y| < |\delta_x|$ and $ov_{max}(x,y) \leq k - |\delta_y|$. Then, using Lemma 2.2, $ov_{m'}(x,x)$ exists for $m' = |ov(y,x)| + |ov(x,y)| - k + |\delta_y| = m + |\delta_y|$, where m is obtained from the proof of the first point of the lemma. By the upper bound assumption of the lemma, $m > |\delta_x|$ so by Lemma 2.4 $m' - m \equiv 0 \bmod |\delta_x|$. However, $m - m' = |\delta_y| < |\delta_x|$, which is a contradiction. ∎

The next lemma follows from Lemmas 3.10 and 3.14; to use Lemma 3.14 let $k = |ov(y,y)|$ and notice that $|c_{x,y}| = |x| + |y| - |ov(x,y)| - |ov(y,x)|$ and $|c_y| = |y| - |ov(y,y)|$.

Lemma 3.15. *Let x be a string in U_S^* such that $sq(x)$ is finite. Then $c_x^{sq(x)}$ exists and $|x| < (sq(x)+1)|c_x| + |c_{SQ(x)}|$. Furthermore, if $sq(x) < sh(x)$, then for $y = SQ(x)$*

1. *$ov(y,y)$ is periodic in $[\delta_x]$,*
2. *$|y| < 2|c_y| + |c_x|$,*
3. *the edge $e_{max}(x,y)$ is a winning edge between the 1-expansions $c_x^{sq(x)}$ and c_y^q, where $q = 1 + \frac{\min(|c_x|,|c_y|)}{|c_y|}$.*

We are primarily concerned with $sq(x)$ when $sq(x) < sh(x)$ because of the properties guaranteed by the previous lemma for $SQ(x)$.

Example 2.8 (continued). *The 1-expansions in this example are $c_{x_0}^{1+\frac{2\ell}{2j+1}}$, $\ell = 0,1,\ldots,j$, c_{x_1}, $c_{x_1}^{|x_1|/|c_{x_1}|}$, and $c_{x_2}^{\ell}$, $\ell = 1,2,\ldots,k$. The string functions are*

1. *shallow: $sh(x_0) = 1$, $sh(x_1) = 1$, and $sh(x_2) = k-1$;*
2. *trade-off: $tr(x_0,x_2) = 1$, $tr(x_2,x_0) = k-j$, $tr(x_1,x_2) = 1$, $tr(x_2,x_1) = 1$;*
3. *stop: $st(x_1) = st(x_2) = \infty$; if $j = k$, then $st(x_0) = 1$ and $ST(x_0) = x_2$; if $j > k$, then $st(x_0) = \infty$;*

```
1. Construct $G_S$ and find ${\mathcal{C}_S}^*$. Compute $U_S^*$ and the string functions.
2. Build the set of merging edges $W$.
3. Let $\mathcal{C} = {\mathcal{C}_S}^*$.
     While $W$ is nonempty do
          Let $e = \langle s, t \rangle$ be a minimum-overlap edge in $W$.
          If $s$ and $t$ are in different cycles of $\mathcal{C}$, then $\mathcal{C} = \mathcal{X}(\mathcal{C}, e)$.
          $W = W \setminus \{e\}$
4. Set $AOPT_S$ to the concatenation of $op(c)$, $c \in \mathcal{C}$.
```

FIG. 5. *The approximation algorithm.*

4. *squeeze:* $sq(x_0) \geq sh(x_0)$, $sq(x_1) \geq sh(x_1)$, *and either* $sq(x_2) \geq sh(x)$ *or* $sq(x) = 2(k - j - 1) \geq 1$ *and* $SQ(x_2) = x_0$.

4. The approximation algorithm. The approximation algorithm is given in Figure 5. In step 2 the algorithm builds a set of *merging edges* W; these are a subset of the tight edges of $\mathcal{G}_S$. In step 3 the algorithm merges cycles of ${\mathcal{C}_S}^*$ by edge exchanges in G_S based on the edges in W. The cycles of the resulting cover are opened and concatenated to produce the string $AOPT_S$.

During the course of step 2, the algorithm constructs an *expansion value function* and, implicitly, the lower bound cover $\mathcal{C}_L$.

DEFINITION 4.1. *A function* $\mathcal{V}_S : U_S^* \to \mathcal{Q}$ *is an* expansion value function *if for any* $x \in U_S^*$ *there is an expansion of* e_x *with pseudolength* $\mathcal{V}_S(x)|c_x|$. *Furthermore, we say that* $\mathcal{V}_S$ *is* valid *if it satisfies the following conditions:*

1. *For any* $x \in U_S^*$, $\mathcal{V}_S(x) \leq \min(sh(x), sq(x), st(x))$.
2. *For any* $x, y \in U_S^*$, $c_x \neq c_y$, *either* $\mathcal{V}_S(x) \leq tr(x, y)$ *or* $\mathcal{V}_S(y) \leq tr(y, x)$.

FACT 4.2. *Let* $\mathcal{V}_S$ *be an expansion value function. Then the edges*

$$\{\hat{e}_x \mid x \in U_S^* \text{ and } \hat{e}_x \text{ is an expansion of } e_x \text{ with pseudolength } \mathcal{V}_S(x)|c_x|\}$$

form a cycle cover that is an expansion of ${\mathcal{C}_S}^*$.

During step 2, the algorithm also assigns each cycle in ${\mathcal{C}_S}^*$ to a color class in $\{$*blue*, *red*, *yellow*$\}$, and for each *red* and *yellow* cycle c selects an $x \in OP(c)$ as *representative*, denoted *rep*(c). The *red* and *yellow* cycles are the ones for which we need a better bound in $\mathcal{C}_L$. The cover $\mathcal{C}_L$ will include each *blue* cycle and a 1-expansion of $c_{rep(c)}$ for each *red* and *yellow* cycle c. In section 6 we prove that $|\mathcal{C}_L| \leq |OPT_{S'}|$, where S' is constructed from S by replacing the strings associated with each *red* cycle c by *rep*(c). This reduces the constraints that $\mathcal{C}_L$ must satisfy, but at a price, since the best bound we have for $OPT_{S'}$ is, by Lemma 2.11, $|OPT_{S'}| \leq |OPT_S| + \sum_{c \in \mathcal{R}} |c|$, where $\mathcal{R}$ denotes the set of *red* cycles. The function f in Theorem 1.1 is $f(\mathcal{C}_L) = |\mathcal{C}_L| - \sum_{c \in \mathcal{R}} |c|$. We begin by discussing $\mathcal{V}_S$, the coloring, and the representatives. Then we define the cover $\mathcal{C}_L$. Finally, we discuss the selection of edges for W.

Step 2 is broken down into two parts. Part 1 is shown in Figure 6. Let N be the number of cycles in ${\mathcal{C}_S}^*$. The algorithm begins by ordering the cycles of ${\mathcal{C}_S}^*$ by increasing length. We use $c < c'$ (resp., $c > c'$) to denote the fact that c precedes (resp., follows) c' in the order. Throughout our discussion of part 1 of step 2, c_i, $1 \leq i \leq N$, refers to the ith cycle in the algorithm's order. The algorithm initializes $\mathcal{V}_S(x) = 1$ for every $x \in U_S^*$. The algorithm then proceeds in rounds (j, i), $j = 1, 2$ and $i = 1, \ldots, N$; in each round it selects zero or more cycles to color. For notational purposes, each $x \in U_S^*$ adopts the color of c_x and is uncolored while c_x is uncolored.

```
Order the cycles of $\mathcal{C}_S{}^*$ so $|c_1| \le |c_2| \le \cdots \le |c_N|$. Each cycle is initially uncolored.
For each $x \in U_S^*$, $\mathcal{V}_S(x) = 1$.
For $j = 1, 2$ {
        For $i = 1, \dots N$ {
            If $c_i$ is uncolored, then
                For each $x$ in $OP(c_i)$ compute $fo(x)$ and set

                        $g(x) = \min(sh(x), sq(x), st(x), of(x))$.

                Color $c_i$ by the first rule, if any, that holds for some $w \in OP(c_i)$.

                    Rule 1: $g(w) \le \frac{3}{2}$ and stopping condition 1 holds:
                      Color $c_i$ blue.
                      If $G(w)$ is defined and $G(w)$ is uncolored, then color $c_{G(w)}$ blue .
                      If $G(w)$ is defined, then add $e_{max}(w, G(w))$ to $W$.

                    Rule 2: $g(w) > \frac{3}{2}$ and some uncolored $u > w$ conflicts with
                    $w$ and stopping conditions 2 holds and if $G(w)$ is defined, then
                    $c_u \ne c_{G(w)}$:
                      Color $c_i$ red, $\mathcal{V}_S(w) = g(w)$, and $rep(c_i) = w$.
                      Color $c_u$ blue and add $e_{max}(w, u)$ to $W$.
                      *If $G(w)$ is defined and $G(w)$ is colored and $c_i$ is unmatched,
                        then add $e_{max}(w, G(w))$ to $W$.
                      If $G(w)$ is defined and $G(w)$ is uncolored, then color $c_{G(w)}$
                        blue and add $e_{max}(w, G(w))$ to $W$.

                    Rule 3: $g(w) > \frac{3}{2}$ and $G(w)$ is defined and $G(w)$ is uncolored
                    and $G(w) > w$ and stopping condition 3 holds:
                      Color $c_i$ red, $\mathcal{V}_S(w) = g(w)$, and $rep(c_i) = w$.
                      Color $c_{G(w)}$ blue and add $e_{max}(w, G(w))$ to $W$.

                    Rule 4: $j = 2$:
                      Color $c_i$ yellow.
                      For each $x \in OP(c_i)$ {
                          $\mathcal{V}_S(x) = g(x)$.
                          *If $x = rep(c_i)$ and $c_i$ is unmatched and $G(x)$ is defined and
                            $G(x)$ is colored non-yellow, then add $e_{max}(x, G(x))$ to $W$.
                          If $ST(x)$ is defined and $ST(x)$ is uncolored, then color $c_{ST(x)}$
                            blue and add $e_{max}(rep(c_i), ST(x))$ to $W$.
                      }
                }
```

FIG. 6. *Construction of the merging set W, part* 1.

Also for $x, y \in U_S^*$, we use $x < y$ to denote the fact that $c_x < c_y$. Similarly $x > y$ denotes $c_x > c_y$.

Let c_i be a cycle in $\mathcal{C}_S{}^*$. If c_i is colored when round (j, i) begins, the algorithm takes no action during the round. If c_i is uncolored when the round begins, the algorithm computes two functions $fo(x)$ and $g(x) = \min(sh(x), sq(x), st(x), fo(x))$ for each $x \in OP(c_i)$.

DEFINITION 4.3. *Let x be a string in U_S^*. The* force of x *during round (j,i) of the algorithm is defined as*

$$fo(x) = \min\{\max(\lfloor tr(x,y)\rfloor, 1) \mid y \in U_S^* \text{ is colored, } c_y \neq c_x, \text{ and } \mathcal{V}_S(y) \geq tr(y,x)\},$$

where min *over an empty set is ∞. If $fo(x)$ is finite in some round, then $FO(x)$ is set to $ST(x)$ if $st(x) = fo(x)$ and to an arbitrary $y \in U_S^*$ that determines the value of $fo(x)$ otherwise.*

LEMMA 4.4. *Let x be a string in U_S^*. Suppose the algorithm computes $g(x)$ during some round. Then $g(x) \geq 1$ and there is an expansion of e_x with pseudolength $g(x)|c_x|$.*

The proof follows from the string functions and lemmas. Hereafter we include $fo(\cdot)$ when we refer to the "string functions." After computing $fo(x)$ and $g(x)$ for $x \in OP(c_i)$ the algorithm may choose to color c_i *blue*, *red*, or *yellow*. If the algorithm colors c_i it may take additional action as well; it may select additional uncolored cycles and color them *blue*, it may set $\mathcal{V}_S(x) = g(x)$ for some strings $x \in OP(c_i)$, it may add edges incident to vertices of c_i to W, and it may choose an $x \in OP(c_i)$ as the representative of c_i. The next fact follows from the construction.

FACT 4.5. *Let x be a string in $OP(c_i)$ for some $c_i \in \mathcal{C}_S{}^*$. If $\mathcal{V}_S(x) \neq 1$ at the end of some round of the algorithm, then c_i is colored. Furthermore, c_i was colored in round (j,i), where j is either* 1 *or* 2, *and $\mathcal{V}_S(x)$ is set to $g(x)$ as computed at the beginning of round (j,i).*

LEMMA 4.6. *At every point of the algorithm after initialization, $\mathcal{V}_S$ is a valid expansion value function.*

Proof. Let x be a string in U_S^*. We first show that there is an expansion of e_x with pseudolength $\mathcal{V}_S(x)|c_x|$. By Fact 4.5, $\mathcal{V}_S(x)$ is either 1 or is set to $g(x)$ as computed in the round when x is colored. In the first case e_x satisfies the claim. The claim follows in the second case by Lemma 4.4.

Consider the first condition for validity. By the definition of the string functions and lemmas, $1 \leq \min(sh(x), q(x), st(x), fo(x))$. Then the claim follows from Fact 4.5.

Finally, we show that the second condition for validity holds. Let y be a string in U_S^* such that $c_x \neq c_y$. First consider the case where $\mathcal{V}_S(x) = 1$. If $tr(x,y) \geq 1$, we are done, so assume $tr(x,y) < 1$. Then $st(y) \leq tr(y,x)$ and since $\mathcal{V}_S$ is an expansion value function, $\mathcal{V}_S(y) \leq st(y)$. A symmetric argument holds if $\mathcal{V}_S(y) = 1$. Thus we need only consider the case where $\mathcal{V}_S(x) > 1$ and $\mathcal{V}_S(y) > 1$. By Fact 4.5 and the fact that $c_x \neq c_y$, x and y are colored in different rounds. Assume, without loss of generality, that y is colored before x. Then in the round when x is colored, $\mathcal{V}_S(x)$ is set to $g(x)$, where $1 < g(x) \leq fo(x) \leq \max(\lfloor tr(x,y)\rfloor, 1) = \lfloor tr(x,y)\rfloor$, and the condition is satisfied. ☐

DEFINITION 4.7. *Let x be a string in $OP(c_i)$, $c_i \in \mathcal{C}_S{}^*$, and let y be a string in U_S^*, $c_y \neq c_i$. Suppose c_i is uncolored when round (j,i) begins. Then we say that y conflicts with x during the round if $tr(x,y) \leq g(x)$ and $tr(y,x) \leq 2$.*

LEMMA 4.8. *Let x be a string in U_S^* such that in some round the algorithm computes $fo(x) = 1$. Assume $c_{FO(x)} = c_k$. If $FO(x)$ is uncolored when round (j,k) begins, then x conflicts with $FO(x)$ during round (j,k).*

Proof. When the algorithm computes $fo(x) = 1$, $FO(x)$ is colored, $\mathcal{V}_S(FO(x)) \geq tr(FO(x),x)$, and $tr(x, FO(x)) \leq fo(x) + 1 = 2$. Assume $\mathcal{V}_S(FO(x)) = q$. Then in round (j,k), $g(FO(x)) \geq q \geq tr(FO(x),x)$ and $tr(x, FO(x)) \leq 2$, so x conflicts with $FO(x)$. ☐

The algorithm is complicated by the possible presence of strings $x \in U_S^*$ such that $st(x) = 1$ or 2; we handle these cases carefully and are not always able to do so seamlessly. These cases necessitate the *stopping conditions* in the first three rules of the algorithm.

DEFINITION 4.9. *Let c_i be a cycle in ${\mathcal{C}_S}^*$ and let x be a string in $OP(c_i)$. If c_i is uncolored when round (j,i) begins, we say that $g(x)$* is set by f *during the round, where $f = sh, sq, fo$, or st, according to the following rules:*

1. *If $j = 2$, $g(x) = st(x) = 2$, and $ST(x) > x$ is uncolored, then $g(x)$ is set by st.*
2. *Otherwise, f is the highest priority function such that $g(x) = f(x)$, where the functions are prioritized as sh, sq, fo, and st with sh having highest priority.*

We also may say that $g(x)$ is set by $f(x) = q$ to mean that $g(x)$ is set by f and $f(x) = q$.

In Figure 5 $G(w)$ should be read by replacing G with SH, SQ, FO, or ST where $g(w)$ is set by sh, sq, fo, or st. We include SH for convenience of notation but leave $SH(x)$ undefined. Any action on $G(w)$ by the algorithm is explicitly conditioned by the fact that $G(w)$ is defined.

DEFINITION 4.10. *The statement "*stopping condition *i holds" is logically equivalent to the following statements:*

1. *Stopping condition 1 holds $\leftrightarrow$ If $g(w)$ is set by $st(w) = 1$, then $ST(w) > w$.*
2. *Stopping condition 2 holds $\leftrightarrow$ If $g(w)$ is set by $st(w) = 2$, then $ST(w) > w$.*
3. *Stopping condition 3 holds $\leftrightarrow$ $g(w)$ is not set by $st(w) = 2$.*

The two lines of the algorithm marked by * rely on the definition of unmatched and on the choice of representative of *yellow* cycles; these lines affect only the choice of edges of W. We delay discussion until later and assume, for the moment, that the algorithm ignores these lines when it is executed. Under this assumption the algorithm clearly concludes. The next fact follows from the construction of Rule 4.

FACT 4.11. *Let c_i be a cycle of ${\mathcal{C}_S}^*$. Then c_i is colored by the algorithm no later than round $(2,i)$ of Rule 4.*

The next three lemmas describe important features of the coloring and $\mathcal{V}_S$. Proof of the last two are given in the appendix.

LEMMA 4.12. *Let x be a string in $OP(c_i)$ for some $c_i \in OP(c_i)$. Suppose c_i is uncolored when round (j,i) begins. If $g(x)$ is nonintegral, then $g(x)$ is set by $sh(x)$.*

Suppose c_i is uncolored when round $(2,i)$ begins. If $g(x)$ is set by $sh(x) = q$ (resp., $sq(x) = k$, $st(x) = k$) in round $(1,i)$, then $g(x)$ is set by $sh(x) = q$ (resp., $sq(x) = k$, $st(x) = k$) or $fo(x) < q$ (resp., $fo(x) < k$, $fo(x) \leq k$) in round $(2,i)$. If $g(x)$ is set by $sh(x) = q$ (resp., $sq(x) = k$, $st(x) = k$) in round $(2,i)$, then $g(x)$ is set by $sh(x) = q$ (resp., $sq(x) = k$, $st(x) = k$) in round $(1,i)$.

Proof. The first point follows from Corollary 3.8, Lemma 4.6, and our priority rule. The remaining points follow from our priority rule and the fact that for any $x \in U_S^*$ the functions $sh(x)$, $sq(x)$, and $st(x)$ are constant over the course of the algorithm. □

LEMMA 4.13. *Let c_i be a cycle in ${\mathcal{C}_S}^*$. Suppose c_i is colored in round (j,i). Let x be a string in $OP(c_i)$ where, if c_i is colored by Rules 1, 2, or 3, then x is the string by which the rule is chosen.*

1. *If $g(x)$ is set by sq, then $SQ(x)$ is colored no later than round (j,i). Furthermore, if $j = 2$, then $SQ(x)$ is colored before the round begins.*
2. *If $g(x)$ is set by fo, then $FO(x)$ is colored before round (j,i) begins.*
3. *If $g(x)$ is set by st, then $ST(x)$ is colored during round (j,i).*

LEMMA 4.14. *Let c_i be a cycle of ${\mathcal{C}_S}^*$. Suppose that c_i is colored in round $(2,i)$. Let x be a string in $OP(c_i)$ where, if c_i is colored by Rules 1, 2, or 3, then x is the string by which the rule is chosen.*

```
Construct the graph $\mathcal{G}(\mathcal{V}_S)$ and find a minimum-weight matching $M^*$.
    1. For each yellow $c$ set $rep(c)$ to $w \in OP(c)$ that determines the edge incident
       to $c$ in $M^*$. Add $e_{max}(rep(c), G(rep(c)))$ as required by part 1, to $W$.
    2. For each matched pair $(c, c')$ add $e_{max}(rep(c), rep(c'))$ to $W$.
```

FIG. 7. *Construction of the merging set W, part 2.*

1. *If c_i is colored by Rule 1, then $g(x)$ is set by $fo(x) = 1$ and $FO(x)$ is colored nonyellow (in a previous round).*
2. *If c_i is colored by Rule 2, then $g(x)$ is set by $fo(x) = 2$ and $FO(x) = ST(x)$.*
3. *If c_i is colored by Rule 3 we get a contradiction. This case cannot occur.*
4. *If c_i is colored by Rule 4, then $g(x) > \frac{3}{2}$. If $g(x)$ is set by sq or fo and $G(x)$ is yellow , then $|x| \leq \frac{5}{2}g(x)|c_i|$. If $ST(x)$ is defined and colored during the round, then $g(x)$ is set by $sh(x) < 2$ or by $st(x) = 2$.*

After constructing $\mathcal{V}_S$ the algorithm builds the graph $\mathcal{G}(\mathcal{V}_S)$ and finds a minimum-weight matching M^*; see part 2 in Figure 7. The purpose of this step is to select representatives for the *yellow* cycles and to identify *red* and *yellow* cycles that have large overlap edges between them. As described earlier $\mathcal{R}$ is the set of cycles colored *red* by the algorithm. Let $\mathcal{R}_1$ be the *red* cycles such that $\mathcal{V}_S(rep(c)) \leq 2$ and let $\mathcal{R}_2$ be the *red* cycles such that $\mathcal{V}_S(rep(c)) > 2$. Let $\mathcal{Y}$ be the set of cycles colored *yellow.* (Note that we drop the convention used in the discussion of part 1 that c_i refers to the ith cycle in the algorithm's order.)

DEFINITION 4.15. *Let c_1 and c_2 be distinct cycles of ${\mathcal{C}_S}^*$ in $\mathcal{R}_2 \cup \mathcal{Y}$. For $i = 1, 2$, let $X_i = \{rep(c_i)\}$ if $c_i \in \mathcal{R}_2$ and let $X_i = OP(c_i)$ otherwise. For $i = 1, 2$, let x_i be a string in X_i. Then $pair(x_1, x_2)$ is a cycle in $\mathcal{G}_S$ constructed as follows:*

1. *Set $pair(x_1, x_2)$ to the tight cycle $\mathcal{X}(\{c_1, c_2\}, e_{max}(x_1, x_2))$. Assume that e_i is the out-edge of $x_{i,last}$ in $pair(x_1, x_2)$.*
2. *For $i = 1, 2$, if c_i is red and $ov(e_i) > 0$, then replace e_i in $pair(x_1, x_2)$ by its maximum-overlap expansion $\hat{e}_i$ such that $ov(\hat{e}_i) < ov(e_i)$.*

Notice that if the edge e_i in the above definition has positive overlap, then the zero-overlap edge of e_i has overlap strictly less than e_i. Thus the replacement edge $\hat{e}_i$ always exists. Also, notice that $pair(\cdot, \cdot)$ is a symmetric function.

DEFINITION 4.16. *$\mathcal{G}(\mathcal{V}_S)$ is an undirected, weighted graph with vertex set ${\mathcal{C}_S}^*$. Each vertex c of the graph has a self-loop with weight $|c|$ if c is blue, $\max(\mathcal{V}_S(rep(c)), 2)|c|$ if c is red, and $\min_{x \in OP(c)} \mathcal{V}_S(x)|c|$ if c is yellow. Each pair $c_1 \neq c_2$, $c_i \in \mathcal{R}_2 \cup \mathcal{Y}$, $i = 1, 2$, shares an edge with weight $\min_{x_i \in X_i} |pair(x_1, x_2)|$, where $X_i = \{rep(c_i)\}$ if c_i is red and $X_i = OP(c_i)$ if c_i is yellow.*

After constructing $\mathcal{G}(\mathcal{V}_S)$ the algorithm finds a minimum-weight matching M^*. For vertices $c_1 \neq c_2$ in $\mathcal{G}(\mathcal{V}_S)$, the matching algorithm chooses the self-loops of c_1 and c_2 over the edge (c_1, c_2) in the event of a tie. Because of the self-loops each vertex c is incident to some edge in M^*; if the edge incident to c is a self-loop we say that c is *unmatched*; otherwise we say that c is *matched.* For each *yellow* cycle c the algorithm sets $rep(c)$ to an $x \in OP(c)$ that determines the weight of the edge incident to the vertex c in M^* subject to the following:

1. Suppose that c is *yellow*, unmatched; the self-loop incident to c in M^* has weight $2|c|$; and some $c' \neq c$ is colored in the same round as c in part 1. Then $c' = c_{ST(x)}$ for some $x \in OP(c)$. In this case the algorithm sets $rep(c_i)$ to such an x. (Note that by Lemma 4.14 $st(x) = 2$, so this is purely a tie-breaking rule.)

2. If (c, c') is in M^*, $c' \neq c$, then $rep(c)$ is chosen consistently with $rep(c')$ in the sense that the weight of the edge (c, c') in $\mathcal{G}(\mathcal{V}_S)$ is $|pair(rep(c_1), rep(c_2))|$.

The following fact is obvious from the construction. We will use this fact frequently in the next section.

FACT 4.17. *Let c be a matched cycle of ${\mathcal{C}_S}^*$. Then c is red or yellow. In the first case $\mathcal{V}_S(rep(c)) > 2$ and in the latter case $\mathcal{V}_S(rep(c)) > \frac{3}{2}$.*

Let (c_1, c_2) be an edge of M^*, $c_1 \neq c_2$. We call c_1 the max-*overlap cycle of the pair* if $|rep(c_1)| - \mathcal{V}_S(rep(c_1))|c_1| \geq |rep(c_2)| - \mathcal{V}_S(rep(c_2))|c_2|$, i.e., if the expansion of $e_{rep(c_1)}$ corresponding to $\mathcal{V}_S(rep(c_1))$ has overlap at least as large as the expansion of $e_{rep(c_2)}$ corresponding to $\mathcal{V}_S(rep(c_2))$. Otherwise we call c_2 the max-overlap cycle of the matched pair.

LEMMA 4.18. *Let M^* be a minimum-weight matching of $\mathcal{G}(\mathcal{V}_S)$. Suppose (c_1, c_2) is an edge of M^*, $c_1 \neq c_2$, where c_1 is the* max-*overlap cycle of the pair. Then $c_{x_2}^k$ exists for the integer $k = |pair(rep(c_1), rep(c_2))| - \mathcal{V}_S(rep(c_1))|c_1|$. Furthermore, if c_{x_2} is red , then $k \geq 2$.*

Proof. Assume that $rep(c_i) = x_i$, $i = 1, 2$. Since $\mathcal{V}_S$ is an expansion value function there exists an expansion of e_{x_i}, $i = 1, 2$, with pseudolength $\mathcal{V}_S(x_i)|c_i|$. Let $\hat{e}_i$, $i = 1, 2$, denote the expanded edge. Cycle c_1 is the max-overlap cycle of the pair, so $ov(\hat{e}_1) \geq ov(\hat{e}_2)$. Since $\mathcal{V}_S$ is a valid expansion value function, $\hat{e}_1$ is a winning edge for the cycle $pair(x_1, x_2)$. Then $\mathcal{X}(\{pair(x_1, x_2)\}, \hat{e}_1)$ consists of $c_{x_1}^{\mathcal{V}_S(x_1)}$ and a 1-expansion $\hat{c}_{x_2}$ of c_{x_2} of length at most $|pair(x_1, x_2)| - \mathcal{V}_S(x_1)|c_1|$. Next we show that equality holds. Suppose that $|\hat{c}_{x_2}| < |pair(x_1, x_2)| - \mathcal{V}_S(x_1)|c_1|$. By Lemma 3.3, the expansion of e_{x_2} in $\hat{c}_{x_2}$ has zero-overlap and thus $|\hat{c}_{x_2}| = |x_2|$. Since c_1 and c_2 are in $\mathcal{R}_2 \cup \mathcal{Y}$, the self-loops of c_1 and c_2 in $\mathcal{G}(\mathcal{V}_S)$ have total weight no more than

$$\mathcal{V}_S(x_1)|c_1| + \mathcal{V}_S(x_1)|c_2| \leq \mathcal{V}_S(x_1)|c_1| + sh(x_2)|c_2| \leq \mathcal{V}_S(x_1)|c_1| + |x_2| \leq |pair(x_1, x_2)|$$

which contradicts the matching algorithm's tie-breaking rule; i.e., self-loops are chosen over nonloop edges. Thus equality must hold. By the same argument, $|\hat{c}_{x_2}| < sh(x_2)|c_{x_2}|$ so by Corollary 3.8 $\hat{c}_{x_2} = c_{x_2}^k$ for some integer k.

Now suppose that c_{x_2} is *red*. By construction the out-edge of $x_{2,last}$ in $pair(x_1, x_2)$ has either zero-overlap or overlap strictly less than $ov(e_{x_2})$. In the first case the matching algorithm would choose the self-loops of c_1 and c_2. Thus $\hat{c}_{x_2} = c_{x_2}^k$ for some integer $k > 1$. ☐

DEFINITION 4.19. *The cover $\mathcal{C}_L$ consists of the following cycles:*

1. *If c is blue , then $c \in \mathcal{C}_L$.*
2. *If c is red and unmatched, then $c_{rep(c)}^q$ is in $\mathcal{C}_L$ where*

$$q = \min(|rep(c)|/|c|, \max(\mathcal{V}_S(rep(c)), 2)).$$

3. *If c is yellow and unmatched, then $c_{rep(c)}^{\mathcal{V}_S(rep(c)}$ is in $\mathcal{C}_L$.*
4. *If $c_1 \neq c_2$ are matched, where (c_1, c_2) is an edge in M^* and c_1 is the* max-*overlap cycle of the pair, then $c_{rep(c_1)}^{\mathcal{V}_S(rep(c_1))}$ and the 1-expansion of $c_{rep(c_2)}$ with length $|pair(rep(c_1), rep(c_2))| - \mathcal{V}_S(rep(c_1))|c_1|$ are in $\mathcal{C}_L$.*

The existence of these cycles is guaranteed by Lemma 4.6, Corollary 3.8, and Lemma 4.18. In section 6 we prove the following lemma.

LEMMA 4.20. $|\mathcal{C}_L| - \sum_{c \in \mathcal{R}} |c| \leq |OPT_S|$.

The selection of the edges W is shown in Figures 6 and 7. Several choices in part 1 (i.e., the lines marked by *) depend on the outcome of the matching and on representatives selected in part 2. We assume that the algorithm records the current

TABLE 1
Summary of alloc(c).

Color of c	c Matched ?	Allocation
blue	N/A	$\frac{5}{2}\|c\|$
red	Yes	$\|m(c)\| - \frac{1}{2}\|c\|$
	No	$\frac{5}{2}(\max(\mathcal{V}_S(rep(c)), 2) - 1)\|c\|$
yellow	Yes	$\|m(c)\|$
	No	$\frac{5}{2}\mathcal{V}_S(rep(c))\|c\|$

status when these lines are invoked and then in part 2 goes back and adds the necessary edges to W. In part 2 the algorithm also adds edges for matched cycles to W.

The proof of Theorem 1.1 follows from Lemma 4.20 and the next lemma which we prove in section 5.

LEMMA 4.21. *There exists a cover $\mathcal{C}_U$ of $\mathcal{G}_S$ such that $|AOPT_S| \leq |\mathcal{C}_U| \leq \frac{5}{2}(|\mathcal{C}_L| - \sum_{c\in\mathcal{R}} |c|)$.*

5. The upper bound cover $\mathcal{C}_U$.

To prove Lemma 4.21, we explicitly construct $\mathcal{C}_U$. We begin with $\mathcal{C}_U$ and W empty and simulate the construction of W by the algorithm. In a round of part 1, where the algorithm colors a cycle $c \in \mathcal{C}_S{}^*$, we add an expansion of c, denoted $m(c)$, to $\mathcal{C}_U$. We may also *expand* an existing cycle in $\mathcal{C}_U$, i.e., replace some edge of the cycle by one of its expansions.

LEMMA 5.1. *At each point of the construction, $\mathcal{C}_U$ and W satisfy the following invariants:*

1. *$\mathcal{C}_U$ is a cycle cover of the graph $G_{S'}$, where $S' = \{s \in S \mid s$ is a vertex of a colored $c \in \mathcal{C}_S{}^*\}$. Furthermore, for any colored $c \in \mathcal{C}_S{}^*$, $\mathcal{C}_U$ contains an expansion of $m(c)$.*
2. *Every weakly connected component of the subgraph $\mathcal{C}_U \cup W$ of $\mathcal{G}_S$, that is, every connected component where the edges are considered in an undirected sense, contains a zero-overlap edge or an expansion of $m(c)$, $c \in \mathcal{C}_S{}^*$, such that c is matched.*
3. *Every edge of W is a winning edge for $\mathcal{C}_U$.*
4. *$|\mathcal{C}_U| \leq \sum_{\text{colored } c} alloc(c)$, where $alloc(\cdot)$ is defined in Table 1.*

For any $(c, c') \in M^*$, $c \neq c'$, the algorithm adds $e_{max}(rep(c), rep(c'))$ to W in part 2. Thus, the next fact and invariant 2 imply that when the construction concludes, every weakly connected component of $\mathcal{C}_U \cup W$ has a zero-overlap edge.

FACT 5.2. *Suppose (c, c') is an edge of M^*, where $c \neq c'$. Then either $m(c)$ or $m(c')$ has a zero-overlap edge.*

Thus Lemma 5.1 together with the next lemma implies that $|AOPT_S| \leq |\mathcal{C}_U|$.

LEMMA 5.3. *Let $\tilde{\mathcal{C}}$ be an expansion of $\mathcal{C}_S{}^*$ and let W be the set of merging edges constructed by the algorithm. If every edge of W is a winning edge for $\tilde{\mathcal{C}}$ and every weakly connected component of the subgraph $\tilde{\mathcal{C}} \cup W$ has a zero-overlap edge, then $|AOPT_S| \leq |\tilde{\mathcal{C}}|$.*

When the construction concludes, invariant 4 implies that $|\mathcal{C}_U| \leq \sum_{c\in\mathcal{C}_S{}^*} alloc(c)$. By the next lemma, the length of $\mathcal{C}_U$ satisfies the bound of Lemma 4.21.

LEMMA 5.4. *For $alloc(c), c \in \mathcal{C}_S{}^*$, defined in Table 1,*

$$\sum_{c\in\mathcal{C}_S{}^*} alloc(c) \leq \frac{5}{2}\left(|\mathcal{C}_L| - \sum_{c\in\mathcal{R}} |c|\right).$$

Table 2
Construction of $m(c)$.

Case				Construction			
$g(w)$ set by	c_w matched	c_z colored in curr round	$e_{max}(w,z)$ added to W	$m(c_w)$	$m(c_u)$ Rule 2	$m(c_z)$	$m(c_v)$ Rule 4
sh	no	NA	NA	w	c_u^2	NA	c_v^2
	yes	NA	NA	See Tbl. 3	c_u^2	NA	c_v^2
sq	no	no	no	w	c_u^2	NC	c_v^2
	no	no	yes	$c_w^{sq(w)}$	c_u^2	Expand $m(c_z)$	c_v^2
	no	yes	yes	$c_w^{sq(w)}$	c_u^2	z	c_v^2
	yes	no	no	See Tbl. 3	c_u^2	NC	c_v^2
	yes	yes	yes	See Tbl. 3	c_u^2	c_z^2	c_v^2
fo	no	no	no	w	c_u^2	NC	c_v^2
	no	no	yes	$c_w^{fo(w)+1}$	c_u^2	NC	c_v^2
	yes	no	no	See Tbl. 3	c_u^2	NC	c_v^2
st	no	yes	yes	w	c_u^2	c_z	c_v^2
	yes	yes	yes	See Tbl. 3	c_u^2	c_z	c_v^2

Table 3
Construction for matched cycles (c_x, c_y), where c_x is the max-*overlap cycle of the pair and $x = rep(c_x)$, $y = rep(c_y)$.*

Color of c_x	Expansion Value of x	$m(c_x)$	$m(c_y)$
yellow	$\mathcal{V}_S(x) < 2$	x	y
	$\mathcal{V}_S(x) \geq 2$	$c_x^{\mathcal{V}_S(x)+2}$	
red	$\mathcal{V}_S(x) < 3$	$c_x^{\mathcal{V}_S(x)+\frac{\lvert c_y\rvert}{\lvert c_x\rvert}}$	
	$\mathcal{V}_S(x) \geq 3$	$c_x^{\mathcal{V}_S(x)+1}$	

Thus, to complete the proof of Lemma 4.21 it suffices to demonstrate Fact 5.2 and prove Lemmas 5.1, 5.3, and 5.4. The proofs of Lemmas 5.1 and 5.4 are very technical; they are given in the appendix. The proof of Lemma 5.3 is given at the end of this section.

Next we describe the construction in detail. The construction of $m(c)$ is given in Table 2, with reference to Table 3. For ease of notation throughout this section, we use c_x^q to mean $c_x^{\min(q,|x|/|c_x|)}$ for $x \in U_S^*$. With some abuse of notation in these tables we also let x, $x \in U_S^*$, denote the 1-expansion of c_x with length $|x|$, i.e., the 1-expansion of c_x that includes the zero-overlap version of e_x. We refer to this policy as our naming convention. Fact 5.2 is obvious from Table 3. NC and NA in Table 2 mean *no change* and *not applicable* .

We now describe the construction in round (j,i). If c_i is not colored in the round we do not change $\mathcal{C}_U$. So assume c_i is colored in round (j,i). If c_i is colored by Rule 1, 2, or 3, let w be the string by which the rule is chosen. If c_i is colored by Rule 4, let $w = rep(c_i)$. Thus $c_i = c_w$. If c_w is colored by Rule 2 then c_u, where u conflicts with w, is also colored in the round. If $G(w)$ is defined, then for convenience of notation we let z denote $G(w)$. If c_w is colored by Rule 4 and a cycle c, $c \neq c_w$ and $c \neq c_z$, and is also colored in round (j,i), then $j = 2$ and $c = c_{ST(x)}$ for some $x \in OP(c_w), x \neq w$. Let v denote such a string $ST(x)$. There may be several such

v in a round corresponding to different $x \in OP(c_w)$; the construction is identical for each.

We first describe the construction of $m(c_w)$. Suppose $g(w)$ is set by sh. The construction of $m(c_w)$, as shown in Table 2, depends on whether c_w is unmatched or matched. If c_w is unmatched then $m(c) = w$, where w denotes the 1-expansion of c_w of length $|w|$ as described earlier. If c_w is matched, the construction of $m(c_w)$ is given in Table 3. It depends on whether or not c_w is the winning cycle of the matched pair, on the color of c_w, and on the value of $\mathcal{V}_S(w)$.

Now suppose that $g(w)$ is set by sq, fo, or st. Then z is defined. In this case $m(c_w)$ depends on whether c_w is unmatched or matched, whether c_z is colored in the current round, and whether $e_{max}(w, z)$ is added to W. (Note that by Lemma 4.13 c_z is colored in either round (j, i) or in an earlier round.) Thus there are eight possible cases, but not all of them can occur. Specifically, if c_z is colored in round (j, i), then the algorithm always adds $e_{max}(w, z)$ to W. Also, if c_w is matched, and thus colored in Rules 2–4, and c_z is colored in a previous round, $e_{max}(w, z)$ is never added to W. The remaining five cases can occur when $g(w)$ is set by sq, and Table 2 handles each possibility. If $g(w)$ is set by fo, then by Lemma 4.13 c_z is colored in a previous round so there are only three possible cases. If $g(w)$ is set by st, then by Lemma 4.13 c_z is colored in the current round and so there are only two possible cases. Table 2 handles every possibility, with reference to Table 3 when c_w is matched.

When u is defined, Table 2 gives the construction of $m(c_u)$. When z is defined, the column labeled $m(c_z)$ gives the new construction for $m(c_z)$ for the cases where c_z is colored in round (j, i). If c_z is colored in a previous round, then $\mathcal{C}_U$ already contains an expansion of $m(c_z)$ and the table describes any necessary modifications to this cycle; expanding this cycle by $|c_w|$ should be interpreted as replacing the version $\hat{e}_z$ of e_z in the cycle by its expansion with overlap $\max(ov(\hat{e}_z) - |c_w|, 0)$. Finally, if v is defined, the column labeled $m(c_v)$ gives the appropriate construction.

Proof of Lemma 5.3. Let the edge of W ordered by increasing overlap be $e_1, \ldots, e_k$. We define a sequence of cycle covers $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ of $\mathcal{G}_S$ as follows. $\mathcal{C}_0 = \tilde{\mathcal{C}}$. Assume $e_i = \langle s, t \rangle$. If s and t are in the same cycle of $\mathcal{C}_{i-1}$, then $\mathcal{C}_i = \mathcal{C}_{i-1}$ and otherwise $\mathcal{C}_i = \tilde{\mathcal{X}}(\mathcal{C}_{i-1}, e_i)$. We claim that $|\mathcal{C}_i| \leq |\mathcal{C}_{i+1}|$, every edge in $\{e_{i+1}, \ldots, e_k\}$ is a winning edge for $\mathcal{C}_i$, and every weakly connected component of $\mathcal{C}_i \cup \{e_{i+1}, \ldots, e_k\}$ has a zero-overlap edge.

The claim is true by assumption when the construction begins. Suppose it is true for $\mathcal{C}_{i-1}$, $i \geq 1$, and assume $e_i = \langle s, t \rangle$. If s and t are in the same cycle in $\mathcal{C}_{i-1}$, then $\mathcal{C}_i = \mathcal{C}_{i-1}$ and the claim holds. Now suppose s and t are in different cycles. Assume these cycles are, respectively, c_s and c_t. Then $\mathcal{C}_i = \tilde{\mathcal{X}}(\mathcal{C}_{i-1}, e_i) = \mathcal{C}_i - \{c_s, c_t\} \cup \tilde{\mathcal{X}}(\{c_s, c_t\}, e_i)$. Since e_i is a winning edge for $\mathcal{C}_{i-1}$, $|\mathcal{C}_i| \leq |\mathcal{C}_{i-1}|$. If e_j, $j > i$, is a winning edge for $\mathcal{C}_{i-1}$ but not for $\mathcal{C}_i$, then by Lemma 3.3 e_j must be an out-edge of s or an in-edge of t. However, $ov(e_j) < ov(e_i)$, which contradicts the fact that the edges are ordered by increasing overlap. Thus each edge of $\{e_{i+1}, \ldots, e_k\}$ is a winning edge for $\mathcal{C}_i$. Finally, consider the last point. The only edges of $\mathcal{C}_{i-1} \cup \{e_i, \ldots, e_k\}$ that are not in $\mathcal{C}_i \cup \{e_{i+1}, \ldots, e_k\}$ are the edges removed by the edge exchange. If either of these edges has zero-overlap, then by Lemma 3.3 the losing edge of the exchange does as well. Thus the component of $\mathcal{C}_i$ containing the cycle $\tilde{\mathcal{X}}(\{c_s, c_t\}, e_i)$ includes a zero-overlap edge. The remaining components satisfy the claim by the inductive hypothesis. The final cover $\mathcal{C}_k$ has the property that $|\mathcal{C}_k| \leq |\tilde{\mathcal{C}}|$ and furthermore, each cycle c of $\mathcal{C}_k$ has a zero-overlap edge so $|op(c)| = |c|$.

Notice that our construction simulates step 4 of the algorithm (see Figure 5) but

```
Construction Protocol
1. Color the edges of $\mathcal{C}_S{}^*$ *white* and the remaining edges of $\mathcal{G}_S$ *black*.
2. Initialize $\mathcal{C}_0$ to a Hamiltonian cycle of $\mathcal{G}_s$ and set $i = 0$.
3. Each cycle of $c \in \mathcal{C}_S{}^*$ is initially *unmarked*.
4. While $\mathcal{C}_i$ contains a mixed cycle.
    (i) **Mark:** If $c$ is unmarked and represented in a mixed cycle in $\mathcal{C}_i$, then
        mark $c$ "represented." Recolor the missing edge $e_x$ *black* and its expansion
            $\tilde{e}_x$ with pseudolength $\mathcal{V}_S(x)|c_x|$ *white*.
    (ii) **Edge exchange:** Let $e_W$ be the maximum overlap *white* edge of $\mathcal{G}_S$ such
        that (1) $e_W$ is not in $\mathcal{C}_i$, and (2) $\mathcal{C}_i$ is not uniform for $c(e_W)$.
            Set $\mathcal{C}_{i+1} = \tilde{\mathcal{X}}(\mathcal{C}_i, e_W)$ and $i = i + 1$.
    (iii) **Eliminate-uniform-cycle:** While $\mathcal{C}_i$ includes a mixed cycle $\hat{c}_{mixed}$ with
        a vertex $s$ and a uniform cycle $\hat{c}_{uniform}$ with the vertex $next(s)$ such that
        $|join(\hat{c}_{uniform}, \hat{c}_{mixed}, s)| \leq |\hat{c}_{uniform}| + |\hat{c}_{mixed}|$
        Set $\mathcal{C}_{i+1} = \mathcal{C}_i - \{\hat{c}_{uniform}, \hat{c}_{mixed}\} \cup \{join(\hat{c}_{uniform}, \hat{c}_{mixed}, s)\}$ and $i = i+1$.
    (iv) **Reorder-path:** While $\mathcal{C}_i$ includes a mixed cycle $\hat{c}_{mixed}$ such that for some
        $c \in \mathcal{C}_S{}^*$, the vertices of $c$ comprise a path in $\hat{c}_{mixed}$, $c$ is not represented in
        $\hat{c}_{mixed}$, and $|represent(\hat{c}_{mixed}, c)| \leq |\hat{c}_{mixed}|$
            Set $\mathcal{C}_{i+1} = \mathcal{C}_i - \{\hat{c}_{mixed}\} \cup \{represent(\hat{c}_{mixed}, c)\}$ and $i = i + 1$.
```

FIG. 8.

beginning with an expansion of $\mathcal{C}_S{}^*$, i.e., $\tilde{\mathcal{C}}$, and using parsimonious edge exchanges. At each point our current cover $\mathcal{C}_i$ is an expansion of the algorithm's current cover $\mathcal{C}$. When the construction concludes,

$$|AOPT_S| = \sum_{c \in \mathcal{C}} op(c) \leq \sum_{c \in \mathcal{C}_k} op(c) = \sum_{c \in \mathcal{C}_k} |c| = |\mathcal{C}_k| \leq |\tilde{\mathcal{C}}|. \qquad \square$$

6. The lower bound cover.

In this section we prove Lemma 4.20. Our strategy is similar to the proof of Theorem 2.7 given at the end of section 2. The construction protocol for the proof of Lemma 4.20 is given in Figure 6. We construct a series of covers of $\mathcal{G}_S$ that are nonincreasing in length beginning with a Hamiltonian cycle $\mathcal{C}_0$. After we've described the general construction, we choose $\mathcal{C}_0$ so as to get the desired bound of Lemma 4.20.

We begin by coloring the edges of $\mathcal{C}_S{}^*$ *white* and the remaining edges of $\mathcal{G}_S$ *black*. At each point in the protocol, $\mathcal{C}_W$ denotes the set of white edges of $\mathcal{G}_S$. We may recolor edges during the protocol but we'll see that the construction satisfies the following invariant.

FACT 6.1. *$\mathcal{C}_W$ is always an expansion of $\mathcal{C}_S{}^*$. For any $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ constructed by the protocol,*

1. *if e is a* white *edge in $\mathcal{C}_i$, then e is a* white *edge of $\mathcal{C}_{i+1}$, and*
2. *$\mathcal{C}_{i+1}$ has strictly more* white *edges than $\mathcal{C}_i$.*

At any point in the protocol the *current cover* is the cover $\mathcal{C}_i$ for the current value of i. The protocol continues as long as the current cover includes a *mixed* cycle. A cycle $\hat{c}$ of $\mathcal{G}_S$ is *mixed* if it includes vertices s and t such that $c_s \neq c_t$, where c_v denotes the cycle of $\mathcal{C}_S$ containing the vertex v. A cycle $\hat{c}$ of $\mathcal{G}_S$ that is not mixed is *uniform*. If $\hat{c}$ is uniform, then for some $c \in \mathcal{C}_s$ every vertex of $\hat{c}$ is also a vertex of c and we may also say that $\hat{c}$ is uniform for c. The cycle $\hat{c}$ need not include every vertex of c and need not include any edges of c. A cycle cover $\mathcal{C}$ of $\mathcal{G}_S$ is uniform for $c \in \mathcal{C}_S{}^*$ if every vertex of c is in a cycle uniform for c in $\mathcal{C}$. By our earlier assumption $\mathcal{C}_S{}^*$ consists of more than one cycle so the Hamiltonian cycle $\mathcal{C}_0$ is mixed. Fact 6.1 guarantees that the protocol concludes. The final cover of the protocol, which we denote as $\mathcal{C}_{final}$, is uniform for every $c \in \mathcal{C}_S{}^*$.

Step (i) of the protocol marks cycles of $\mathcal{C}_S{}^*$ that are *represented* in a mixed cycle of the current cover. We say that a cycle $\hat{c}$ of $\mathcal{G}_S$ *represents* $x \in U_S^*$ if $order(c_x, x_{first})$ is a tight path of $\hat{c}$; here $order(c, s)$, for a cycle $c \in \mathcal{C}_S{}^*$ and a vertex $s \in c$, is a list of the vertices of c in the order they occur in c beginning at vertex s. If a cycle $\hat{c}$ of $\mathcal{G}_S$ represents $x \in U_S^*$ we also say it represents c_x. Note that if a cycle $\hat{c}$ of $\mathcal{G}_S$ represents $c \in \mathcal{C}_S{}^*$ and $\hat{c} \neq c$, then $\hat{c}$ represents a unique $x \in OP(c)$.

Let c be a cycle of $\mathcal{C}_S{}^*$. Initially c is unmarked. In step (i) we mark c "represented" if c is still unmarked and some mixed cycle of the current cover represents c. The protocol maintains the following invariant.

FACT 6.2. *Let c be a cycle of $\mathcal{C}_S{}^*$. If c is unmarked at some point in the protocol, then $c \in \mathcal{C}_W$. If c is marked "represented," then $c_x^{\mathcal{V}_S(x)} \in \mathcal{C}_W$ for some $x \in OP(c)$.*

Suppose we mark c in some round. Let $\hat{c}$ be the cycle of the current cover that represents c. Since $\hat{c}$ is mixed (and hence $\hat{c} \neq c$), $\hat{c}$ represents a unique $x \in OP(c)$. Since $c \in \mathcal{C}_W$ when the step begins, by Fact 6.2 e_x is *white*. When we mark c in step (i) we also recolor e_x *black* and recolor the expansion of e_x with pseudolength $\mathcal{V}_S(x)|c|$ *white*. Since $\hat{c}$ is a mixed cycle and represents c, the current cover does not contain e_x or any of its expansions. Thus the recoloring has no effect on the current cover. After the recoloring, $c_x^{\mathcal{V}_S(x)} \in \mathcal{C}_W$. By Facts 6.1 and 6.2 if $\mathcal{C}_i$ represents $x \in U_S^*$, then $\mathcal{C}_{i+1}$ represents x.

In step (ii) the protocol performs an edge exchange based on a maximum overlap *white* edge e_W that satisfies two conditions. The first condition is that e_W is not in the current cover. By Fact 6.1 e_W is an expansion of an edge of some $c \in \mathcal{C}_S{}^*$; we use $c(e_W)$ to denote the cycle c. The second condition is that the current cover is not uniform for $c(e_W)$.

LEMMA 6.3. *Suppose the protocol constructs $\mathcal{C}_{i+1} = \tilde{\mathcal{X}}(\mathcal{C}_i, e_W)$ in some round. Then e_W is a winning edge for $\mathcal{C}_i$.*

Proof. Assume that $e_W = \langle s, t, k\rangle$, the out-edge of s in the current cover is $e = \langle s, u, j\rangle$, and the *white* in-edge of u is $e' = \langle v, u, \ell\rangle$. Suppose that $c_s = c_u$, where c_v is the cycle of $\mathcal{C}_S{}^*$ containing the vertex v. Since $\langle s, u\rangle$ is not an edge of $\mathcal{C}_S{}^*$, c_s is not represented in the current cover, c_s is not marked "represented," and e_W and e' are tight edges. Since $k \geq \ell$, by Fact 2.16 $k \geq j$. Now suppose that $c_s \neq c_u$. Then s and u are x_{last} and y_{first} for some $x, y \in U_S^*$, $c_x \neq c_y$. The edge e_W is either tight or has pseudolength $\mathcal{V}_S(x)|c_x|$; in either case $k \geq |x| - \mathcal{V}_S(x)|c_x|$. By a symmetric argument $\ell \geq |y| - \mathcal{V}_S(y)|c_y|$. Since $k \geq \ell$, $k \geq \max(|x| - \mathcal{V}_S(x)|c_x|, |y| - \mathcal{V}_S(y)|c_y|)$. By Lemma 4.6 $\mathcal{V}_S$ is a valid expansion value function so $k \geq \max(|x| - \mathcal{V}_S(x)|c_x|, |y| - \mathcal{V}_S(y)|c_y|) \geq ov_{max}(x, y) \geq ov(s, t') = j$.

By a symmetric argument k is at least as large as the overlap of the in-edge of t in the current cover. Thus e_W is a winning edge for the current cover. □

The previous proof relies on the fact that if the edge $e_W = \langle s, t, k\rangle$ is loose, then for some $x \in OP(c_s)$, $s = x_{last}$, $t = x_{first}$, the out-edge of s in the current cover is to some u such that $c_u \neq c_s$, and a symmetric statement holds for the in-edge of t.

These conditions hold when $c(e_W)$ is represented in a mixed cycle. In steps (iii) and (iv) we attempt to force the representation of each $c \in \mathcal{C}_S{}^*$ in a mixed cycle at some point in the protocol. We can guarantee this only for cycles in $\mathcal{C}_S{}^*$ with *depth* at least 3; the depth of c of $\mathcal{G}_S$, denoted d_c, is defined as $d_c = |op(c)|/|c|$. We'll see, however, that this is enough. In the next lemma, $len_{final}(c)$ is the total length of the cycles uniform for $c \in \mathcal{C}_S{}^*$ in the final cover $\mathcal{C}_{final}$.

LEMMA 6.4. *Let c be a cycle of $\mathcal{C}_S{}^*$. If c is not marked "represented" when the protocol concludes, then $d_c < 3$ and $len_{final}(c) \geq sh(x)|c|$ for some $x \in OP(c)$.*

The proof of this lemma is given in the appendix. Next we describe steps (iii) and (iv). Let c be a cycle of $\mathcal{C}_S{}^*$ and suppose that when step (iii) begins, the current cover $\mathcal{C}_i$ contains a cycle uniform for c and a mixed cycle containing a vertex of c. In this case there must be vertices s and t of c such that $\langle s, t\rangle$ is an edge of c, s is in a mixed cycle of $\mathcal{C}_i$, and t is in a cycle uniform for c in $\mathcal{C}_i$. In step (iii) we insert the vertices of the uniform cycle into the mixed cycle provided we can do so without increasing cover length. We use $next(s)$, $s \in S$, to denote the vertex t (possibly $s = t$) such that $\langle s, t\rangle$ is in $\mathcal{C}_S{}^*$.

DEFINITION 6.5. *Let c be a cycle of $\mathcal{C}_S{}^*$, let*

$$c_{uniform} = \langle t_0, t_1, k_0\rangle, \ldots, \langle t_j, t_0, k_j\rangle$$

be a cycle of $\mathcal{G}_S$ that is uniform for c, and let

$$c_{mixed} = \langle s_0, s_1, \ell_0\rangle, \ldots, \langle s_m, s_0, \ell_m\rangle$$

be a mixed cycle of $\mathcal{G}_S$ that includes a vertex s such that $next(s)$ is in $c_{uniform}$. Assume $s = s_0$ and $next(s) = t_0$. The cycle $join(c_{uniform}, c_{mixed}, s)$ is defined as

$$\langle s_0, t_0\rangle, \langle t_0, t_{i_1}\rangle, \ldots, \langle t_{i_{m-1}, t_{i_m}}\rangle, \langle t_{i_m}, s_1, \ell'\rangle, \langle s_1, s_2, \ell_2\rangle, \ldots, \langle s_m, s_0, \ell_m\rangle,$$

where the vertices of $c_{uniform}$ occur in c in order $t_0, t_{i_1}, \ldots, t_{i_j}$ and ℓ' is the maximum overlap of any edge from t_{i_m} to s_1 such that $\ell' \leq \ell_0$.

Let c be a cycle of $\mathcal{C}_S{}^*$. Suppose at the beginning of step (iv) the vertices of c comprise a path in some mixed cycle of the current cover. If c is not represented by $\hat{c}$, then the vertices of c may be out of order or the path may include some loose edges. If so, we reorganize the path so that c is represented provided we can do so without increasing cover length.

DEFINITION 6.6. *Let c be a cycle of $\mathcal{C}_S{}^*$ and let*

$$c_{mixed} = \langle s_0, s_1, k_0\rangle, \langle s_1, s_2, k_1\rangle, \ldots, \langle s_j, t_0, k_j\rangle\langle t_0, t_1, \ell_0\rangle, \ldots, \langle t_m, s_0, \ell_m\rangle$$

be a mixed cycle such that $s_0, \ldots, s_j$ are the vertices of c. Assume that $order(c, s_0) = s_0, s_{i_1}, s_{i_2}, \ldots, s_{i_j}$. The cycle represent$(c_{mixed}, c)$ *is defined as*

$$\langle s_0, s_{i_1}\rangle, \langle s_{i_1}, s_{i_2}\rangle, \ldots, \langle s_{i_{j-1}}, s_{i_j}\rangle, \langle s_{i_j}, t_0, k'\rangle, \langle t_0, t_1, \ell_1\rangle, \ldots, \langle t_m, s_0, \ell_m\rangle,$$

where k' is the maximum overlap of any edge from s_{i_j} to t_0 such that $k' \leq k_j$.

We can assume, without loss of generality, that if some $c \in \mathcal{C}_0$ satisfies the conditions of step (iv), then c is represented in $\mathcal{C}_0$.

LEMMA 6.7. *Let c be a nonblue cycle of $\mathcal{C}_S{}^*$ such that $len_{final}(c) < \mathcal{V}_S(x)|c|$ for every $x \in OP(c)$. Then for some covers $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ constructed by the protocol*

$$\mathcal{C}_{i+1} = \tilde{\mathcal{X}}(\mathcal{C}_i, \hat{e}_y) = \mathcal{C}_i - \{\hat{c}\} \cup \{c_x^q, c_y^{\mathcal{V}_S(y)}\},$$

where $x \in OP(c)$, $y \in U_S^*$, $c_y \neq c$, $\hat{e}_y$ is a strict expansion of e_y with pseudolength $\mathcal{V}_S(y)|c_y|$, $\hat{c}$ is some expansion of $\mathcal{X}(\{c_x, c_y\}, e_{max}(x,y))$, and $q|c| = len_{final}(c)$.

Proof. Let $\mathcal{C}_i$ be the first cover that is uniform for c. By Lemma 6.4 $d_c \geq 3$ and c is represented in a mixed cycle $\hat{c}$ in $\mathcal{C}_{i-1}$. Assume $\hat{c}$ represents $x \in OP(c)$. The only step that can create a uniform cycle is the edge exchange so

$$\mathcal{C}_i = \tilde{\mathcal{X}}(\mathcal{C}_{i-1}, e) = \mathcal{C}_{i-1} - \{\hat{c}\} \cup \{c_1, c_2\},$$

where c_2 is a 1-expansion c_x^q with length $len_{final}(c)$. If e is the expansion of e_x with length $\mathcal{V}_S(x)|c|$, then $c_2 = c_x^{\mathcal{V}_S(x)}$, which is a contradiction. Thus e is an edge from y_{last} to y_{first} for some $y \in U_S^*$, $c_y \neq c$ and the losing edge of the exchange $\hat{e}_x$ is an expansion of e_x. If $ov(\hat{e}_x) < |x| - sh(x)|c|$, then $|c_2| \geq sh(x)|c| \geq \mathcal{V}_S(x)|c|$, which is a contradiction. So assume that $ov(\hat{e}_x) \geq |x| - sh(x)|c|$. If e is tight, then $ov(\hat{e}_x) \leq |x| - sq(x)|c|$ so $|\hat{c}_x| \geq sq(x)|c| \geq \mathcal{V}_S(x)|c|$, which is a contradiction. Thus e is a loose edge. By Fact 6.2 $\hat{c}$ also represents y. Since $\hat{c}$ must also include edges from x_{last} to y_{first} and from y_{last} to x_{first}, $\hat{c}$ is an expansion of $\mathcal{X}(\{c_x, c_y\}, e_{max}(x,y))$. □

Proof of Lemma 4.20. To prove this theorem we construct a cover $\tilde{\mathcal{C}}_L$ such that

$$|\mathcal{C}_L| \leq |\tilde{\mathcal{C}}_L| \leq OPT_S + \sum_{c \in \mathcal{R}} |c|.$$

First consider the case where $\mathcal{R}$ is empty. Set $\mathcal{C}_0 = \{c_{\pi_S^*}\}$ and construct $\mathcal{C}_{final}$ by the construction protocol. By construction and Lemma 6.3 $|\mathcal{C}_{final}| \leq |c_{\pi_S^*}| \leq OPT_S$. For each $c \in \mathcal{C}_S{}^*$ such that $\mathcal{C}_{final}$ does not include a 1-expansion of c, replace the cycles uniform for c by the 1-expansion $c_x^{sh(x)}$ for the x defined by Lemma 6.4; these modifications do not increase the length of $\mathcal{C}_{final}$. Let $\tilde{\mathcal{C}}_L = \mathcal{C}_{final}$. Lemma 6.7 implies a matching between *yellow* cycles c_x and c_y such that $len_{final}(c_x) < \mathcal{V}_S(x)|c_x|$ and $len_{final}(c_x) + len_{final}(c_y) \geq pair(x,y)$. For any remaining cycle c, $len_f(c) \geq \mathcal{V}_S(x)|c|$ for some $x \in OP(c)$. Thus $\tilde{\mathcal{C}}_L$ corresponds to a matching of $\mathcal{G}(\mathcal{V}_S)$ with length at most $|\tilde{\mathcal{C}}_L| \leq |OPT_S|$. Our claim then follows since $|\mathcal{C}_L|$ is the weight of a minimum-weight matching M^* of $\mathcal{G}(\mathcal{V}_S)$.

Now consider the case where $\mathcal{R}_1$ is empty. We begin by setting $\mathcal{C}_0 = c_{\pi_S^*}$. For each $c \in \mathcal{R}_2$ do the following. Delete the vertices of c, except for $rep(c)_{first}$, from $\mathcal{C}_0$ using Lemma 2.3. Duplicate the vertex $rep(c)_{first}$ and insert the deleted vertices between the two copies in the order the vertices occur in c using tight edges. Delete the second copy of $rep(c)_{first}$. When the construction concludes, $|\mathcal{C}_0| \leq |OPT_S| + \sum_{c \in \mathcal{R}_2} |c|$. Also by the construction, for any $c \in \mathcal{R}_2$, the out-edge of $rep(c)_{last}$ in $\mathcal{C}_0$ is either zero or strictly less than $ov(rep(c)_{last}, rep(c)_{first})$. Construct $\mathcal{C}_{final}$ as above. The same arguments hold for this case except for matched pairs where one or both of c_x and c_y are *red*. To prove this case it is sufficient to show that the cycle $\hat{c}_{mixed}$ which represents x and y in the proof of Lemma 6.7 is an expansion of $pair(x,y)$. By Lemma 3.3 and the construction of steps (iii) and (iv) the overlap of the out-edge of x_{last} (resp., y_{last}) is nonincreasing over the construction, so the claim holds.

Now suppose that $\mathcal{R}_1$ is nonempty. Construct $\mathcal{C}_0$ as in the last case. We will treat $c \in \mathcal{R}_1$ as a *blue* cycle in the construction, i.e., set $\mathcal{V}_S(x) = 1$ for every $x \in OP(c)$. Construct $\tilde{\mathcal{C}}_L$ as in the previous case. Then for each $c \in \mathcal{R}_1$, replace the expansion of c in $\tilde{\mathcal{C}}_L$ by the cycle c_x^2 where $x = rep(c)$. Then $|\tilde{\mathcal{C}}_L| \leq |OPT_S| + \sum_{c \in \mathcal{R}} |c|$ and, as above, $\tilde{\mathcal{C}}_L$ corresponds to a matching of $\mathcal{G}(\mathcal{V}_S)$. □

7. Conclusions. Probably the most interesting open question in superstring study is whether GREEDY yields a 2-approximation. GREEDY has an (almost) obvious interpretation relative to the protocol we described in section 6. At each point in its operation, the strings in GREEDY's current set defines a collection of simple, disjoint paths in G_S. When GREEDY merges x and y (i.e., replaces them by $pref(x, y) \cdot y$) we perform an edge exchange based on the edge from the end of x's path to the beginning of y's path. The difficulty with analyzing GREEDY in this way is that these edges are not always winning edges and thus we must account for the cost increases along the way. The construction protocol in section 6 may be useful in analyzing these increases. In addition, the *stop* and *squeeze* functions seem important for the study of GREEDY. If for every $x \in U^*$, $st(x) > sh(x)$, there is a simple algorithm that gives a $2\frac{1}{3}$-approximation bound (see [10]).

Of course the other important question in this area is whether OPT_S can be approximated within a factor of 2 by any algorithm. We conjecture that our algorithm can be modified slightly and the analysis improved to prove a $2\frac{1}{3}$ bound. Unfortunately the analysis is even more complicated and, perhaps worse, the algorithm becomes extremely complex.

Appendix. Proofs of Lemmas 4.13, 4.14, 5.1, 5.3, 5.4, and 6.4.

Proof of Lemma 4.13. 1. Suppose $g(x)$ is set by sq. Consider the case where c_i is colored by Rules 1, 2, or 3. By Lemma 4.12 c_i is colored in round $(1, i)$. In every rule $G(x) = SQ(x)$ is colored during the round if it is not already colored. Now suppose that c_i is colored by Rule 4. In this case $j = 2$. Since Rule 1 is not used in round $(1, i)$, $sq(x) > \frac{3}{2}$. Then since Rule 3 is not used in round $(1, i)$, $SQ(x) < x$. However, by Fact 4.11, $SQ(x)$ is colored when round $(2, i)$ begins.

2. The second point follows by the definition of force.

3. Now suppose that c_i is set by st. In any rule by which c_i can be colored, $ST(x)$ is also colored during the round if it isn't already colored when the round begins. Thus it suffices to show that $ST(x)$ is uncolored when the round begins. Suppose $ST(x)$ is colored in an earlier round. Referring to the definition of *set by*, since $ST(x)$ is colored the first rule does not apply. Thus the second must. By definition of stop $tr(ST(x), x) < 1$, and by Lemma 3.10, $tr(x, ST(x)) \geq 1$. Then since $\mathcal{V}_S(ST(x)) \geq 1$, $fo(x) \leq \max(\lfloor tr(x, ST(x)) \rfloor, 1) = st(x)$. However, this contradicts our priority rule. ☐

Proof of Lemma 4.14. 1. If $g(x)$ is set by sh, sq, or st, then by Lemma 4.12 the same conditions hold in round $(1, i)$, which is a contradiction. Thus $g(x)$ is set by fo in round $(2, i)$. Also by this lemma $fo(x)$ is integral, and since Rule 1 applies, $fo(x) < \frac{3}{2}$ so $fo(x) = 1$.

Assume that $c_{FO(x)} = c_k$. By Lemma 4.13, $FO(x)$ is colored when round $(2, i)$ begins. Suppose $FO(x)$ is *yellow*. Then $FO(x)$ is colored in round $(2, k)$ of Rule 4. Since $FO(x)$ is colored before round $(2, i)$ begins, $k < i$ and $FO(x) < x$. By Lemma 4.8 x conflicts with $FO(x)$ in round $(2, k)$. Since Rule 2 is not chosen, one of the following three conditions holds:

(a) $g(FO(x)) \leq \frac{3}{2}$. However, $FO(x)$ is colored *blue* in Rule 1, which is a contradiction.

(b) Stopping condition 2 fails to hold. Then $g(FO(x))$ is set by $st(FO(x)) = 2$ and $ST(FO(x)) < FO(x)$. This contradicts either Fact 4.11 or Lemma 4.13.

(c) $G(FO(x))$ is defined and $G(FO(x)) \in OP(c_i)$. Then by Lemma 4.13 c_i is colored no later than round $(2, k)$, which is a contradiction.

2. Suppose c_i is colored by Rule 2 in round $(2,i)$. Since $g(x) > \frac{3}{2}$ in round $(2,i)$ and since $g(x)$ in nonincreasing, $g(x) > \frac{3}{2}$ and $u > x$ is uncolored and conflicts with x in round $(1,i)$. Since x is not colored by Rule 2 in round $(1,i)$, one of the following conditions holds in round $(1,i)$:

(a) Stopping condition 2 fails to hold. Then $g(x)$ is set by $st(x) = 2$ and $ST(x) < x$. By Lemma 4.12 $g(x)$ is set by $st(x) = 2$ or by $fo(x) \leq 2$ in round $(2,i)$. In the first case we get a contradiction to either Fact 4.11 or Lemma 4.13. Thus $g(x)$ is set by $\frac{3}{2} < fo(x) \leq 2$; by Lemma 4.12 $fo(x) = 2$. Furthermore, since $st(x) = 2$ and $ST(x)$ is colored (by Fact 4.11), $FO(x) = ST(x)$.

(b) $G(x)$ is defined and $c_u = c_{G(x)}$. Since $G(x)$ is uncolored in round $(2,i)$, by Lemma 4.13 $g(x)$ is set by sq or st in round $(1,i)$. Since $G(x) \in c_u$ and $c_u > c_i$ but Rule 3 is not used in round $(1,i)$, $g(x)$ is set by $st(x) = 2$ in that round. By Lemma 4.12 $g(x) = 2$ in round $(2,i)$, and since $ST(x) > x$, this is the special case of *set by*. Thus $g(x)$ is set by st. However, stopping condition 2 fails to hold in round $(2,i)$, which is a contradiction.

3. Suppose c_i is colored by Rule 3 in round $(2,i)$. By Lemma 4.12 the same conditions hold in round $(1,i)$, which is a contradiction.

4. Suppose that for some $x \in OP(c_i)$ $g(x) \leq \frac{3}{2}$. Since Rule 1 is not used, $g(x)$ is set by $st(x) = 1$ and $ST(x) < x$. Then we get a contradiction to either Fact 4.11 or Lemma 4.13.

Now suppose that $g(x)$ is set by sq or fo in the round. By Lemma 4.12 and the fact that $g(x) > \frac{3}{2}$, $g(x) \geq 2$. Let $y = G(x)$ and assume $c_y = c_k$. By Lemma 3.10 $|x| \leq (tr(x,y)+1)|c_i| + |c_k|$. By Lemma 4.13 y is colored when the round begins. Suppose it is colored *yellow*. Then y is colored in round $(2,k)$ by Rule 4, and since it is colored when round $(2,i)$ begins, $k < i$, $|c_k| \leq |c_i|$. Hence $|x| \leq (tr(x,y)+2)|c_x|$. If $g(x)$ is set by sq, then by definition of squeeze $tr(x,y) \leq sq(x)$. If $g(x)$ is set by fo, then by the definition of force $tr(x,y) \leq fo(x)+1$. Thus $|x| \leq (g(x)+3)|c_x|$. Then since $g(x) \geq 2$, $|x| \leq \frac{5}{2}g(x)|c_x|$.

Now suppose that $ST(x)$ is defined and colored during the round. Since $st(x) \geq g(x) > \frac{3}{2}$, by Lemma 4.12, $st(x) \geq 2$. By Fact 4.11, $ST(x) > x$. If $st(x) \geq 3$, then c_i is colored no later than round $(1,i)$ of Rule 3, which is a contradiction, so $st(x) = 2$. If $g(x) = 2$, then this is the special case of *set by* and $g(x)$ is set by st. If $g(x) < 2$, then by Lemma 4.12 $g(x)$ is set by sh. $\square$

Proof of Lemma 5.1. The invariant holds when the construction begins. Suppose it holds at the beginning of round (j,i). If c_i is not colored in the round, the algorithm takes no action during the round and $\mathcal{C}_U$ is unchanged. So assume c_i is colored in round (j,i). Let w, u, z, and v be defined as above (conditioned on the various cases).

Invariant 1. To show that invariant 1 holds we need to show that the construction is correct in the sense that the newly created cycles and modified cycles exist in $\mathcal{G}_S$. Each new $m(c)$ construction in Table 2 exists by the string lemmas, Lemma 4.6, Corollary 3.8, and our naming convention. Now consider the case where an existing cycle of $\mathcal{C}_U$ is modified in the round. The only possibility is that $g(w)$ is set by sq, c_z is colored in an earlier round, and $e_{max}(w,z)$ is added to W. Since $g(w)$ is set by sq, $sq(w) < sh(w)$. Thus by Lemma 3.15, $ov(z,z)$ is periodic in $[c_w]$. Then if $ov_k(z,z)$ exists, so does $ov_{\max(k-|c_w|,0)}(z,z)$. The correctness of the modification to $\mathcal{C}_U$ then follows from Lemma 2.10.

Now suppose that c_w is matched. Each construction in Table 3 exists by Lemma 4.6, Corollary 3.8, and our naming convention, except the case when c_w is the winning cycle of the matched pair, c_w is *red*, and $\mathcal{V}_S(w) < 3$. We now prove the correctness of this last case. Assume (c_w, c_y) is an edge of M^*, where $y = rep(c_y)$. Consider the cycle $pair(w, y)$ that determines the weight of the edge (c_w, c_y). Let e be the version of e_w with pseudolength $\mathcal{V}_S(w)|c_w|$ and assume its overlap is k. Since c_w is the winning cycle of the pair, e is a winning edge for the cycle $pair(w, c)$. Furthermore, $\tilde{\mathcal{X}}(\{pair(w, y)\}, e)$ consists of the 1-expansion $c_w^{\mathcal{V}_S(w)}$ and the 1-expansion $\hat{c}_y$ of c_y with length $|pair(w, y)| - \mathcal{V}_S(w)|c_w|$. Since the self-loops of c_w and c_y are chosen by the matching algorithm over the edge (c_w, c_y) in the event of a tie in weight, it must be the case that $|\hat{c}_y| = |pair(w, y)| - \mathcal{V}_S(w)|c_w| < \mathcal{V}_S(y)|c_y|$. By Lemma 4.6 $\mathcal{V}_S(y) \leq sh(y)$. Thus the expansion of e_y in $\hat{c}_y$ has overlap more than $|y| - sh(y)|c_y|$. Then by Lemma 3.14 $ov_k(w, w)$ is periodic in $[\delta_y]$. Of course, it is also the case that $ov_k(w, w)$ is periodic in $[\delta_w]$. Then by Lemma 4.6 $c_w^{\mathcal{V}_S(w)}$ exists, so using Lemma 2.10 $c_w^{\mathcal{V}_S(w)+\min(|c_w|+|c_y|)/|c_w|}$ exists as well.

Invariant 2. Now consider invariant 2. Our modifications to existing cycles of $\mathcal{C}_U$ never eliminate a zero-overlap edge, so any weakly connected component of $\mathcal{C}_U \cup W$ that satisfies the invariant at the beginning of round (j, i) also satisfies the invariant at the end of the round. Thus we need only be concerned with the new cycles added to $\mathcal{C}_U$ during round (j, i). If $m(c)$ is created in round (j, i) for some $c \neq c_w$, then $e_{max}(w, y)$ is added to W for some $y \in OP(c)$. Thus it suffices to show that at the end of round (j, i) the weakly connected component of $\mathcal{C}_U \cup W$ containing $m(c_w)$ satisfies the invariant. If c_w is matched we are done, so assume c_w is unmatched. Referring to Table 2, the only cases in which one of the created cycles does not explicitly have a zero-overlap edge is when $g(w)$ is set by sq or fo, c_z is colored in a previous round, and $e_{max}(w, z)$ is added to W. However, in these cases, $m(c_w)$ is attached via $e_{max}(w, z)$ to the existing weakly connected component of $\mathcal{C}_U$ containing the expansion of $m(c_z)$; this component satisfies the invariant by our induction hypothesis.

Invariant 3. Now consider invariant 3. We point out that our naming convention does not affect our arguments in the proof of this invariant; i.e., in the case of a 1-expansion c_x^q where $q > |c_x|/|x|$, the edge from x_{last} to x_{first} has zero-overlap and any out-edge of x_{last} or in-edge of x_{first} has at least as much overlap. In proving that this invariant holds we will repeatedly use the fact that if c_x is non-*yellow* , then $m(c_x)$ is an expansion of $c_x^{\mathcal{V}_S(x)}$, which obviously holds if $\mathcal{V}_S(x) = 1$ and when $\mathcal{V}_S(x) > 1$ holds by construction since x must in fact be $rep(c_x)$. Also for the string w, $m(c_w)$ is an expansion of $c_w^{g(w)}$.

Suppose e is added to W during round (j, i). Then $e = e_{max}(w, u)$ or $e = e_{max}(w, z)$ or $e = e_{max}(w, v)$.

1. $e = e_{max}(w, u)$: In this case c_w is colored by Rule 2. As pointed out above, $m(c_w)$ is an expansion of $c_w^{g(w)}$ and, referring to Table 2, $m(c_u) = c_u^2$. By the definition of conflict, $tr(w, u) \leq g(w)$ and $tr(u, w) \leq 2$. Thus by Lemma 3.10, e is a winning edge for $\mathcal{C}_U$.

2. $e = e_{max}(w, z)$: Since z is defined, $g(w)$ is not set by sh. Suppose $g(w)$ is set by sq. Then $\mathcal{V}_S(w) = sq(w)$. Referring to Tables 2 and 3 for the three cases where $e_{max(w,z)}$ is added to W, $m(c_w)$ is an expansion of $c_w^{sq(w)}$ and the version of $m(c_z)$ in $\mathcal{C}_U$ is always an expansion of $c_z^{1+\min(|c_w|,|c_z|)/|c_z|}$. By Lemma 3.15 e is a winning edge for $\mathcal{C}_U$.

Suppose $g(w)$ is set by fo. In the only case where $e_{max}(w, z)$ is added to W, $m(c_w) = c_w^{fo(w)+1}$. By definition $fo(w) + 1 > tr(w, z)$. Since z is colored in an earlier round, $\mathcal{C}_U$ includes an expansion of c_z and we claim that it is in fact an expansion of c_z^q where $q \geq tr(z, w)$, so e is a winning edge by Lemma 3.10. If z is non-*yellow*, then let $q = \mathcal{V}_S(z)$ and the claim holds; note that earlier we argued that $m(c_z)$ is an expansion of $c_z^{\mathcal{V}_S(z)}$ if z is non-*yellow*. Now suppose that z is *yellow*. If c_w is colored by Rules 1–3, then by Lemma 4.14 c_w is colored by Rule 2, $z = ST(w)$, and so $q = 1$ satisfies the claim. If c_w is colored in Rule 4, then since c_z is *yellow* the algorithm does not add $e_{max}(w, z)$ to W.

Finally, suppose that $g(w)$ is set by st. When c_w is unmatched $m(c_w) = w$. Now suppose c_w is matched. If c_w is *yellow*, then $st(w) \geq 2$ and if c_w is *red*, then c_w is in $\mathcal{R}_2$ and $st(w) \geq 3$. In each case $m(c_w)$ is an expansion of $c_w^{st(w)+1}$. Also, $m(c_z) = c_z$. By the definition of stop, $1 > tr(z, w)$ and $st(w) + 1 > tr(w, z)$. By Lemma 3.10 e is a winning edge for $\mathcal{C}_U$.

3. $e = e_{max}(w, v)$: Now c_w is *yellow*, the round is $(2, i)$, $w = rep(c_w)$, $\frac{3}{2} < \mathcal{V}_S(w) \leq 2$, and $v = ST(x)$ for some $x \in OP(c_w)$ such that $st(x) = 2$. By Lemma 2.10 $\delta_w \in [\delta_x]$. Since c_v is uncolored when round $(2, i)$ begins, by Lemma 4.13 $|c_v| \geq |c_w|$. Thus $tr(v, w) < tr(v, x) + 1$. By definition of stop $tr(v, x) < 1$ so $tr(v, w) < 2$. In every case $m(c_v) = c_v^2$. Similarly, $tr(w, v) \leq tr(x, v) + |c_w|/|c_v| \leq tr(x, v) + 1$. By definition of stop $tr(x, v) < st(x) + 1$. Thus $tr(w, v) < st(x) + 2 = 4$. Referring to Table 2 for the cases where c_w is unmatched, $m(c) = w$. If c_w is matched, then $m(c_w)$ is either w or $c_w^{\mathcal{V}_S(w)+2}$, where $\mathcal{V}_S(w) \geq 2$. By Lemma 3.10 e is a winning edge for $\mathcal{C}_U$.

Invariant 4. Finally, consider Invariant 4. We analyze the lengths and allocations depending on the rule by which c_w is colored. The analysis is summarized in Table 4.

1. Suppose that c_w is colored by Rule 1. Then c_w is *blue* and, by Fact 4.17, it is unmatched. Also, $1 \leq g(w) < \frac{3}{2}$ and if $g(w)$ is not set by sh, then by Lemma 4.12 $g(w) = 1$. Furthermore, if z is defined then $e_{max}(w, z)$ is added to W in the round. If in addition c_z is colored during the round, then c_z is *blue*. The possible cases are shown in Table 4 for Rule 1. This table also gives the construction of $m(c_w)$ and, when applicable, the column *length bound* gives an upper bound on the increase in $|\mathcal{C}_U|$ during the round. The length increase is due to the newly created cycles and any expansions to existing cycles. The bound in each case is obvious from the construction and the length bounds given by the string lemmas. The column *allocation* gives the total allocation to c_w and, if c_z is colored in the round, to c_z. In each case the total allocation is at least as large as the total length increase. In the first case, by Lemma 3.7, $|w| \leq (sh(w)+1)|c_w|$. Since c_w is *blue*, its allocation is $alloc(c_w) = \frac{5}{2}|c_w|$. Since $sh(w) \leq \frac{3}{2}$, $|w| \leq alloc(c)$. In the second case $|c_w^{sq(w)}| = sq(w)|c_w|$ (by definition) and c_z is expanded by up to $|c_w|$. Thus the total length increase is no more than $(sq(w)+1)|c_w|$. This is no more than $alloc(c_w) = \frac{5}{2}|c_w|$ since $sq(w) = 1$. In the third case $|c_w^{sq(w)}| = sq(w)|c_w|$, and by Lemma 3.15 $|z| \leq 2|c_z| + |c_w|$ for a total length increase of no more than $(sq(w) + 1)|c_w| + 2|c_z|$, which is no more than the total allocation. The fourth case follows in a similar way using Lemma 3.10. The last case also uses Lemma 3.10 for its length bound, which is no more than the total allocation since $|c_z| \geq |c_w|$; this last fact holds because stopping condition 1 holds.

2. Suppose c_w is colored by Rule 2. Then c_w is *red*, $\mathcal{V}_S(w) = g(w)$ as computed at the beginning of the round, and $g(w) > \frac{3}{2}$. If c_w is matched, then by Fact 4.17 $g(w) > 2$. As before, by Lemma 4.12 $g(x)$ is integral if set by something other than sh. The *no*/*no*/*no* cases of Table 2 are not applicable under this rule since $e_{max}(z, w)$ is added to W provided z is defined and c_i is unmatched. The table gives the construction for $m(c_w)$ and, if z is defined, for $m(c_z)$. For this rule u is defined, colored *blue* in the round, and $m(c_u) = c_u^2$. If z is defined and colored in the round it is also *blue*.

TABLE 4
Cost accounting.

Case from Table 2		Construction		Analysis	
		$m(c_w)$	$m(c_z)$	Length bound	Allocation
Rule 1 - All cycles *blue*.					
$1 \leq sh(w) \leq \frac{5}{2}$	no/NA/NA	w	NA	$(sh(w)+1)\|c_w\|$	$\frac{5}{2}\|c_w\|$
$sq(w) = 1$	no/no/yes	$c_w^{sq(w)}$	expand	$(sq(w)+1)\|c_w\|$	$\frac{5}{2}\|c_w\|$
	no/yes/yes	$c_w^{sq(w)}$	z	$(sq(w)+1)\|c_w\|+2\|c_z\|$	$\frac{5}{2}(\|c_w\|+\|c_z\|)$
$fo(w) = 1$	no/no/yes	$c_w^{fo(w)+1}$	NC	$(fo(w)+1)\|c_w\|$	$\frac{5}{2}\|c_w\|$
$st(w) = 1$	no/yes/yes	w	c_z	$(st(w)+2)\|c_w\|+2\|c_z\|$	$\frac{5}{2}(\|c_w\|+c_z\|)$
Rule 2 - c_w is *red* and all other cycles are *blue*, $m(c_u) = c_u^2$.					
$sh(w) > \frac{3}{2}$	no/NA/NA	w	NA	$(sh(w)+1)\|c_w\|+2\|c_u\|$	$\frac{5}{2}(\max(sh(w),2)-1)\|c_w\|$ $+\frac{5}{2}\|c_u\|)$
$sh(w) > 2$	yes/NA/NA	$m(c_w)$	NA	$\|m(c_w)\|+2\|c_u\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}\|c_u\|$
$sq(w) \geq 2$	no/no/yes	$c_w^{sq(w)}$	expand	$(sq(w)+1)\|c_w\|+2\|c_u\|$	$\frac{5}{2}((sq(w)-1)\|c_w\|+\|c_u\|)$
	no/yes/yes	$c_w^{sq(w)}$	z	$(sq(w)+1)\|c_w\|$ $+2\|c_z\|+2\|c_u\|$	$\frac{5}{2}((sq(w)-1)\|c_w\|+\|c_z\|+\|c_u\|)$
$sq(w) \geq 3$	yes/no/no	$m(c_w)$	NC	$\|m(c_w)\|+2\|c_u\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}\|c_u\|$
	yes/yes/yes	$m(c_w)$	c_z^2	$\|m(c_w)\|+2\|c_z\|+2\|c_u\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}(\|c_z\|+\|c_u\|)$
$fo(w) \geq 2$	no/no/yes	$c_w^{fo(w)+1}$	NC	$(fo(w)+1)\|c_w\|+2\|c_u\|$	$\frac{5}{2}((fo(w)-1)\|c_w\|+\|c_u\|)$
$fo(w) \geq 3$	yes/no/no	$m(c_w)$	NC	$\|m(c_w)\|+2\|c_u\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}\|c_u\|$
$st(w) \geq 2$	no/yes/yes	w	c_z	$\|w\|+\|c_z\|+2\|c_u\|$	$\frac{5}{2}((st(w)-1)\|c_w\|+\|c_z\|+\|c_u\|)$
$st(w) \geq 3$	yes/yes/yes	$m(c_w)$	c_z	$\|m(c_w)\|+\|c_z\|+2\|c_u\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}(\|c_z\|+\|c_u\|)$
Rule 3 - c_w is *red* and c_z is *blue*.					
$sq(w) \geq 2$	no/yes/yes	$c_w^{sq(w)}$	z	$(sq(w)+1)\|c_w\|+2\|c_z\|$	$\frac{5}{2}((sq(w)-1)\|c_w\|+\|c_z\|)$
$sq(w) \geq 3$	yes/yes/yes	$m(c_w)$	c_z^2	$\|m(c_w)\|+2\|c_z\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}(\|c_z\|)$
$st(w) \geq 3$	no/yes/yes	w	c_z	$(st(w)+2)\|c_w\|+2\|c_z\|$	$\frac{5}{2}((st(w)-1)\|c_w\|+\|c_z\|)$
$st(w) \geq 3$	yes/yes/yes	$m(c_w)$	c_z	$\|m(c_w)\|+\|c_z\|$	$\|m(c_w)\|-\frac{1}{2}\|c_w\|+\frac{5}{2}\|c_z\|$
Rule 4 - c_w is *yellow* and all other cycles are *blue*. For each v colored in the round $\|m(c_v)\| = 2\|c_v\| < \frac{5}{2}\|c_v\| = alloc(c_v)$.					
$sh(w) > \frac{3}{2}$	no/NA/NA	w	NA	$(sh(w)+1)\|c_w\|$	$\frac{5}{2}sh(w)\|c_w\|$
	yes/NA/NA	$m(c_w)$	NA	$\|m(c_w)\|$	$\|m(c_w)\|$
$sq(w) \geq 2$	no/no/no	w	NC	$\|w\|$	$\frac{5}{2}sq(w)\|c_w\|$
	no/no/yes	$c_w^{sq(w)}$	expand	$(sq(w)+1)\|c_w\|$	$\frac{5}{2}sq(w)\|c_w\|$
	yes/no/no	$m(c_w)$	NC	$\|m(c_w)\|$	$\|m(c_w)\|$
$fo(w) \geq 2$	no/no/no	w	NC	$\|w\|$	$\frac{5}{2}fo(w)\|c_w\|$
	no/no/yes	$c_w^{fo(w)+1}$	NC	$(fo(w)+1)\|c_w\|$	$\frac{5}{2}fo(w)\|c_w\|$
	yes/no/no	$m(c_w)$	NC	$\|m(c_w)\|$	$\|m(c_w)\|$
$st(w) \geq 2$	no/yes/yes	w	c_z	$(st(w)+2)\|c_w\|+2\|c_z\|$	$\frac{5}{2}(st(w)\|c_w\|+\|c_z\|)$
	yes/yes/yes	$m(c_w)$	c_z	$\|m(c_w)\|+\|c_z\|$	$\|m(c_w)\|+\frac{5}{2}\|c_z\|$

In every case the length bound is obvious using the case and the string lemmas. In every case the allocation is clearly as large as the length bound, using the fact that $|c_u| \geq |c_w|$ and the range for $g(w)$, except the case $st(w) \geq 2$ *no/yes/yes*. We prove this case next. Using the definition of stop we get the bound $|w| \leq (st(w)+2)|c_w| + |c_z|$ from Lemma 3.10. This gives a total length increase of no more than $(st(w)+2)|c_w| + 2|c_z| + 2|c_u|$, which is no more than the allocation when $st(w) \geq 3$. Now suppose $st(w) = 2$. Since stopping condition 2 holds, $c_z > c_w$. Using the length bound of Lemma 3.10 with the definitions of stop and conflict we get

$$\begin{aligned} |w| &\leq \frac{1}{2}((st(w)+2)|c_w| + |c_z| + (st(w)+1)|c_w| + |c_u|) \\ &\leq \left(st(w) + \frac{3}{2}\right)|c_w| + \frac{1}{2}|c_z| + \frac{1}{2}|c_u|. \end{aligned}$$

Then the total length increase is no more than

$$\left(st(w) + \frac{3}{2}\right)|c_w| + \frac{3}{2}|c_z| + \frac{5}{2}|c_u|,$$

which is no more than the allocation since $|c_z| \geq |c_w|$, $|c_u| \geq |c_w|$, and $st(w) = 2$.

3. Suppose c_w is colored by Rule 3. In this case $g(w)$ is set by sq or st, z is defined, $c_z > c_w$, c_z is colored in the round, and $e_{max}(w,z)$ is added to W. If $g(w)$ is set by st, then $st(w) \geq 3$. If $g(w)$ is set by sq, then $sq(w) \geq 2$ with equality only if $c_z \in \mathcal{R}_1$, in which case c_w is unmatched. The table handles all the possibilities. The bound and allocation arguments follow as before.

4. Suppose c_w is colored by Rule 4. In this case c_w is *yellow*, $w = rep(c_w)$, and $g(w) > \frac{3}{2}$ subject to the integrality constraints of Lemma 4.12. By Lemma 4.13, if $g(w)$ is set by sq, then $SQ(w)$ is colored in an earlier round so the $*/yes/*$ cases do not apply. Furthermore, by our choice of representative, if $g(w)$ is set by $st(w)$, then $ST(w) > w$ and is colored during the round. The analysis follows as described for the previous rules except for the *no/no/no* cases; these follow from Lemma 4.14. Finally, for every v that is defined, $|m(c_v)| = 2|c_v| \leq \frac{5}{2}|c_v|$. ◻

Proof of Lemma 5.4. Let c be a cycle of $\mathcal{C}_S{}^*$ and let $\hat{c}$ be the 1-expansion of c in $\mathcal{C}_L$. In each of the first three cases of Definition 4.19, the allocation in Table 1 satisfies the following:

$$alloc(c) = \frac{5}{2}(|\hat{c}| - 1_{c \in \mathcal{R}} \cdot |c|),$$

where $1_{c\in\mathcal{R}}$ is 1 if $c \in \mathcal{R}$ and 0 otherwise. Thus it suffices to show that for matched pairs $(c_1, c_2) \in M^*$,

$$\sum_{i=1,2} |m(c_i)| - \frac{1}{2} \cdot 1_{c\in\mathcal{R}_2} \cdot |c| \leq \frac{5}{2} \sum_{i=1,2} (|\hat{c}_i| - 1_{c_i\in\mathcal{R}_2} \cdot |c_i|).$$

Note that by construction of $\mathcal{G}(\mathcal{V}_S)$ if $c \in \mathcal{R}_1$, then c is unmatched.

Let c_1 and c_2 be a matched pair in M^*. Assume for $i = 1, 2$ that $x_i = rep(c_i)$, $\hat{c}_i$ is the 1-expansion of c_i in $\mathcal{C}_L$, $\hat{e}_i$ is the expansion of $e_{rep(c_i)}$ in $\hat{c}_i$, and $k_i = ov(\hat{e}_i)$. Also assume that c_1 is the max-overlap cycle of the pair. Next we assemble the following facts about these cycles:

1. $k_2 > |c_2|$: This was argued in the proof of invariant 1 for Lemma 5.1 when c_w is *red*, but the proof doesn't rely on the coloring.

2. $ov_{k_1}(x_1, x_1)$ is periodic in $[\delta_{x_1}]$: This also was in the proof cited above.
3. $k_1 \leq |c_1| + |c_2|$: This follows from the previous fact by Lemma 2.10.
4. If $\mathcal{V}_S(x_1)$ is nonintegral, then $k_1 < |c_1|$. This holds by Corollary 3.8 and Lemma 4.6.
5. $k_2 \leq k_1$: This holds by Lemma 3.3.
6. If c_2 is *red*, then $|\hat{c}_2| = \ell|c_2|$ for some $\ell \geq 2$: This holds by Lemma 4.18.
7. $\hat{c}_1 = c_{x_1}^{\mathcal{V}_S(x_1)}$: This holds by Definition 4.19.

We now consider the four possible constructions in Table 3.

1. c_1 is *yellow* and $\frac{3}{2} < \mathcal{V}_S(x_1) < 2$: In this case $1_{c_1 \in \mathcal{R}_2} = 0$, $m(c_i) = x_i$, and by points 4 and 5 above $|x_i| \leq |\hat{c}_i| + |c_1|$. These facts give the first inequality below. By this case and point 7 above, $\frac{3}{2}|c_1| < |\hat{c}_1| < 2|c_1|$, and by point 6 $|\hat{c}_2| - 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| = k|c_2|$ for some $k \geq 1$. These facts yield the second inequality, where we again use $1_{c_1 \in \mathcal{R}_2} = 0$:

$$\begin{aligned}|m(c_1)| + |m(c_2)| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| &= |x_1| + |x_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq |\hat{c}_1| + |\hat{c}_2| + 2|c_1| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq \tfrac{5}{2} \textstyle\sum_{i=1,2}(|\hat{c}_i| - 1_{c_i \in \mathcal{R}_i} \cdot |c_i|).\end{aligned}$$

2. c_1 is *yellow* and $\mathcal{V}_S(x_1) \geq 2$: In this case $1_{c_1 \in \mathcal{R}_2} = 0$, $m(c_1) = c_{x_1}^{\mathcal{V}_S(x_1)+2}$, and $m(c_2) = x_2$, so by point 7 we get the equality below. By points 3 and 5 above, $|x_2| \leq |\hat{c}_2| + |c_1| + |c_2|$ so we get the following first inequality. By this case and point 7 above $|\hat{c}_1| \geq 2|c_1|$, and by point 6 $|\hat{c}_2| - 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| = k|c_2|$ for some $k \geq 1$. These facts yield the second inequality, where we again use $1_{c_1 \in \mathcal{R}_2} = 0$:

$$\begin{aligned}|m(c_1)| + |m(c_2)| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| &= |\hat{c}_1| + 2|c_1| + |x_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq |\hat{c}_1| + 3|c_2| + |\hat{c}_2| + |c_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq \tfrac{5}{2} \textstyle\sum_{i=1,2}(|\hat{c}_i| - 1_{c_i \in \mathcal{R}_i} \cdot |c_i|).\end{aligned}$$

3. c_1 is *red* and $2 < \mathcal{V}_S(x_1) < 3$: In this case $1_{c_1 \in \mathcal{R}_2} = 1$, $m(c_1) = c_{x_1}^{\mathcal{V}_S(x_1)+\frac{|c_2|}{|c_1|}}$, and $m(c_2) = x_2$. These facts plus point 7 yield the equality below. By points 4 and 5 $|x_2| \leq |\hat{c}_2| + |c_1|$, giving us the first inequality below. By point 7, $2|c_1| \leq |\hat{c}_1| \leq 3|c_1|$, and by point 6, $|\hat{c}_2| - 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| = k|c_2|$ for some $k \geq 1$. These facts yield the second inequality:

$$\begin{aligned}|m(c_1)| - \tfrac{1}{2}|c_1| + |m(c_2)| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| &= |\hat{c}_1| + |c_2| - \tfrac{1}{2}|c_1| + |x_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq |\hat{c}_1| + \tfrac{1}{2}|c_1| + |\hat{c}_2| + |c_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq \tfrac{5}{2} \textstyle\sum_{i=1,2}(|\hat{c}_i| - 1_{c_i \in \mathcal{R}_i} \cdot |c_i|).\end{aligned}$$

4. c_1 is *red* and $\mathcal{V}_S(x_1) \geq 3$: In this case $1_{c_1 \in \mathcal{R}_2} = 1$, $m(c_1) = c_{x_1}^{\mathcal{V}_S(x_1)+1}$, and $m(c_2) = x_2$. These facts plus point 7 yield the equality below. By points 3 and 5 $|x_2| \leq |\hat{c}_2| + |c_1| + |c_2|$, giving us the first inequality below. By point 7, $|\hat{c}_1| \geq 3|c_1|$, and by point 6, $|\hat{c}_2| - 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| = k|c_2|$ for some $k \geq 1$. These facts yield the second inequality:

$$\begin{aligned}|m(c_1)| - \tfrac{1}{2}|c_1| + |m(c_2)| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| &= |\hat{c}_1| + \tfrac{1}{2}|c_1| + |x_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq |\hat{c}_1| + \tfrac{3}{2}|c_1| + |\hat{c}_2| + |c_2| - \tfrac{1}{2} \cdot 1_{c_2 \in \mathcal{R}_2} \cdot |c_2| \\ &\leq \tfrac{5}{2} \textstyle\sum_{i=1,2}(|\hat{c}_i| - 1_{c_i \in \mathcal{R}_i} \cdot |c_i|). \qquad \square\end{aligned}$$

To prove Lemma 6.4 we need the following lemma and corollaries.

LEMMA A.1. *If c is a cycle of $\mathcal{C}_S{}^*$ such that $d_c \geq 3$, then for any vertex s of c, $[\delta_s] = [c]$.*

Proof. Suppose $[\delta_s] \neq [c]$. By definition, δ_s is irreducible. By Lemma 2.10, every $\alpha \in [c]$ is irreducible. Furthermore, s is periodic in $[\delta_s]$ and $[c]$ so, using Lemma 2.5, $|s| < |\delta_s| + |c|$. Let x be the open of c such that $x_{first} = s$. Then $|x| < |s| + |c| < |\delta_s| + 2|c|$. Since s is periodic in $[c]$, $|\delta_s| < |c|$. Thus $|x| < 3|c|$ and $d_c < 3$. □

COROLLARY A.2. *Let c be a cycle in $\mathcal{C}_S{}^*$ and let $\hat{c}$ be a cycle of $\mathcal{G}_S$ that is uniform for c. If $d_c \geq 3$, then $|\hat{c}| \geq |c|$.*

Proof. This follows from the fact that any cycle including a vertex s must have length at least $|\delta_s|$. □

The length of a path in $\mathcal{G}_S$ is the sum of the edge lengths of the path.

COROLLARY A.3. *Let c be a cycle in $\mathcal{C}^*$ containing an edge $\langle s, t\rangle$, $s \neq t$. Let p be any simple path in $\mathcal{G}_S$ from t to s. If $d_c \geq 3$, then $|p| \geq |c| - |pref(s,t)|$.*

Proof. Consider the tight cycle on the vertices s and t; this cycle has length $|pref(s,t)| + |pref(t,s)|$. By the previous corollary the length of this cycle is at least $|c|$. Thus $|pref(t,s)| \geq |c| - |pref(s,t)|$. The tight edges of G_S satisfy the triangle inequality so any path from t to s has length at least $|c| - |pref(s,t)|$. □

Proof of Lemma 6.4. Let c be a cycle of $\mathcal{C}_S{}^*$ such that $d_c \geq 3$. Since $\mathcal{C}_{final}$ is uniform for c and $\mathcal{C}_0$ is not, there is some smallest $i > 0$ such that $\mathcal{C}_i$ is uniform for c. Steps (i), (iii), and (iv) cannot create a uniform cycle, so it must be the case that $\mathcal{C}_i = \tilde{\mathcal{X}}(\mathcal{C}_{i-1}, e)$ for some edge e in $\mathcal{G}_S$. Since $\mathcal{C}_{i-1}$ is not uniform for c, some vertex s of c is in a mixed cycle in $\mathcal{C}_{i-1}$. Furthermore, an edge exchange can increase the number of cycles uniform for some c by at most 1. Thus if $i = 1$, by our assumptions on $\mathcal{C}_0$, c is represented in $\mathcal{C}_0$ and marked in the first step. Suppose $i > 1$. Then $\mathcal{C}_{i-1}$ is produced during the previous round. By step (iii) we can assume that $\mathcal{C}_{i-1}$ does not contain any cycles uniform for c. Thus $\mathcal{C}_i$ contains exactly one cycle uniform for c. It must be the case that the vertices of c comprise a path in a mixed cycle of $\mathcal{C}_{i-1}$. By step (iv) c is represented in the mixed cycle. Thus c is marked in step (i) of the current round.

Let c be a cycle of $\mathcal{C}_S{}^*$ such that $d_c < 3$. Suppose that $c \in \mathcal{C}_{final}$. As argued above there are covers $\mathcal{C}_{i-1}$ and $\mathcal{C}_i$, where $\mathcal{C}_i$ is uniform for c, $\mathcal{C}_{i-1}$ is not, and $\mathcal{C}_i = \tilde{\mathcal{X}}(\mathcal{C}_{i-1}, e)$ for some e of $\mathcal{G}_S$. By construction, no subsequent steps modify the cycles uniform for c, so since $c \in \mathcal{C}_{final}$, $c \in \mathcal{C}_i$. Then it is easy to see that the edge exchange must replace a mixed cycle in $\mathcal{C}_{i-1}$ by two cycles, one of which is c. However, it must be the case that c is represented in the mixed cycle in $\mathcal{C}_{i-1}$ and thus is marked in step (i).

Now suppose that $c \notin \mathcal{C}_{final}$. Then $\mathcal{C}_{final}$ includes cycles $\hat{c}_1, \ldots, \hat{c}_k$ that are uniform for c and partition its vertices. Suppose ($k = 1$ and) $\hat{c}_1$ is a 1-expansion c_x^q of c. If q is nonintegral, then by Corollary 3.8 $q \geq sh(x)$. Now suppose q is integral. Since $c_x^q \neq c$, by Corollary 3.8, $q \geq 2$. Since $d_c < 3$, there is some open y of c such that $|y| < 3|c|$ and so $sh(y) \leq 2$.

If the above case does not hold, we can perform edge exchanges based on the edges of c (in order by increasing overlap) until we reach some final exchange producing a 1-expansion of c. If this 1-expansion is different from c, the proof follows as above. Otherwise c is produced by an edge exchange $\tilde{\mathcal{X}}(\{\tilde{c}_1, \tilde{c}_2\}, e)$, where $\tilde{c}_1$ and $\tilde{c}_2$ partition the vertices of c, $|\tilde{c}_1| + |\tilde{c}_2| = |c|$, and $e = \langle s, t\rangle$ for some s in $\tilde{c}_1$ and t in $\tilde{c}_2$. If $[\tilde{c}_1] = [\tilde{c}_2]$, then every vertex in c is periodic in $[\tilde{c}_1]$ and there is a cycle on these vertices with length $|\tilde{c}_1|$; this contradicts the optimality of $\mathcal{C}_S{}^*$. Thus $[\tilde{c}_1] \neq [\tilde{c}_2]$. Then by Corollary 2.6 $ov(e) < |\tilde{c}_1| + |\tilde{c}_2| = |c|$. The edge e is e_x for some $x \in OP(c)$ and $|x| = |c| + ov(e) < 2|c|$ so $sh(x) = 1$. □

Acknowledgment. We'd like to thank Sampath Kannan for his careful checking of our proof and suggestions on improving the exposition.

REFERENCES

[1] C. Armen and C. Stein, *Improved length bounds for the shortest superstring problem*, in Proceedings 5th International Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 494–505.

[2] C. Armen and C. Stein, *A $2\frac{2}{3}$ approximation algorithm for the shortest superstring problem*, in Proceedings Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 1075, Springer-Verlag, Berlin, 1996, pp. 87–101.

[3] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, *Linear approximation of shortest superstrings*, in Proceedings 23rd Annual ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 328–336.

[4] D. Breslauer, T. Jiang, and Z. Jiang, *Rotations of periodic strings and short superstrings*, J. Algorithms, 24 (1997), pp. 340–353.

[5] A. Czumaj, L. Gasieniec, M. Piotrow, and W. Rytter, *Parallel and sequential approximations of shortest superstrings*, in Proceedings First Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 824, Springer-Verlag, Berlin, 1994, pp. 95–106.

[6] T. R. Gingeras, J. P. Milao, P. Sciaky, and R. J. Roberts, *Computer programs for the assembly of dna sequences*, Nucleic Acid Res., 7 (1979), pp. 529–545.

[7] R. Kosaraju, J. Park, and C. Stein, *Long tours and short superstrings*, in Proc. 35th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1994, pp. 166–177.

[8] A. Lesk, ed., *Computational Molecular Biology, Sources and Methods for Sequence Analysis*, Oxford University Press, Oxford, 1988.

[9] J. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

[10] E. Sweedyk, *A $2\frac{1}{2}$ Approximation Algorithm for Shortest Common Superstring*, Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1995.

[11] S. Teng and F. Yao, *Approximating shortest superstrings*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 158–165.

[12] J. Turner, *Approximation algorithms for the shortest common superstring problem*, Inform. and Comput., 83 (1989), pp. 1–20.

[13] E. Ukkonen, *On-line Construction of Suffix-Trees*, Technical Report A-1993, Department of Computer Science, University of Helsinki, Finland, 1993.

SIAM J. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 29, No. 3, pp. 987–1007

ON THE STRUCTURE OF LOGSPACE PROBABILISTIC COMPLEXITY CLASSES[*]

IOAN I. MACARIE[†]

Abstract. We investigate hierarchical properties and logspace reductions of languages recognized by logspace probabilistic Turing machines, Arthur–Merlin games, and games against nature with logspace probabilistic verifiers. Each logspace complexity class is decomposed into a hierarchy based on corresponding two-way multihead finite-state automata and we (eventually) prove the separation of the hierarchy levels (even for languages over a single-letter alphabet); furthermore, we show efficient reductions of each logspace complexity class to, or between, low levels of its corresponding hierarchy.

We find probabilistic and probabilistic-plus-nondeterministic variants of Savitch's maze threading problem which are logspace complete for **PL** (the class of languages recognized by logspace probabilistic Turing machines) and, respectively, **P** (the class of languages recognized by polynomial-time deterministic Turing machines), and which can be recognized by one-way non-sensing two-head (or one-way one-head one-counter) finite-state automata with probabilistic and both probabilistic and nondeterministic states, respectively.

Key words. multihead finite automata, heads hierarchy, probabilistic computation, probabilistic Turing machines, Arthur–Merlin games, games against nature, logspace reductions

AMS subject classifications. 68Q15, 68Q75, 68Q05, 68Q10, 68Q22, 68R99

PII. S0097539796298339

1. Introduction.

Deterministic and nondeterministic multihead finite(-state) automata were studied primarily in the 1970s. Several results were published, including characterizations of hierarchies of multihead finite automata, their connections with logspace Turing machines, and transformations of languages recognized by one type of devices to languages recognized by the same or different types of devices [16, 18, 44, 30, 39, 40, 46, 31]. More specifically, some of these results show that the classes recognized by deterministic and nondeterministic logspace Turing machines (i.e., **L** and **NL**, respectively) can be represented as proper hierarchies defined by deterministic and, respectively, nondeterministic two-way multihead finite automata [30, 31]. Other results provide variants of Savitch's maze threading problem that are logspace complete for **NL** and can be solved by simple nondeterministic automata (more exactly, by one-way non-sensing two-head or one-way one-head one-counter nondeterministic finite automata [44]). The corresponding relations hold for alternating computation as well [24]. In the probabilistic settings only some of the analogous results have been reported so far. All that is currently known are characterizations of logspace probabilistic complexity classes in terms of hierarchies defined by the corresponding two-way multihead finite automata and separations of head hierarchies for two-way multihead unbounded-error and one-sided-error probabilistic finite automata [29].

[*]Received by the editors February 2, 1996; accepted for publication (in revised form) June 19, 1997; published electronically December 15, 1999. This research was supported in part by National Science Foundation grant CDA 8822724 while the author was at the Department of Computer Science, University of Rochester. An extended abstract appears in Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (STACS'95).

http://www.siam.org/journals/sicomp/29-3/29833.html

[†]Research and Advanced Development Group, Metamor Software Solutions, 2851 Clover Street, Pittsford, NY 14534 (macarie@metamorsw.com).

One goal of this paper is to further investigate properties of logspace probabilistic Turing machines, focusing on their connection with multihead *one*-way probabilistic finite automata. We show that the languages recognized by polynomial-time logarithmic-space (bounded-error and one-sided-error) probabilistic Turing machines are logspace reducible to languages recognized by the corresponding one-way non-sensing four-head probabilistic finite automata that move at least one head at each computation step. In the setting of *un*bounded-error probabilistic computation we obtain the even stronger reductions to languages recognized by one-way non-sensing two-head probabilistic finite automata (or one-way one-head one-counter probabilistic finite automata) that move at least one head at each computation step. (See section 4.) These last reductions parallel the reductions of Sudborough for nondeterministic computation [44]. They imply immediately Jung's well-known result which states the equivalence, with respect to the language recognition, of logspace probabilistic Turing machines and polynomial-time logspace probabilistic Turing machines [20]. As a consequence of these reductions it follows that **PL** (the class of languages recognized by logspace probabilistic Turing machines) can be characterized as the class of languages logspace reducible to languages recognized by these simple one-way probabilistic finite-state automata. These reductions are quite surprising to hold in the setting of probabilistic computation. For instance, it is known that several versions of matrix inversion are logspace complete for **PL** [2, 20, 28]. The above-mentioned reductions show that **PL** is efficiently reducible to classes of languages recognized by one-way one-head one-counter or one-way non-sensing two-head probabilistic finite automata although these devices do not seem to have enough computational power to invert matrices.

Another goal of this paper is to generalize these separations and reductions obtained for traditional probabilistic devices to the settings of Arthur–Merlin games (AM-games) and games against nature having as verifiers logspace probabilistic Turing machines or two-way and one-way multihead probabilistic finite automata (see sections 3 and 4). We obtain that **P** (the class of languages recognized by polynomial-time deterministic Turing machines) is logspace reducible to the class of languages recognized by one-way one-head one-counter or one-way non-sensing two-head finite automata with both probabilistic and nondeterministic states.

The structure of the paper is as follows.

In section 2 we give some definitions and notations, and we state the new results in terms of these notations. Table 2.1 contains a summary of these results and shows some of the remaining open problems.

In section 3 we notice the equalities between the classes of languages recognized by AM-games (and games against nature) with logspace probabilistic Turing machines as verifiers and the classes of languages recognized by the same games with two-way multihead probabilistic finite-state verifiers. Next we show that for several AM-games (and games against nature) with two-way non-sensing multihead probabilistic finite-state verifiers, some of the corresponding head hierarchies are proper.

In section 4 we present efficient reductions of logspace probabilistic complexity classes to classes of languages recognized by simple probabilistic finite automata. Also, we generalize these results and show similar reductions in the settings of AM-games and games against nature. To prove some of these results, we design probabilistic and, respectively, probabilistic-plus-nondeterministic variants of Savitch's maze threading problem that are logspace complete for the corresponding logspace complexity classes. These problems can be solved by one-way non-sensing two-head or one-way one-

TABLE 2.1
Hierarchy properties of logspace complexity classes.

	Head-hierarchy characterization	Properness of the hierarchy	No. of heads of 1-way automata recognizing logspace complete problems
L	[16]	[30, 31]	N/A
NL	[16]	[30, 31]	2 [44]
AL	[24]	[24]	2 [24]
PL	[29]	[29]	2 [Theorem 4.4]
$\mathbf{PL}_{poly}$	[29]	[29]	2 [Theorem 4.4]
BPL	[29]	not known	not known
$\mathbf{BPL}_{poly}$	[29]	not known	4 [Theorem 4.3]
RL	[29]	[29]	not known
$\mathbf{RL}_{poly}$	[29]	not known	4 [Theorem 4.3]
AML	[Theorem 1]	[Theorems 3.2, 3.7]	not known
$\mathbf{AML}_{poly}$	[Theorem 1]	not known	4 [Theorem 4.9]
UAML	[Theorem 1]	[Theorems 3.2, 3.7]	2 [Theorem 4.9]
$\mathbf{UAML}_{poly}$	[Theorem 1]	[Theorems 3.2, 3.7]	2 [Theorem 4.9]

head one-counter finite-state automata with probabilistic and both probabilistic and nondeterministic states, respectively.

2. Background. This section presents the definitions and notations used in this article. In its final part we summarize our results in terms of these notations. Also see Table 2.1.

A *two-way (non-sensing)*[1] *k-head probabilistic finite (-state) automaton* is a probabilistic finite automaton having k heads on the input string, which is delimited by two distinct endmarkers. A configuration of the probabilistic automaton is defined by its state and the positions of the k heads on the input string. From each configuration the automaton executes transitions to next configurations with probabilities from the set $\{0, 1/2, 1\}$.[2] The transition probability to another configuration depends on the present state and the symbols scanned by the k heads (and not on the fact that some heads do or do not have the same position). The acceptance probability of a probabilistic finite automaton for an input w is the probability of eventually reaching the accepting states when processing w. The automaton accepts the input string if this probability is larger than 1/2.

Formally, the definition for two-way (non-sensing) multihead probabilistic automata is as follows.

DEFINITION 2.1. *A two-way (non-sensing) k-head probabilistic finite automaton is a structure*

$$S = (Q, \Sigma, \delta, q_0, Q_{acc}, Q_{rej}),$$

where

- *Q is a finite, nonempty set of states;*
- *$q_0 \in Q$ is the initial state;*

[1]All the multihead finite automata used in this paper have non-sensing heads, i.e., heads that do not detect each other's position. Sometimes we drop the word "non-sensing," but this should not create misunderstanding.

[2]See [29] for results concerning probabilistic automata with transition probabilities from more general sets.

- *Q_{acc} and Q_{rej} are disjoint sets of accepting and rejecting states ($Q_{acc}, Q_{rej} \subset Q$);*
- *Σ is a nonempty input alphabet that does not contain the symbols (left and right endmarkers) $\cancel{c}$ and $\$$;*
- *$\delta : Q \times (\Sigma \cup \{\cancel{c}, \$\})^k \times Q \times (\{-1, 0, 1\})^k \to \{0, 1/2, 1\}$ is the transition function that, depending on the current state and the symbols scanned by the k heads, may change the automaton state and increment, decrement, or leave unchanged its head positions with probabilities from the set $\{0, 1/2, 1\}$. More precisely, the meaning of $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k)$ is as follows: if the automaton is in state q scanning the symbols $s_1, \ldots, s_k$, then with probability $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k)$ it enters state q' and moves each input head $i \in \{1, \ldots, k\}$ one symbol in direction d_i (left if $d_i = -1$, right if $d_i = 1$, stationary if $d_i = 0$). There are the following restrictions on δ:*
  - *$\forall (q, s_1, \ldots, s_k) \in Q \times (\Sigma \cup \{\cancel{c}, \$\})^k$, it holds that*

$$\sum_{q', d_1, \ldots, d_k} \delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k) = 1$$

    (i.e., δ is a probability function).
  - *$\delta(q, s_1, \ldots, s_{j-1}, \cancel{c}, s_{j+1}, \ldots, s_k, q', d_1, \ldots, d_{j-1}, -1, d_{j+1}, \ldots, d_k) = 0$ (i.e., no head can move to the left of the left endmarker $\cancel{c}$) and $\delta(q, s_1, \ldots, s_{j-1}, \$, s_{j+1}, \ldots, s_k, q', d_1, \ldots, d_{j-1}, 1, d_{j+1}, \ldots, d_k) = 0$ (i.e., no head can move to the right of the right endmarker $\$$).*

Without loss of generality, we can assume that the accepting (Q_{acc}) and rejecting (Q_{rej}) states are halting states (i.e., the automaton stops when it enters them). For an input string x, the acceptance probability of a k-head probabilistic finite automaton is the probability to reach the accepting state when processing x. The input x is accepted, if its acceptance probability is larger than $1/2$.

The definition of probabilistic Turing machine is standard [13].

We recall a classification of probabilistic automata depending on the type of acceptance. If there is an interval around $1/2$ such that for every input string the acceptance probability never falls inside, then the automaton is called (*two-sided*) *bounded-error*. If this is not the case, it is called *unbounded-error*. If for every input string the acceptance probability is either 0 or above $1/2$, the automaton is called *one-sided error* (or *randomized*).

Several equivalent definitions of AM-games or interactive proof systems with public coins have been published in the literature [1, 15]. See [4] for an extensive survey. For our purpose, the most appropriate is the definition based on automata with guess (i.e., nondeterministic) states and random (i.e., probabilistic) states [15]. Such an automaton A accepts an input string x if there is a strategy such that if the nondeterministic transitions are decided according to this strategy, then the acceptance probability of x by A is larger than $1-\epsilon$, for some $\epsilon < 1/2$. A rejects x if for any strategy (used to "decide" the nondeterministic transitions) the acceptance probability of x by A is less than $1/2$. If $\epsilon = 1/2$, we obtain an unbounded-error variant of AM-games (UAM-games) called games against nature [33]. In this paper, we focus on cases where the machine A is a multihead finite automaton or a logspace Turing machine with both nondeterministic and probabilistic states that recognizes with bounded-error or unbounded-error.[3] We call these devices either probabilistic-plus-nondeterministic automata or automata with both probabilistic and nondeterministic states.

The formal definition of a two-way multihead finite automaton with both nondeterministic and probabilistic states and of its (nondeterministic) strategy is a straightforward generalization of the definition for the corresponding two-way one-head finite automaton [6].

DEFINITION 2.2. *A two-way (non-sensing) k-head finite automaton with nondeterministic and probabilistic states is a structure* $S = (Q, \Sigma, \delta, q_0, N, R, Q_{acc}, Q_{rej})$, *where*

- Q *is a finite, nonempty set of states partitioned into the (disjoint) sets* N, R, Q_{acc}, Q_{rej};
- Σ *is the input alphabet and does not contain the symbols (left and right endmarkers)* $\cancel{c}$ *and* \$;
- $q_0 \in Q$ *is the initial state;*
- N *is the subset of nondeterministic states;*
- R *is the subset of probabilistic states;*
- Q_{acc} *and* Q_{rej} *are the sets of accepting and rejecting states;*
- $\delta : Q \times (\Sigma \cup \{\cancel{c}, \$\})^k \times Q \times (\{-1, 0, 1\})^k \to \{0, 1/2, 1\}$ *is the transition function; the meaning of* $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k)$ *is as follows: if* $q \in R$ *and the automaton is in state* q *reading the symbols* $s_1, \ldots, s_k$, *then with probability* $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k)$ *the automaton enters state* q' *and moves each head* $j \in \{1, \ldots, k\}$ *one symbol in direction* d_j *(left if* $d_j = -1$, *right if* $d_j = 1$, *stationary if* $d_j = 0$*); if the current state is* $q \in N$ *and the symbols read are* $s_1, \ldots, s_k$, *then the automaton nondeterministically chooses some* q' *and* $d_1, \ldots, d_k$ *such that* $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k) = 1$, *enters state* q', *and moves each head* $j \in \{1, \ldots, k\}$ *one symbol in direction* d_j. *There are the following restrictions on* δ:
  - $\forall q \in R, \forall (s_1, \ldots, s_k) \in (\Sigma \cup \{\cancel{c}, \$\})^k$, *it holds that*
  $$\sum_{q', d_1, \ldots, d_k} \delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k) = 1$$
  (i.e., δ *is a transition probability function);*
  - $\forall q \in N, \forall (s_1, \ldots, s_k) \in (\Sigma \cup \{\cancel{c}, \$\})^k, \forall (q', d_1, \ldots, d_k) \in Q \times (\Sigma \cup \{\cancel{c}, \$\})^k$ *it holds that* $\delta(q, s_1, \ldots, s_k, q', d_1, \ldots, d_k) \in \{0, 1\}$ *(i.e.,* δ *is a nondeterministic transition);*
  - $\delta(q, s_1, \ldots, s_{j-1}, \cancel{c}, s_{j+1}, \ldots, s_k, q', d_1, \ldots, d_{j-1}, -1, d_{j+1}, \ldots, d_k) = 0$ *(i.e., no head can move to the left of the left endmarker* $\cancel{c}$*) and* $\delta(q, s_1, \ldots, s_{j-1}, \$, s_{j+1}, \ldots, s_k, q', d_1, \ldots, d_{j-1}, 1, d_{j+1}, \ldots, d_k) = 0$ *(i.e., no head can move to the right of the right endmarker* \$*).*

DEFINITION 2.3. *For any input string* $x = x_0 x_1 \ldots x_{n+1}$ *(where* $x_0 = \cancel{c}$ *and* $x_{n+1} = \$$ *are the endmarkers) a (nondeterministic) strategy on* x *is a function*

$$S_x : N \times (\{0, \ldots, n+1\})^k \to Q \times (\{-1, 0, 1\})^k$$

such that $\delta(q, s_1, \ldots, \ldots, s_k, q', d_1, \ldots, d_k) = 1$ *whenever* $S_x(q, p_1, \ldots, p_k) = (q', d_1, \ldots, d_k)$ *and* $x_{p_l} = s_l, \forall l \in \{1, \ldots, k\}$. (*Discussions on other possible definitions for (nondeterministic) strategy and their equivalence can be found in* [3].)

[3]Using a different terminology, we investigate properties of interactive proofs with public coins and multihead probabilistic finite automata or logspace probabilistic Turing machines as verifiers that recognize with bounded-error or unbounded-error.

We recall that an automaton A with nondeterministic and probabilistic states accepts an input string x if there is a strategy by which the nondeterministic transitions are decided such that the acceptance probability of x by A is larger than $1-\epsilon$, where $\epsilon < 1/2$ for **AM**-games and $\epsilon = 1/2$ for **UAM**-games. A rejects x if for any strategy (used to "decide" the nondeterministic transitions) the acceptance probability of x by A is less than $1/2$. Without loss of generality, we can assume that Q_{acc} and Q_{rej} contain only halting states.

A multihead automaton has *sweeping* heads if it may reverse the direction of motion of any head only when that head is scanning the right or left endmarker of the input string. A multihead automaton has non-sensing heads if they cannot detect each other's position. We recall that all the multihead automata used in this article have non-sensing heads and that we sometimes omit the word "non-sensing."

Our notations are as follows:

- 2pfa(k) and 2bpfa(k) (1pfa(k) and 1bpfa(k)) denote two-way (one-way) k-head unbounded-error and respectively bounded-error probabilistic finite automata.
- **2PFA**(k), **2BPFA**(k), and **2RPFA**(k) (**1PFA**(k), **1BPFA**(k), and **1RPFA**(k)) denote the classes of languages recognized by the corresponding two-way (one-way) unbounded-error, bounded-error, and one-sided-error probabilistic finite automata, respectively.
- **1PCM**(1) is the class of languages recognized by one-way one-head one-counter unbounded-error probabilistic finite automata.
- **2DFA**(k) and **2NFA**(k) denote the classes of languages recognized by two-way k-head deterministic and nondeterministic finite automata, respectively.
- **DSPACE**(S) is the class of languages recognized by $O(S)$-space-bounded deterministic Turing machines.
- **DTIME**(t) is the class of languages recognized by $O(t)$-time bounded deterministic Turing machines.
- **L**, **NL**, and **AL** are the classes of languages recognized by logspace deterministic, nondeterministic, and alternating Turing machines, respectively.
- **PL**, **BPL**, and **RL** are the classes of languages recognized by unbounded-error, bounded-error, and one-sided-error probabilistic Turing machines, respectively.
- 2pnfa(k) denotes two-way k-head (unbounded-error) finite automata with both probabilistic and nondeterministic states (or, equivalently, **UAM**-games with a two-way k-head probabilistic finite automaton as verifier).
- (P+N)TM denotes Turing machines with both probabilistic and nondeterministic states.
- **AM**($2pfa(k)$) and **AM**($1pfa(k)$) (**UAM**($2pfa(k)$) and **UAM**($1pfa(k)$)) are the classes of languages recognized by **AM**-games (**UAM**-games) with two-way and one-way k-head probabilistic finite automaton, respectively, as verifier.
- **UAM**($1pcm(1)$) is the class of languages recognized by **UAM**-games with one-way one-head one-counter probabilistic finite automaton as verifier.
- **AML** and **UAML** denote the classes of languages recognized by **AM**-games (**UAM**-games) with logspace probabilistic Turing machines as verifiers.
- $\mathbf{C}_{poly}$ is the subclass of languages recognized by the machines that define the class **C** but are restricted to work in polynomial time.
- For any complexity class **C**, $\overbrace{\mathbf{C}}$ is the subclass of unary languages defined

by $\widehat{\mathbf{C}} = \{L_1 \mid L_1 \subseteq \{1^{2^n} \mid n \in \mathbf{N}\}, L_1 \in \mathbf{C}\}$.

- $\subset$ and $\subsetneqq$ denote inclusion (possible not proper) and proper inclusion, respectively.
- $|x|$ is the length of the string x.
- $\mathbf{N}$ is the set of natural numbers (note that we use N to denote the set of nondeterministic states of an automaton).
- By probabilistic-plus-nondeterministic automata we mean automata with both probabilistic and nondeterministic states.

Using the notations introduced above, the results presented in this article are as follows.

- The "equivalence" between the exact space (as opposed to big "O" only) of logspace Turing machines with probabilistic and nondeterministic states, and the number of heads of multihead finite automata with the same kind of states:
  $\mathbf{AML} = \bigcup_{k=1}^{\infty} \mathbf{AM}(2pfa(k))$, $\mathbf{UAML} = \bigcup_{k=1}^{\infty} \mathbf{UAM}(2pfa(k))$,
  $\mathbf{AML}_{poly} = \bigcup_{k=1}^{\infty} \mathbf{AM}_{poly}(2pfa(k))$, and
  $\mathbf{UAML}_{poly} = \bigcup_{k=1}^{\infty} \mathbf{UAM}_{poly}(2pfa(k))$ (Theorem 3.1).
  These relations parallel those recently noticed in the setting of probabilistic computation [29]. The proofs of these decompositions are straightforward adaptations of the proofs of their corresponding deterministic and nondeterministic results [16].
- For $\mathbf{AM}$-games and $\mathbf{UAM}$-games with two-way multihead probabilistic finite-state verifiers the head hierarchy is proper (even for languages over a single-letter alphabet), i.e., for $k \geq 2$,
  $\mathbf{AM}(2pfa(k)) \subsetneqq \mathbf{AM}(2pfa(k+1))$, $\mathbf{UAM}(2pfa(k)) \subsetneqq \mathbf{UAM}(2pfa(k+1))$,
  $\mathbf{UAM}_{poly}(2pfa(k)) \subsetneqq \mathbf{UAM}_{poly}(2pfa(k+1))$ (Theorems 3.2 and 3.7).
  The proofs of these separations are less straightforward adaptations of their deterministic and nondeterministic variants obtained by Monien [30, 31]. Similar separations hold for multihead unbounded-error and one-sided-error probabilistic finite automata [29].
- Using various techniques, we show logspace reductions ($\leq_{log}$) among probabilistic (probabilistic-plus-nondeterministic) complexity classes, as follows:
  - Logspace (bounded-error and one-sided-error) complexity classes are reducible to classes of languages recognized by the corresponding two-way (non-sensing and sweeping) two-head finite automata:
    $\mathbf{BPL} \leq_{log} \mathbf{2BPFA}(2)$, $\mathbf{BPL}_{poly} \leq_{log} \mathbf{2BPFA}_{poly}(2)$,
    $\mathbf{RL} \leq_{log} \mathbf{2RPFA}(2)$, $\mathbf{RL}_{poly} \leq_{log} \mathbf{2RPFA}_{poly}(2)$ (Theorem 4.1), and
    $\mathbf{P} = \mathbf{AML} \leq_{log} \mathbf{AM}(2pfa(2))$ (Theorem 4.9).
  - Polynomial-time logspace (bounded-error and one-sided-error) complexity classes are reducible to languages recognized by the corresponding one-way non-sensing multihead finite automata that additionally move at least one head at each computation step:
    $\mathbf{BPL}_{poly} \leq_{log} \mathbf{1BPFA}(4)$, $\mathbf{RL}_{poly} \leq_{log} \mathbf{1RPFA}(4)$ (Theorem 4.3) and
    $\mathbf{AML}_{poly} \leq_{log} \mathbf{AM}(1pfa(4))$ (Theorem 4.9).
  - Logspace unbounded-error complexity classes are reducible to classes of languages recognized by the corresponding one-way non-sensing two-head finite automata and one-way one-head one-counter finite automata that additionally move at least one head at each computation step:
    $\mathbf{PL} \leq_{log} \mathbf{1PFA}(2)$, $\mathbf{PL} \leq_{log} \mathbf{1PCM}(1)$ (Theorem 4.4) and

$\mathbf{P} = \mathbf{UAML} \leq_{log} \mathbf{UAM}(1pfa(2))$, $\mathbf{P} \leq_{log} \mathbf{UAM}(1pcm(1))$ (Theorem 4.9).

It follows that **PL** (and **P**) can be defined as the classes of languages logspace reducible to languages recognized by one-way two-head or one-way one-head one-counter finite automata that recognize with unbounded-error and have probabilistic (probabilistic and nondeterministic, respectively) states.

Some of these results parallel reductions proven by Hartmanis [16] and Sudborough [44] for the nondeterministic setting. However, our proofs are more complex and require a number of innovations.

- To prove the logspace reductions of **PL** we design probabilistic variants of Savitch's maze threading problem that are logspace complete for **PL**, and can be recognized by simple one-way automata. Surprisingly, these natural variants have not been noticed so far, despite the lack of natural problems known to be complete for **PL**. (Note that several nondeterministic variants of maze threading problems were already known in the early 1970s.) Similarly, to prove logspace reductions of **P**, we present probabilistic-plus-nondeterministic variants of the maze threading problem that are logspace complete for **P**. Each of these complete problems can be recognized by some corresponding one-way non-sensing two-head or one-way one-head one-counter finite automata. These one-way machines use a new recognition technique: during their computation on an input string the automata simulate nondeterministic guesses by "probabilistic guesses" (in a standard way); however, this method alone may change the acceptance probability of the input string and, consequently, can change the recognized languages; fortunately, the automata have an elegant way to compensate for the acceptance probability changes, using only a small amount of additional randomness.

We conclude this section with some additional comments on the results we have listed.

- We show that translational methods used in the 1960s and the 1970s for separating complexity classes [34, 30, 31] can be adapted to the settings of probabilistic and probabilistic-plus-nondeterministic computations. Here is a further example of the strength of these methods: Dwork and Stockmeyer proved that the set of palindromes, i.e., $\{x \in \Sigma^* | x = x^R\}$, separates $\mathbf{AM}(2pfa(1))$ from $\mathbf{AML} = \mathbf{P}$ [8]; we prove that there are entire natural hierarchies between $\mathbf{AM}(2pfa(1))$ and **P**, which can be separated even by languages over a one-letter alphabet.
- Our reductions $\mathbf{BPL}_{poly} \leq_{log} \mathbf{1BPFA}(4)$, $\mathbf{RL}_{poly} \leq_{log} \mathbf{1RPFA}(4)$, and $\mathbf{AML}_{poly} \leq_{log} \mathbf{AM}(1pfa(4))$ are somewhat surprising because the reduction of two-way bounded-error probabilistic computation to one-way probabilistic computation does not naturally preserve the separation from 1/2 of the error-bound (see [22], for example).
- Our reduction $\mathbf{PL} \leq_{log} \mathbf{1PFA}(2)$ is stronger than the well-known result $\mathbf{PL} = \mathbf{PL}_{poly}$ [21]. Note that **PL** contains several matrix inversion problems whose solutions seem to require rather complex computations. It is surprising that these problems can be reduced to languages recognized by one-way non-sensing two-head or one-way one-head one-counter probabilistic finite automata, which do not seem to have strong computational capabilities.

- Using our result $\mathbf{P} = \mathbf{UAML} \leq_{log} \mathbf{UAM}(1pfa(2))$, we can reason that

$$\begin{aligned}\mathbf{P} \subset \mathbf{DSPACE}(\log^k n) &\Leftrightarrow \mathbf{AM}(2pfa(2)) \subset \mathbf{DSPACE}(\log^k n)\\ &\Leftrightarrow \mathbf{UAM}(1pfa(2)) \subset \mathbf{DSPACE}(\log^k n)\\ &\Leftrightarrow \mathbf{UAM}(1pcm(1)) \subset \mathbf{DSPACE}(\log^k n).\end{aligned}$$

  This apparent simplification might eventually lead to the possible inclusion of $\mathbf{P}$ in a small-space-bounded complexity class, and to its separation from $\mathbf{PSPACE}$.

3. Heads hierarchy.

This section presents analogues, in the setting of probabilistic-plus-nondeterministic computation, of results presented in [29] for probabilistic computation.

First, we notice the equality of the class of languages recognized by **AM**-games and **UAM**-games with two-way multihead probabilistic finite-state verifiers and the class of languages recognized by the same games with logspace probabilistic Turing machines as verifiers. Next, we separate head hierarchies for **AM**-games and **UAM**-games with two-way multihead probabilistic finite-state verifiers.

THEOREM 3.1. *$\mathbf{AML}_{poly} = \bigcup_{k=1}^{\infty} \mathbf{AM}_{poly}(2pfa(k))$, $\mathbf{UAML}_{poly} = \bigcup_{k=1}^{\infty} \mathbf{UAM}_{poly}(2pfa(k))$, $\mathbf{AML} = \bigcup_{k=1}^{\infty} \mathbf{AM}(2pfa(k))$, $\mathbf{UAML} = \bigcup_{k=1}^{\infty} \mathbf{UAM}(2pfa(k))$.*

Proof. Adapt the proof for the nondeterministic case [16]. See the appendix, section 6. ☐

In what follows we show separations of head hierarchies. The main result is the following theorem.

THEOREM 3.2. *$\forall\, k \in \mathbf{N}$, $k \geq 2$, $\mathbf{AM}(2pfa(k)) \subsetneq \mathbf{AM}(2pfa(k+1))$, $\mathbf{UAM}(2pfa(k)) \subsetneq \mathbf{UAM}(2pfa(k+1))$, $\mathbf{UAM}_{poly}(2pfa(k)) \subsetneq \mathbf{UAM}_{poly}(2pfa(k+1))$.*

This theorem follows trivially from Theorem 3.7, i.e., from its stronger version over a single-letter alphabet. To prove Theorem 3.7 we follow the same outline as in Monien's proof for the deterministic and nondeterministic automata [31]. We prove first a coarse separation (Theorem 3.3) and then we refine this separation using Lemmas 3.4–3.6.

The proofs of Lemmas 3.4–3.6 are only sketched since they are adaptations of Monien's proofs in the deterministic and nondeterministic settings. In these proofs we use languages $X \subseteq \{1^{2^n} \mid n \in \mathbf{N}\}$ and the family of padding functions

$$f_k : \{1^{2^n} \mid n \in \mathbf{N}\} \to \{1^{2^n} \mid n \in \mathbf{N}\} \text{defined by} f_k(1^{2^n}) = 1^{2^{kn}}.$$

The advantage of using these particular languages is in simplifying the unpadding, encoding, and decoding procedures. Note that checking whether a string is of the form 1^{2^n} can be done deterministically with finite control and only two non-sensing heads. (Repeatedly moving one head twice as fast as the other, in the opposite direction, we can achieve division of the input string length by 2, 2^2, 2^3, etc. Each result of this division is trivially encoded by the position of one head on the input string.)

We recall that, for every class of languages C, $\overbrace{C}$ denotes the languages of type X contained in C.

We show only $\overbrace{\mathbf{AM}(2pfa(k))} \subsetneq \overbrace{\mathbf{AM}(2pfa(k+1))}$ $\forall k \geq 2$ since the proofs for the relations $\overbrace{\mathbf{UAM}(2pfa(k))} \subsetneq \overbrace{\mathbf{UAM}(2pfa(k+1))}$ and $\overbrace{\mathbf{UAM}_{poly}(2pfa(k))} \subsetneq \overbrace{\mathbf{UAM}_{poly}(2pfa(k+1))}$ are similar.

Theorem 3.3 (coarse separation). *For every natural number k,*

$$\overbrace{\mathbf{AM}(2pfa(k))} \subsetneq \overbrace{\mathbf{AML}}\,.$$

Proof. We show that $\exists\alpha > 0$ such that for any $k \in \mathbf{N}$ it holds $\mathbf{AM}(2pfa(k)) \subseteq \mathbf{DTIME}(n^{k\alpha})$. From $\overbrace{\mathbf{DTIME}(n^{k\alpha})} \subsetneq \overbrace{\mathbf{P}} = \overbrace{\mathbf{AML}}$, we obtain the coarse separation.

In what follows we present more details. Let $L \in \mathbf{AM}(2pfa(k))$. Deciding whether a length-n input string x belongs to L is equivalent to solving a linear programming problem with $O(n^k)$ variables. (A detailed proof for the similar case $\mathbf{AML} \subset \mathbf{P}$ can be found in [3].) Using the result of Khachiyan [23], this can be deterministically done in time $O((n^k)^\alpha)$, for some constant α. It follows $L \in \mathbf{DTIME}(n^{k\alpha})$. We obtain $\mathbf{AM}(2pfa(k)) \subset \mathbf{DTIME}(n^{k\alpha})$. As a corollary, we have $\overbrace{\mathbf{AM}(2pfa(k))} \subset \overbrace{\mathbf{DTIME}(n^{k\alpha})}$. To prove $\overbrace{\mathbf{DTIME}(n^{k\alpha})} \subsetneq \overbrace{\mathbf{P}}$, we use a universal deterministic Turing machine that stops its computation after $n^{2k\alpha+1}$ steps and diagonalizes over all one-tape deterministic Turing machines that run in time less than $n^{2k\alpha+1}$. (We recall that any multitape deterministic Turing machine that runs in time $O(n^{k\alpha})$ can be simulated by a one-tape deterministic Turing machine that runs in time $O(n^{2k\alpha})$.) The witness language consists of all the strings of the form 1^{2^m}, where m is a valid encoding of a one-tape deterministic Turing machine M_m, and 1^{2^m} is not accepted by M_m in time $(2^m)^{2k\alpha+1}$. ∎

Lemma 3.4. *For all languages $X \in \overbrace{\mathbf{AML}}$ there is a natural number u such that*

$$f_u(X) \in \overbrace{\mathbf{AM}(2pfa(3))}\,.$$

Proof (sketch). For every language $X \in \overbrace{\mathbf{AML}}$ we modify the Turing machine with probabilistic and nondeterministic states (i.e., a (P+N)TM) that recognizes it in the following way: The worktape of the new (P+N)TM has four tracks that store the binary representation of the input string, the position of the input-tape head, the content of the worktape of the original (P+N)TM, and the position of the worktape head on the original work tape. (Without loss of generality we suppose that the original Turing machine has only one head on its worktape.) All this information can be represented in binary using space $c\log n$, where c is a computable constant. After the deterministic computation of the binary representation of the input string, the new (P+N)TM uses only the worktape head in simulating the original (P+N)TM. This new (P+N)TM can be simulated by a three-counter machine with both probabilistic and nondeterministic states that uses two counters to encode the worktape contents from the left and right, respectively, of the work head and another counter to implement divisions and multiplication by 2 and to increment and decrement of the first two counters. The only problem is the capacity requirement of the three counters. We notice that these counters can be simulated by three heads that move on a "stretched" version of the input. If we choose $u = c$, then the padding function f_u can perform the needed "stretch" operation. ∎

Lemma 3.5. *For all languages $X \in \overbrace{\mathbf{AML}}$ and for $u, v \geq 1$,*

$$f_u(X) \in \overbrace{\mathbf{AM}(2pfa(v))} \Rightarrow X \in \overbrace{\mathbf{AM}(2pfa(u \cdot v))}\,.$$

Proof (sketch). A natural number in the interval $[0, 2^{u\cdot n})$ can be represented in base 2^n by u integers from the interval $[0, 2^n)$. Therefore, the position on the "stretched" input of each head of a 2pnfa(v) that recognizes a language of the form $f_u(X)$ can be encoded by the positions on a normal ("unstretched") input of u heads of a 2pnfa($u \cdot v$). The simulation is straightforward. □

LEMMA 3.6. *For all languages $X \in \overbrace{\mathbf{AML}}$ and for $u > v > 1$,*

$$f_{u+1}(X) \in \overbrace{\mathbf{AM}(2pfa(v))} \Rightarrow f_u(X) \in \overbrace{\mathbf{AM}(2pfa(v+1))}.$$

Proof (sketch). Let A be a 2pnfa(v) that recognizes $f_{u+1}(X)$. We build a 2pnfa(v+1) (called B) that recognizes $f_u(X)$. In order to simulate A's moves, B encodes each head position h_i of A ($h_i \in [0, 2^{(u+1)n}+1]$) by the position of a corresponding head g_i ($g_i \in [0, 2^{u\cdot n}+1]$) and by an additional number $x_i \in [0, 2^n)$ such that $h_i = g_i + x_i \cdot 2^{u\cdot n}$ ($\forall\ i = 1, \ldots, v$). All the numbers $x_i, i = 1, \ldots, v$ are stored as the coefficients of the representation in base 2^n of a number encoded by the $(v + 1)$st head of V. The condition $u > v$ guarantees that the $(v+1)$th head of V can represent a number that is large enough to make that representation possible. The details of this simulation are similar to those from Monien's simulation for the deterministic case [31]. □

Using the relations $\mathbf{P} = \mathbf{AML} = \mathbf{UAML} = \mathbf{UAML}_{poly}$ [3], we can prove that Theorem 3.3 and Lemmas 3.4–3.6 hold for **UAM**-games as well. Furthermore, by adapting the technique of Monien [31], we obtain the following theorem.

THEOREM 3.7. $\forall\ k \in \mathbf{N}$, $k \geq 2$,

$\overbrace{\mathbf{AM}(2pfa(k))} \subsetneq \overbrace{\mathbf{AM}(2pfa(k+1))}$, $\overbrace{\mathbf{UAM}(2pfa(k))} \subsetneq \overbrace{\mathbf{UAM}(2pfa(k+1))}$,
$\overbrace{\mathbf{UAM}_{poly}(2pfa(k))} \subsetneq \overbrace{\mathbf{UAM}_{poly}(2pfa(k+1))}$.

Proof (sketch). We prove the first claim, the other proofs being similar. Suppose that for some $h \geq 2$ we have $\overbrace{\mathbf{AM}(2pfa(h))} = \overbrace{\mathbf{AM}(2pfa(h+1))}$. We prove that this assumption implies $\overbrace{\mathbf{AM}(2pfa(h(h+1)))} = \overbrace{\mathbf{AML}}$, which contradicts Theorem 3.3.

Let $H \in \overbrace{\mathbf{AML}}$. There is $p \in \mathbf{N}$ such that $f_p(H) \in \overbrace{\mathbf{AM}(2pfa(3))}$ (by Lemma 3.4). As a result $f_p(H) \in \overbrace{\mathbf{AM}(2pfa(h))}$. If $p > h + 1$, then, by Lemma 3.6, $f_{p-1}(H) \in \overbrace{\mathbf{AM}(2pfa(h+1))} = \overbrace{\mathbf{AM}(2pfa(h))}$, and so on, until $p = h + 1$. If $p \leq h + 1$, by Lemma 3.5 it follows that $H \in \overbrace{\mathbf{AM}(2pfa(h(h+1)))}$. □

4. Efficient reductions of logspace probabilistic complexity classes. In this section we present logspace reductions of logspace probabilistic (probabilistic-plus-nondeterministic) complexity classes to classes of languages recognized by one-way or two-way multihead probabilistic (probabilistic-plus-nondeterministic) automata. Our simulations could also help to investigate space upper bounds for deterministic space simulation of probabilistic (probabilistic-plus-nondeterministic) automata. Some of our results parallel reductions obtained in the nondeterministic setting by Hartmanis [16] and Sudborough [44] and in the alternating setting by King [24].

The main steps in proving our reductions are as follows. The classes of languages recognized by polynomial-time logspace Turing machines can be recognized by the corresponding two-way multihead finite-state automata; these languages are logspace reducible to languages recognized by $O(n^2)$-time-bounded two-way non-sensing and sweeping two-head finite-state automata that move at least one head at each computation step, which languages, in turn, are reducible to languages recognized by one-way non-sensing multihead finite-state automata that have only a small number of heads and move at least one head at each computation step.

First we state a general theorem that relates the classes of languages recognized by logspace Turing machines to the classes of languages recognized by the corresponding two-way two-head finite-state automata.

THEOREM 4.1. *For any of the computation modes nondeterministic, probabilistic, alternating, and probabilistic-plus-nondeterministic, the languages recognized by logspace Turing machines are logspace reducible to languages recognized, with only a multiplicative polynomial-time loss, by the corresponding two-way two-head non-sensing and sweeping finite-state automata that move at least one head at each computation step.*

Proof. We use the following transformation defined by Monien [30].

Let Σ be an alphabet and $\vdash$ and $\dashv$ be (pseudo-endmarker) symbols not in Σ. For any $h \geq 1$, the transformation $g_{\Sigma,h} : \Sigma^* \longrightarrow ((\Sigma \cup \{\vdash, \dashv\})^h)^*$ is defined as follows: For any positive integer m and any length-m string $a_1 \dots a_m \in \Sigma^m$,

$$g_{\Sigma,h}(a_1 \dots a_m) = b_0 b_1 \dots b_{n^h-1},$$

where $n = m+2$, $a_0 = \vdash$, $a_{n-1} = \dashv$, and $b_j = (a_{i_1}, \dots, a_{i_h})$ for any $j = i_1 + i_2 n + \cdots + i_h n^{h-1}$ with $0 \leq i_p \leq n-1$, $\forall p \in \{1, \dots, k\}$.

First we recall that, for all the computation modes mentioned above, a language L recognized by a logspace Turing machine can be recognized by a corresponding two-way non-sensing multihead finite automaton with only polynomial-time loss [16, 24, 29]. It remains to show that for any $h \in \mathbf{N}$ and any language L over Σ^*, recognized in $O(f(n))$-time by a two-way non-sensing h-head finite automaton, the language $g_{\Sigma,h}(L)$ is recognized in $O(f(n)n^h)$-time by a two-way non-sensing and sweeping two-head finite automaton that moves at least one head at each computation step.

More precisely, we prove that any two-way non-sensing h-head finite automaton processing a string $a_1 \dots a_m \in L$ can be simulated (with a multiplicative $O(n^h)$-time loss) by a two-way non-sensing and sweeping two-head finite automaton processing the string $g_{\Sigma,h}(a_1 \dots a_m)$.

First, at the beginning of its computation, the simulating automaton checks whether the length-u input string $b_0 b_1 \dots b_{u-1}$ is of the form $g_{\Sigma,k}(a_1 \dots a_m)$. This check can be done deterministically in $O(u)$ time with two sweeping heads. If the input string is not of the right form the automaton rejects it. Otherwise it starts the simulation. Note that in this case $u = (m+2)^h$.

In the transformation $g_{\Sigma,h}(a_1 \dots a_m) = b_0 b_1 \dots b_{n^h-1}$ each symbol b_j encodes h ordered symbols $a_{i_1}, \dots, a_{i_h}$. Thus, one head scanning the string $b_0 b_1 \dots b_{n^h-1}$ (called the "encoding head") keeps track of the symbols scanned by h heads on the string $a_0 \dots a_n$. Incrementing (decrementing) the position of the ith head on the string $a_0 \dots a_n$ is equivalent to moving the encoding head n^{i-1} symbols right (left) on the string $b_0 b_1 \dots b_{n^h-1}$. To perform these operations, the encoding head deterministically cooperates with an auxiliary head, whose normal position is at one end of the input string. Note that the auxiliary head can easily count n^{i-1} steps using the particular form of $g_{\Sigma,k}(a_1 \dots a_m)$ and then move to the other end of the input string. The number of computation steps required to perform such an operation is $O(n^h)$. When the encoding head needs to change direction inside the input string, it switches roles with the auxiliary head coming from the opposite direction. To perform this operation the auxiliary head moves first at the end of the input string currently headed toward by the encoding head. Next both heads move toward each other with the same speed until the encoding head reaches the end of the string. At that moment the auxiliary head is in the position where the encoding head had to change direction and it becomes the new encoding head.

It follows that both the encoding and auxiliary heads can be non-sensing and sweeping. Also, we can pad the computation so that at least one head moves at each computation step. To achieve this, it suffices to move the auxiliary head from one end of the input string to the other end for each computation step when the encoding head does not move. The time required to perform this operation is $O(n^h)$.

Overall, the running time of the simulation is $O(n^h)$ times the running time of the simulated automaton. □

OBSERVATION 1. *Theorem* 4.1 *can be stated in the following, more precise, form: For any $h \in \mathbf{N}$ and any of the computation modes nondeterministic, probabilistic, alternating, and probabilistic-plus-nondeterministic, the languages recognized by two-way non-sensing h-head finite automata are logspace reducible (by a transformation that converts a length-n string into a length-n^h string) to languages recognized, with only a multiplicative $O(n^h)$-time loss, by the corresponding two-way non-sensing and sweeping two-head finite-state automata that move at least one head at each computation step.*

For our purpose this form is more useful. These reductions to languages recognized by automata having at least one moving head at each computation step are used in the proof of Proposition 4.2.

As in the nondeterministic setting, we can ask whether **PL**, **BPL**, **RL** are logspace reducible to classes of languages recognized by the corresponding *one-way* multihead probabilistic finite automata. Sudborough proved $\mathbf{NL} \leq_{log} \mathbf{1NFA}(2)$. Unfortunately, his technique uses properties of nondeterministic computation (like particular forms of some logspace complete problems for **NL**) that are not known to hold for probabilistic computation. However, by adding to his proofs ([44], Theorem 1) the idea to "reuse one-way heads," we obtain the following proposition.

PROPOSITION 4.2. *For any integer k, $k > 1$, and any language L recognized in $O(n^k)$-time by a two-way non-sensing and sweeping two-head finite automaton that moves at least one head at each computation step, there is a logspace transformation f such that $f(L)$ is recognized by a one-way non-sensing $(k+2)$-head finite automaton that moves at least one head at each computation step.*

Proof. Without loss of generality we consider a language L recognized in time less than n^k by a two-way (sweeping, non-sensing) two-head finite automaton A, for some $k \in \mathbf{N}$. We chose the transformation f defined by $f(x) = u = (axbbx^Ra)^{n^k}$, where a, b are symbols not contained in the alphabet of L and x^R is the reverse of the length-n string x.

We show that $f(L)$ is recognized by a one-way $(k+2)$-head automaton. First, notice that the form of each input string u can be deterministically checked using only $k+1$ one-way heads as follows:

- Two one-way heads check whether u is of the form $ax_1bbx_2aax_3bbx_4aa\dots bx_{2m}a$ and whether $x_1 = x_3 = \cdots = x_{2m-1}$ and $x_2 = x_4 = \cdots = x_{2m}$. (All they have to do is to keep a constant distance of $|x_1| + |x_2| + 4$ between them.) The head from behind is also used for the next check.
- Two one-way heads check that $|x_1| = |x_2|$ (assuming that the previous check is done this is equivalent to $|x_{2i-1}| = |x_{2i}|$, $\forall\ i \in \{1, \cdots, m\}$). (In what follows, n denotes $|x_1|$.) Then, the same heads check whether x_{2i} is the reverse of x_{2i+1}, $\forall\ i \in \{1, \cdots, m\}$. Assuming that $x_1 = x_3 = \cdots = x_{2m-1}$

and $x_2 = x_4 = \cdots = x_{2m}$, this is equivalent to checking whether the symbol from position i of x_{2i} is equal to the symbol from position $n - i + 1$ of x_{2i+1}, for $i = 1, \ldots, n$. To do this check, for each $i = 1, \ldots, n$ one head scans the ith symbol from the beginning of the string x_{2i} and compares it with the ith symbol from the end of the string x_{2i+1} (which is scanned by the second head), as follows: When the two heads sweep the substring $bx_{2i}aax_{2i+1}b$, they maintain a constant distance between them equal to $2n + 3 - i$; when the left head scans the b before x_{2i} then the right head scans the symbol from position i from the end of x_{2i+1}; when the right head scans the first b after x_{2i+1} then the left head scans the symbol from position i of x_{2i}. Note that using more finite control it is possible to check the equality of many pairs of symbols during one sweeping of the substring $bx_{2i}aax_{2i+1}b$. The front head used in this check is the same as the head from behind used in the previous check. After sweeping $x_i, i = 1, \ldots, 2n+1$ the two heads are available for the next check.

- k one-way heads can check whether $m = n^k$; each head i counts the number of blocks x_j scanned by head $i - 1$, for $i = 2, \ldots, k$. The first head just scans the symbols of each block x_j. Two heads become available from the check described above (in fact after sweeping $x_i, i = 1, \cdots, 2n + 1$, but this is an insignificant inconvenience) so we need only $k - 2$ extra heads for this check.

If during one of the above checks B discovers that u is not of the right form (i.e., $u \notin f(L)$), then B rejects the string. While checking whether u is of the right form (i.e., whether $u = f(x)$ for some $x \in L$), with two one-way heads scanning u, B tries to simulate the computation of A on x. Also, the head from behind used for the last check can be used as one head simulating one of the two heads of A. In fact, the moves of this head trigger the moves of the heads performing the check of the input string, so we have to make sure that, in case of acceptance, this head will move up to the end of the input string. To finish the proof, one other detail remains to be clarified: When the two one-way heads of B "simulate" the sweeping heads of A, the automaton B does not know yet whether $u = f(x)$. In fact if $u \neq f(x)$, B does not even simulate A on x. However, in this case B is going to reject u since it knows whether both its heads, used to simulate the heads of A, stop moving before reaching the end of u. In this case, if B has reached a state corresponding to an accepting state of A and if the check whether $u = f(x)$ is not yet done, B continues the check, by moving its heads up to the end of the input string and accepting u only if $u = f(x)$. At this moment it becomes obvious why in Theorem 4.1 we use reductions to languages recognized by two-way two-head finite automata having at least one moving head at each computation step. □

Using this proposition we obtain the next relations.

THEOREM 4.3. $\mathbf{BPL}_{poly} \leq_{log} \mathbf{1BPFA}(4)$, $\mathbf{RL}_{poly} \leq_{log} \mathbf{1RPFA}(4)$.

Proof. We prove only the first relation, since the proof of the second relation is similar. It is enough to show that $\mathbf{BPL}_{poly}$ is logspace reducible to languages recognized by (non-sensing, sweeping) 2bpfa(2) that run in *subquadratic* time and move at least one head at each computation step. Using the proof of Proposition 4.2 with $k = 2$ it follows our claim.

For any language $L \in \mathbf{BPL}_{poly}$ over the alphabet Σ, we consider a 2bpfa(h) A that recognizes any length-n input string from L in time less than n^p, for some constants p and h. As in the proof of Theorem 4.1, we choose a transformation $g_{\Sigma,l}$, where l is much larger than $\max(p, h)$. We denote by B the corresponding (sweeping, non-sensing) 2bpfa(2) that recognizes $g_{\Sigma,l}(L)$. The computation time of B on a length-m input string w is $O(m)$ if $w \neq g_{\Sigma,l}(x)$ for any $x \in \Sigma^*$, and is $O(n^p m)$ if $w = g_{\Sigma,l}(x)$ for some string x of length n. In the latter case $|w| = n^l$ and n^l is much larger than n^p and thus it follows that the computation time of B is $o(m^2)$. ∎

THEOREM 4.4. $\mathbf{PL} \leq_{log} \mathbf{1PFA}(2)$, $\mathbf{PL} \leq_{log} \mathbf{1PCM}(1)$.

Proof. The proof follows from Lemmas 4.7 and 4.8 stated after the following definitions. ∎

We define two languages that are logspace complete for **PL** and can be recognized by "simple" probabilistic finite automata, i.e., by one-way two-head and respectively one-way one-head one-counter unbounded-error probabilistic finite automata. These languages may be seen as probabilistic versions of the Savitch's Maze threading problem [38] and of the languages L_P and L_Q of Sudborough [44]. There are several interesting aspects involved in the design of these languages since they have to be quite complex in order to be complete for **PL**, but they also have to be relatively simple in order to be recognized by simple one-way devices.

DEFINITION 4.5. *PMT_1 is the language over the alphabet $\{[,],a,\#\}$ containing the strings of the following form, denoted by (*):*

$$[a^{p_1}\#a^{m_1(1)}\#a^{m_2(1)}]\cdots[a^{p_t}\#a^{m_1(t)}\#a^{m_2(t)}]a^{v_1}\#\ldots\#a^{v_{h_1}}\#\#a^{r_1}\#\ldots\#a^{r_{h_2}}\#$$

with the properties

$$\sum_{(q,i_1,\ldots,i_q)\in A_{cc}} 1/2^{q-1} > \sum_{(r,i_1,\ldots,i_r)\in R_{ej}} 1/2^{r-1},$$

where $m_i : \mathbf{N} \to \mathbf{N}, i \in \{1,2\}$, and A_{cc} and R_{ej} are sets satisfying the relations

$$(q,i_1,\ldots,i_q) \in A_{cc} \Leftrightarrow \begin{cases} q,i_1,\ldots,i_q \in \mathbf{N}, \\ i_1 = 1, i_2 > 1, \text{and } i_k > i_{k-2} \text{ for } 3 \leq k \leq q, \\ \forall k \in \{2,\ldots,q\}, \exists l \in \{1,2\} \text{ such that } p_{i_k} = m_l(i_{k-1}), \\ \exists l \in \{1,2\} \text{ such that } m_l(i_q) \in \{v_1,\ldots,v_{h_1}\}, \end{cases}$$

$$(r,i_1,\ldots,i_r) \in R_{ej} \Leftrightarrow \begin{cases} r,i_1,\ldots,i_r \in \mathbf{N}, \\ i_1 = 1, i_2 > 1, \text{and } i_k > i_{k-2} \text{ for } 3 \leq k \leq r, \\ \forall k \in \{2,\ldots,r\}, \exists l \in \{1,2\} \text{ such that } p_{i_k} = m_l(i_{k-1}), \\ \exists l \in \{1,2\} \text{ such that } m_l(i_r) \in \{r_1,\ldots,r_{h_2}\}. \end{cases}$$

For each string of the form (*), p_1 is the initial index, $v_i, i \in \{1,\ldots,h_1\}$ and $r_i, i \in \{1,\ldots,h_2\}$ are the accepting and respectively rejecting indices, and A_{cc} and R_{ej} are all the "almost one-way" paths that connect the index p_1 to the accepting and respectively rejecting indices. A string belongs to PMT_1 if the weight of all the "almost one-way" paths connecting the initial index to accepting indices is larger than the weight of all the almost one-way paths connecting the initial index to rejecting indices. ∎

DEFINITION 4.6. *PMT_2 is the language over the alphabet $\{[,],a,\#\}$ defined in a similar way as PMT_1 with the following difference: In the definitions of A_{cc} and R_{ej} the conditions "$i_k > i_{k-2}$ for $3 \leq k \leq q$" and "$i_k > i_{k-2}$ for $3 \leq k \leq r$" are replaced by "$i_k > i_{k-1}$ for $2 \leq k \leq q$"and "$i_k > i_{k-1}$ for $2 \leq k \leq r$," respectively. In this case, A_{cc} and R_{ej} contain all the "one-way" paths that connect the index p_1 to the accepting and respectively rejecting indices.* ∎

To summarize, PMT_1 and PMT_2 are the encodings of the languages recognized by probabilistic devices whose computation successive steps can be encoded in "almost" increasing order and increasing order, respectively. Also note that PMT_1 and PMT_2 are incomparable.

LEMMA 4.7. **PL** $\leq_{log}$ *PMT_1,* **PL** $\leq_{log}$ *PMT_2.*

Proof. Using Proposition 4.2 and some standard techniques from probabilistic computation [36], it can be easily shown that each language in **PL** is logspace reducible to a language recognized by a one-way multihead probabilistic finite automaton which reaches halting (i.e., accepting or rejecting) configurations with probability 1, moves at least one head at every computation step, and from each nonhalting (i.e., nonaccepting or nonrejecting) configuration goes with probability 1/2 to the next two configurations.

Let Σ be a finite alphabet and $L \in \Sigma^*$ be an arbitrary language recognized by such a multihead probabilistic finite automaton A. We show how to reduce (in logspace) L to PMT_1 and PMT_2.

For each input string x, the configurations of a one-way multihead finite automaton A can be indexed in lexicographically increasing order of input heads positions. (Note that during the computation of A on x, the sequence of indices assigned to consecutive configurations is strictly increasing.) For each configuration we build a "configuration transition block" containing the configuration and its successors. The logspace transformation f we are looking for maps each string x to the string consisting of the sequence of "configuration transition blocks" (corresponding to all the configurations of A on x, enumerated in increasing order of their indices) concatenated with the sequences of accepting and rejecting configurations of A on x. It can be checked that $f(x) \in PMT_1$ iff $f(x) \in PMT_2$ iff A accepts x.

More exactly, the transformation $f_A : \Sigma^* \longrightarrow \{[,], a, \#\}^*$ is

$$f_A(x) = [a^{p_1}\#a^{m_1(1)}\#a^{m_2(1)}]\cdots[a^{p_t}\#a^{m_1(t)}\#a^{m_2(t)}]a^{v_1}\#\ldots\#a^{v_{h_1}}\#\#a^{r_1}\#\ldots\#a^{r_{h_2}}\#\#,$$

where

- t is the number of nonhalting configurations of A, h_1 and h_2 are the number of accepting and, respectively, rejecting configurations of A;
- p_1 is the index of the initial configuration of A;
- $\forall\ i \in [1, t]$, from the configuration with index p_i, A moves with probability 1/2 in one of the configurations with indices $m_1(i)$ and $m_1(i)$;
- if $i < j$ then $p_i < p_j$;
- $v_i, i = 1, \ldots, h_1$ and $r_i, i = 1, \ldots, h_2$ are the indices assigned to the accepting and rejecting configurations, respectively.

The transformation f_A can be easily obtained in logarithmic space by writing all the configurations of A (enumerated in increasing order of the input heads positions) followed by their successors. From the fact that during the computation of A on any input x the sequence of indices assigned to consecutive configurations is strictly increasing, it follows that in $f_A(L)$, all the "paths" connecting the initial index to the accepting and rejecting indices are one-way. Additionally, using the fact that the computation of A on x stops with probability 1, it follows that $x \in L \Leftrightarrow$ "the probability that A accepts x is larger than the probability that A rejects x" $\Leftrightarrow f_A(x) \in PMT_1 \Leftrightarrow f_A(x) \in PMT_2$. □

LEMMA 4.8. *$PMT_1 \in$* **1PFA**(2)*, $PMT_2 \in$* **1PCM**(1)*.*

Proof. We describe an 1pfa(2) B that recognizes PMT_1. In parallel to its main computation (described next), B checks whether the input string u is of the form (*)

from the definition of PMT_1. If not, B rejects u. Consequently, in what follows we suppose that u is of the form (*).

By block-i we mean $[a^{p_i}\#a^{m_1(i)}\#a^{m_2(i)}]$ if $i \leq t$, or $a^{v(i-t)}$ if $t < i \leq t + h_1$, or $a^{r(i-t-h_1)}$ if $t + h_1 < i \leq t + h_1 + h_2$.

Suppose that B has one head ($H1$) on the substring $a^{m_l(i)}$ (inside block $i = [a^{p_i}\#a^{m_1(i)}\#a^{m_2(i)}]$) which is the "current" index, and the second head ($H2$) at the end of some block j. B "probabilistically guesses" a block $q > j$ moving head H_2 right and tossing a fair coin for each encountered block. q is the first block for which the outcome of the coin toss is "tails". If H_2 reaches the right end of u without "guessing" any block, then B accepts and rejects with the same probability 1/2. If H_2 guesses a block q, then B deterministically compares a^{p_q} with $a^{m_l(i)}$ moving both heads right over these substrings.

If $p_q \neq m_l(i)$ (i.e., the probabilistic guess is wrong), then B accepts and rejects with probability 1/2.

If $p_q = m_l(i)$ and $q \leq t$ (i.e., B has found the block describing the transitions from the "current index" $m_l(i)$), then B tosses a coin to select the next index. If the outcome is tails, it moves H_2 at the beginning of $a^{m_1(q)}$. Otherwise, it moves H_2 at the beginning of $a^{m_2(q)}$. Next, it continues the operation with H_1 and H_2 interchanged.

If $p_q = m_l(i)$ and $t < q \leq t + h_1$ (or $t + h_1 < q \leq t + h_1 + h_2$) (i.e., B has found an accepting (or rejecting) path), then B performs a procedure that "equalizes" the probability modifications produced by "probabilistic guesses." It keeps both heads moving to the right end of u, one head at a time, and it tosses a fair coin each time when a head encounters a block. If the outcome in all these tosses is all tails, then B accepts (rejects, respectively) u. If not, B accepts and rejects with the same probability 1/2. In this way, B makes sure that the probabilities of all accepting and rejecting almost one-way paths in PMT are multiplied by the same number, independent of the length and the structure of the path.

Using a similar technique, it follows that each language $L \in PMT_2$ can be recognized by a one-way one-counter probabilistic finite automaton B. In this case, the second head is replaced by a counter as follows: B stores into the counter the "current index" $m_l(i)$ and moves its head forward to probabilistically guess the block q describing the transitions from $m_l(i)$. For comparing $m_l(i)$ with p_q, B decrements the counter. If $p_q = m_l(i)$ (so B guessed the right block), then B selects the next index and stores its value into the counter. □

In the settings of **AM**-games and **UAM**-games we have the reductions as follows.

THEOREM 4.9.
$\mathbf{AML}_{poly} \leq_{log} \mathbf{AM}(1pfa(4))$, $\mathbf{AML} \leq_{log} \mathbf{AM}(2pfa(2))$,
$\mathbf{AML} \leq_{log} \mathbf{UAM}(1pfa(2))$, $\mathbf{AML} \leq_{log} \mathbf{UAM}(1pcm(1))$.

Proof (sketch). The proof for the first reduction is similar to that of Theorem 4.3. The second reduction is a corollary of Theorem 4.1. The last two reductions follow from $\mathbf{AML} = \mathbf{P} = \mathbf{UAML}_{poly}$ [3], $\mathbf{UAML}_{poly} \leq_{log} \mathbf{UAM}(1pfa(4))$ (similar to the proof of Theorem 4.3), $\mathbf{UAM}(1pfa(4)) \leq_{log} \mathbf{UAM}(1pfa(2))$, and $\mathbf{UAM}(1pfa(4)) \leq_{log} \mathbf{UAM}(1pcm(1))$. To prove these last two claims, we design probabilistic-plus-nondeterministic variants of Savitch's maze-threading problem ($PNMT_1$, $PNMT_2$) that are logspace complete for **P** and can be solved by **UAM**-games with simple probabilistic finite-state verifiers. The claims follow from Lemmas 4.12 and 4.13. □

DEFINITION 4.10. *$PNMT_1$ is a language over the alphabet $\{[,], a, \#\}$ that has a structure similar to PMT_1 but the "transition" blocks have the form* $[a^{p_i}\#a^k\#a^{m_1(i)}$

$\#a^{m_2(i)}]$, where $k = 1$ if p_i is a "nondeterministic" index and $k = 2$ if p_i is a "probabilistic" index. A nondeterministic strategy for such a string is a function that, for each nondeterministic index, selects its successor from the two indices from its transition block. For each nondeterministic strategy K we define the sets $A_{cc}(K)$ and $R_{ej}(K)$ that contain all almost one-way paths that, according to the strategy, connect the (initial) index p_1 to the accepting and rejecting indices, respectively. Note that, given a strategy K, $A_{cc}(K)$ and $R_{ej}(K)$ are identical to A_{cc} and, respectively, R_{ej} from the definition of PMT_1. A string is in $PNMT_1$ if there is a strategy K such that

$$\sum_{(q,i_1,\ldots,i_q)\in A_{cc}(K)} 1/2^{q-1} > \sum_{(r,i_1,\ldots,i_r)\in R_{ej}(K)} 1/2^{r-1}. \qquad \square$$

DEFINITION 4.11. *$PNMT_2$ is a language over the alphabet $\{[,],a,\#\}$ that has a structure similar to $PNMT_1$, but for each nondeterministic strategy K the sets $A_{cc}(K)$ and $R_{ej}(K)$ contain all one-way paths connecting the initial index p_1 to the accepting or rejecting indices. For a fixed K, $A_{cc}(K)$ and $R_{ej}(K)$ are similar to the sets A_{cc} and R_{ej} used in the definition of PMT_2.* $\square$

LEMMA 4.12. $\mathbf{UAM}(1pfa(4)) \leq_{log} PNMT_1$, $\mathbf{UAM}(1pfa(4)) \leq_{log} PNMT_2$.

Proof. The proof is similar to the proof of the probabilistic case (Lemma 4.7). $\square$

LEMMA 4.13. $PNMT_1 \in \mathbf{UAM}(1pfa(2))$, $PNMT_2 \in \mathbf{UAM}(1pcm(1))$.

Proof. The proof is similar to the proof of the probabilistic case (Lemma 4.8). $\square$

5. Discussion. In this paper, we obtain results for logspace Turing machines and multihead finite automata with probabilistic and both probabilistic and nondeterministic states that parallel well-known relations proven for their nondeterministic counterparts [16, 44, 30, 31].

Each logspace complexity class is decomposed into a hierarchy based on corresponding two-way multihead finite automata. To obtain the properness of several of these head hierarchies, in section 3 we adapt translational methods to the setting of probabilistic-plus-nondeterministic computation. However, we could not prove the corresponding separation for **AM**-games with two-way multihead probabilistic finite-state verifiers that run in polynomial time. We leave it as an open problem. The main difficulty is in the fact that the class $\mathbf{AML}_{poly}$ is not known to be equal to **P** and, consequently, it does not seem large enough to solve linear programming problems. (We used this property of **P** when we proved the "coarse separation" for **AM**-games.)

In section 4 we obtain several reductions of logspace complexity classes to the second or lower level of their corresponding hierarchies. The most interesting ones are the reductions of **PL** (and **P**) to languages recognized by one-way non-sensing two-head or one-way one-head one-counter finite automata with probabilistic (probabilistic and nondeterministic) states. We define probabilistic (probabilistic-plus-nondeterministic) variants of the Savitch's maze threading problem that parallel the languages L_Q and L_P of Sudborough. Part of our proofs make use of Sudborough's techniques to prove the corresponding results for nondeterministic automata. However our proofs are much more complex for the following reason: Whereas in the setting of nondeterministic computation the number of additional guesses required during a simulation does not matter, in the probabilistic setting we can use only probabilistic guesses that unbalance the acceptance probability of the simulating machine; consequently we need additional resources and additional ideas to correct this "unbalance." Using a simple technique, we show that the one-way automata that recognize our probabilistic versions of the maze threading problem are powerful enough to correct the unbalance. These reductions give us characterizations for **PL** (and **P**) as the classes of languages logspace reducible to languages recognized by one-way non-sensing two-head or one-way one-head one-counter finite automata with probabilistic (probabilistic and nondeterministic) states. In the cases of polynomial-time logspace bounded-error probabilistic (probabilistic-plus-nondeterministic) computation we have obtained only reductions to languages recognized by one-way non-sensing four-head finite automata. Our technique to correct the unbalance successfully applied for unbounded-error computation does not seem to work anymore since it does not preserve the error-bounds. We ask whether it is possible to do better in these cases.

6. Appendix.

Proof of Theorem 3.1. We show only $\bigcup_{k=1}^{\infty} \mathbf{UAM}(2pfa(k)) = \mathbf{UAML}$. The other proofs are similar.

($\subset$). Straightforwardly, the multihead automaton's head positions may be stored in $O(\log n)$ work space.

($\supset$). Let B be a Turing machine with probabilistic and nondeterministic states with work space less than $c(\lfloor \log n \rfloor - 1)$, $c \in \mathbf{N}$, and let x be an input string of length n. Without loss of generality, we consider that B writes only 0's and 1's. We show how to simulate B using a multihead finite automaton with probabilistic and nondeterministic states (called A). We considered the work tape of B parsed into c segments of $\lfloor \log n \rfloor - 1$ bits each. A is designed with $c + 3$ heads scanning the input tape, of which one head (we call it the "input head") simulates the input head of B, $c - 1$ heads encode the contents of the segments not currently scanned by B's work head, 2 heads encode the right-of-head and left-of-head contents of the segment currently scanned by B's work head (we call U and V the numbers encoded by these two heads), and one auxiliary head implements some operations required by the simulation. In the encoding of any segment (or of the left or right side of the currently scanned segment) we use the next conventions: the closest bit to the current position of the work head of B is the least significant bit; the most significant bit of a tape segment of B is always 1 (so the segments are "ended" by an imaginary bit); the left and right side of the currently scanned segment do not contain the bit scanned by the work head of B; the value of the scanned bit is stored in the finite control. Using this encoding, the event when the work head of B moves from one segment to another coincides with the moment when U or V get value 1. A move of B's input head is simulated by a move of the input head of A; a move of the working head of B or a modification of the work tape content of B is simulated by modifications of the types $U \to \lfloor U/2 \rfloor$, $U \to 2U$ and $U \to 2U + 1$ (and similar modifications for V). To implement these operations we use the auxiliary head. The overall simulation is as follows. Corresponding to each of the nondeterministic (probabilistic) transitions of B, A nondeterministically (probabilistically) chooses the transition to simulate (with the same probability as B does) and then deterministically executes the corresponding sequence of operations. That sequence of operations is finite, so it can be stored in a finite control. $\square$

7. Acknowledgments. I am very grateful to Joel Seiferas for reading an earlier version of this paper and making useful comments. I also want to thank the anonymous referee for his remarkable suggestions concerning the reorganization of this article.

REFERENCES

[1] L. Babai and S. Moran, *Arthur-Merlin games: A randomized proof system and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[2] A. Borodin, S. Cook, and N. Pippenger, *Parallel computation for well-endowed rings and space-bounded probabilistic machines*, Inform. and Control, 48 (1983), pp. 113–136.

[3] A. Condon, *Computational Models of Games*, MIT Press, Cambridge, MA, 1989.

[4] A. Condon, *The complexity of space bounded interactive proof systems*, in Complexity Theory: Current Research, K. A. Spies, S. Homer, and U. Schoning, eds., Cambridge University Press, Cambridge, UK, 1993, pp. 147–189.

[5] A. Condon and R. Ladner, *Probabilistic game automata*, J. Comput. System Sci., 36 (1988), pp. 452–489.

[6] A. Condon, L. Hellerstein, S. Pottle, and A. Wigderson, *On the Power of Finite Automata with both Nondeterministic and Probabilistic States*, Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, Montreal, 1994, pp. 676–685.

[7] E. W. Dijkstra, *Making a fair roulette from a possibly biased coin*, Inform. Process. Lett., 36 (1990), pp. 193–193.

[8] C. Dwork and L. Stockmeyer, *Finite state verifiers* I: *The power of interaction*, J. ACM, 39 (1992), pp. 800–828.

[9] R. Freivalds, *Probabilistic two-way machines*, in Proceedings, International Symposium on Math. Foundations of Computer Science 1981, Lecture Notes in Comput. Sci., 118, Springer-Verlag, Berlin, New York, 1981, pp. 33–45.

[10] R. Freivalds, *Complexity of probabilistic versus deterministic automata*, Baltic Computer Science, Selected Papers, Lecture Notes in Comput. Sci. 502, Springer-Verlag, Berlin, New York, 1991, pp. 565–613.

[11] Z. Galil, *Two-way deterministic pushdown automata and some open problems in the theory of computation*, Proceedings of the Fifteenth Annual IEEE Symposium on Switching and Automata Theory, 1974, pp. 170–177.

[12] Z. Galil, *Some open problems in the theory of computation as questions about two-way deterministic pushdown automaton languages*, Math. Systems Theory, 10 (1977), pp. 211–228.

[13] J. Gill, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.

[14] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof systems*, SIAM J. Comput., 18 (1989), pp. 186–208.

[15] S. Goldwasser and M. Sipser, *Public coins vs. private coins in interactive proof systems*, in Randomness and Computation, Vol. 5 of Advances in Computing Research, JAI Press, Greenwich, CT, 1989, pp. 73–90.

[16] J. Hartmanis, *On non-determinancy in simple computing devices*, Acta Inform., 1 (1972), pp. 336–344.

[17] F. C. Hennie and R. E. Stearns, *Two-tape simulation of multitape Turing machines*, J. ACM, 13 (1966), pp. 533–546.

[18] O. H. Ibarra, *On two-way multihead automata*, J. Comput. System Sci., 7 (1973), pp. 28–36.

[19] N. Immerman, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17 (1988), pp. 935–938.

[20] H. Jung, *On probabilistic tape complexity and fast circuits for matrix inversion problems*, in Proceedings, Eleventh International Colloquium on Automata, Languages and Programming, ICALP 1984, Lecture Notes in Comput. Sci. 172, Springer-Verlag, Berlin, New York, 1989, pp. 281–291.

[21] H. Jung, *On probabilistic time and space*, in Proceedings, Twelfth International Colloquium on Automata, Languages and Programming, ICALP 1985, Lecture Notes in Comput. Sci. 194, Springer-Verlag, Berlin, New York, 1985, pp. 310–317.

[22] J. Kaneps, *Stochasticity of the languages recognizable by two-way finite probabilistic automata*, Diskret. Mat., 1 (1989), pp. 63–77 (in Russian).

[23] L. G. Khachiyan, *A polynomial algorithm for linear programming*, Soviet Math Dokl., 20 (1979), pp. 191–194.

[24] K. N. King, *Alternating multihead finite automata*, Theoret. Comput. Sci., 61 (1988), pp. 149–174.

[25] Yu. I. Kuklin, *Two-way probabilistic automata*, Avtomat. i Vycisl. Tekhn., 5 (1973), pp. 35–36 (in Russian).

[26] P. M. Lewis, II, R. Stearn, and J. Hartmanis, *Memory bounds for recognition of context-free and context-sensitive languages*, IEEE Conference Record on Switching Circuit Theory and Logical Design, 1965, pp. 191–202.

[27] I. I. Macarie, *Space-efficient deterministic simulations of probabilistic automata*, SIAM J. Comput., 27 (1998), pp. 448–465.
[28] I. I. Macarie, *Decreasing the bandwidth of a configuration transition matrix*, Inform. Process. Lett., 53 (1995), pp. 315–320.
[29] I. I. Macarie, *Multihead two-way probabilistic finite automata*, Theory Comput. Syst., 30 (1997), pp. 91–109.
[30] B. Monien, *Transformational methods and their application to complexity problems*, Acta Inform., 6 (1976), pp. 95–108.
[31] B. Monien, *Two-way Multihead Automata over a one-letter alphabet*, RAIRO Informa. Theor., 14 (1980), pp. 67–82.
[32] John von Neumann, *Various techniques used in connection with random digits*, in John von Neumann, Collected Works, Vol. V, Pergamon, Oxford, UK, 1963, pp. 768–770.
[33] C. H. Papadimitriou, *Games against Nature*, J. Comput. Syst. Sci., 31 (1985), pp. 288–301.
[34] S. Ruby and P. C. Fisher, *Translational methods and computational complexity*, Conference Record IEEE Symposium on Switching Circuit Theory and Logical Design, 1965, pp. 173–178.
[35] B. Ravikumar, *Some Observations on 2-way Probabilistic Finite Automata*, Tech. report TR92-208, Department of Computer Science and Statistics, University of Rhode Island, Kingston, RI, 1992.
[36] W. Ruzzo, J. Simon, and M. Tompa, *Space-bounded hierarchies and probabilistic computation*, J. Comput. Syst. Sci., 28 (1984), pp. 216–230.
[37] E. Santos, *Computability by probabilistic Turing machines*, Trans. Amer. Math. Soc., 159 (1971), pp. 165–184.
[38] W. J. Savitch, *Relationships between nondeterministic and deterministic tape complexity*, J. Comput. Syst. Sci., 4 (1970), pp. 177–192.
[39] J. I. Seiferas, *Relating refined space complexity classes*, J. Comput. Syst. Sci., 14 (1977), pp. 100–129.
[40] J. I. Seiferas, *Techniques for separating space complexity classes*, J. Comput. Syst. Sci., 14 (1977), pp. 73–99.
[41] J. I. Seiferas, M. J. Fischer, and A. R. Meyer, *Separating nondeterministic time complexity classes*, J. ACM, 25 (1978), pp. 146–167.
[42] J. Simon, *On the difference between one and many*, in Proceedings, Fourth International Colloquium on Automata, Languages and Programming, ICALP 1977, Lecture Notes in Comput. Sci., 52, pp. 480–491.
[43] J. Simon, *Space-bounded probabilistic Turing machine complexity classes are closed under complement*, 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 158–167.
[44] I. H. Sudborough, *On tape-bounded complexity classes and multihead finite automata*, J. Comput. Syst. Sci., 10 (1975), pp. 62–76.
[45] R. Szelepcsenyi, *The method of forced enumeration for nondeterministic automata*, Acta Informatica, 26 (1988), pp. 279–284.
[46] A. C. Yao and R. L. Rivest, *$K+1$ heads are better than K*, J. ACM, 25 (1978), pp. 337–340.

SIAM J. COMPUT.
Vol. 29, No. 3, pp. 1008-1029

APPROXIMATION ALGORITHMS FOR THE ORTHOGONAL Z-ORIENTED THREE-DIMENSIONAL PACKING PROBLEM*

F. K. MIYAZAWA† AND Y. WAKABAYASHI‡

Abstract. We present approximation algorithms for the *orthogonal z-oriented three-dimensional packing problem* (TPP^z) and analyze their asymptotic performance bound. This problem consists in packing a list of rectangular boxes $L = (b_1, b_2, \ldots, b_n)$ into a rectangular box $B = (l, w, \infty)$, orthogonally and oriented in the z-axis, in such a way that the height of the packing is minimized. We say that a packing is oriented in the z-axis when the boxes in L are allowed to be rotated (by ninety degrees) around the z-axis. This problem has some nice applications but has been less investigated than the well-known variant of it—denoted by TPP (three-dimensional orthogonal packing problem)—in which rotations of the boxes are not allowed. The problem TPP can be reduced to TPP^z. Given an algorithm for TPP^z, we can obtain an algorithm for TPP with the same asymptotic bound. We present an algorithm for TPP^z, called R, and three other algorithms, called LS, BS, and SS, for special cases of this problem in which the instances are more restricted. The algorithm LS is for the case in which all boxes in L have square bottoms; BS is for the case in which the box B has a square bottom, and SS is for the case in which the box B and all boxes in L have square bottoms. For an algorithm $\mathcal{A}$, we denote by $r(\mathcal{A})$ the asymptotic performance bound of $\mathcal{A}$. We show that $2.5 \le r(R) < 2.67$, $2.5 \le r(LS) \le 2.528$, $2.5 \le r(BS) \le 2.543$, and $2.333 \le r(SS) \le 2.361$. The algorithms presented here have the same complexity $\mathcal{O}(n \log n)$ as the other known algorithms for these problems, but they have better asymptotic performance bounds.

Key words. approximation algorithms, three-dimensional packing, asymptotic performance bound

AMS subject classifications. 68Q25, 52C17

PII. S009753979631391X

1. Introduction.

We present approximation algorithms for the orthogonal z-oriented three-dimensional packing problem and show results concerning their asymptotic performance bound. All algorithms described here have time complexity $\mathcal{O}(n \log n)$, where n is the number of boxes in the input list.

Let $L = (b_1, b_2, \ldots, b_n)$ be a list of rectangular boxes $b_i = (x_i, y_i, z_i)$, where x_i, y_i, and z_i is the *length*, *width*, and *height* of b_i, respectively. The *orthogonal z-oriented three-dimensional packing problem* (TPP^z), can be defined as follows. Given a box $B = (l, w, \infty)$ and a list of boxes $L = (b_1, b_2, \ldots, b_n)$, find an orthogonal z-oriented packing of L into B that minimizes the total height. In the next section we define the concept of *orthogonal z-oriented packing.* For the moment, let us informally say that it is an orthogonal packing in which the boxes may be rotated around the z-axis (but may not be turned down).

A variant of TPP^z, in which the boxes may not be rotated around the z-axis, has been more investigated and is known as the *three-dimensional orthogonal packing*

*Received by the editors December 19, 1996; accepted for publication (in revised form) August 24, 1998; published electronically December 21, 1999. The research for this work was part of the Project ProComb (CNPq, Proc. 680065/94-6) and PCE (CNPq, Proc. 304527/89-0). It was also supported by FAPESP (Proc. 96/4505–2) and ProNEx (MCT/FINEP Proc. 107/97).
http://www.siam.org/journals/sicomp/29-3/31391.html

†Instituto de Computação, Universidade Estadual de Campinas, Caixa Postal 6176, 13083-970, Campinas, SP, Brazil (fkm@dcc.unicamp.br). The work of this author was partially supported by CNPq individual research grant Proc. 300301/98-7.

‡ Instituto de Matemática e Estatística, Universidade de São Paulo, Rua do Matão, 1010, Cidade Universitária, 05508–900, São Paulo, SP, Brazil (yw@ime.usp.br). The work of this author was partially supported by CNPq individual research grant Proc. 304527/89-0.

problem [3, 5, 6]. We denote it by TPP. Since all packings to be mentioned here are orthogonal we omit this term. Here we show that TPP can be reduced to TPP^z. Since the unidimensional packing problem [2] can be reduced to TPP, it follows that both TPP and TPP^z are $\mathcal{NP}$-hard.

If $\mathcal{A}$ is an algorithm for TPP^z or TPP and L is a list of boxes, then $\mathcal{A}(L)$ denotes the height of the packing generated by algorithm $\mathcal{A}$ when applied to a list L; and $\mathrm{OPT}(L)$ denotes the height of an optimal packing of L. We say that α is *an asymptotic performance bound* of an algorithm $\mathcal{A}$ if there exists a constant β such that for all lists L, in which all boxes have a height bounded by a constant Z, the following holds: $\mathcal{A}(L) \leq \alpha \cdot \mathrm{OPT}(L) + \beta \cdot Z$. Furthermore, if for any small ϵ and any large M, both positive, there is an instance L such that $\mathcal{A}(L) > (\alpha - \epsilon)\mathrm{OPT}(L)$ and $\mathrm{OPT}(L) > M$, then we say that α is *the asymptotic performance bound* of algorithm $\mathcal{A}$. We denote by $r(\mathcal{A})$ the asymptotic performance bound of $\mathcal{A}$.

In 1990, Li and Cheng [4] presented TPP^z as a model for a *job scheduling problem in partitionable mesh connected systems*. In this problem a set of jobs $J_1, J_2, \ldots, J_n$ is to be processed in a partitionable mesh connected system that consists of $l \times w$ processing elements connected as a rectangular mesh. Each job J_i is specified by a triplet $J_i = (x_i, y_i, t_i)$ indicating that a submesh of size either (x_i, y_i) or (y_i, x_i) is required by job J_i, and t_i is its processing time. The objective is to assign the jobs to the submeshes so as to minimize the total processing time. The algorithm for TPP^z described in [4] has asymptotic performance bound $4\frac{4}{7}$.

In [3] Li and Cheng describe several algorithms for TPP: for the general case, an algorithm whose asymptotic performance bound is 3.25, and for the special case in which all boxes have square bottom, an algorithm whose asymptotic performance bound is 2.6875. In 1992, these authors [5] also presented an on-line algorithm with an asymptotic performance bound that can be made as close to 2.89 as desired.

In [6] we present an algorithm for TPP whose asymptotic performance bound is less than 2.67. In this paper we describe an algorithm for TPP^z that has a similar asymptotic performance bound. We also describe an algorithm for the special case of TPP^z in which the box B has a square bottom and show that its asymptotic performance bound is less than 2.528. For the case in which all boxes of L have square bottoms, we present an algorithm with an asymptotic performance bound less than 2.543. Moreover, for the case in which all boxes have square bottoms, we present an algorithm whose asymptotic performance is less than 2.361. The algorithms we describe here for special instances of TPP^z are not straightforward simplifications of the algorithm for the general case. Each one resulted from a careful analysis of the instances under consideration.

There is a fundamental aspect in which the algorithms we have developed differ from those of Li and Cheng. Their strategy is to divide the input list into sublists and apply appropriate algorithms for each sublist, returning a packing that is a concatenation of these individual packings. The strategy we use also makes subdivisions (different ones) of the input list, but generates not only packings of each sublist but also those that are obtained by appropriate combinations of different sublists. In fact, we may say that the key idea behind our algorithms is to consider sublists which can be combined to generate better packings.

This paper is organized as follows. In section 2 we define some basic concepts, establish the notation, and discuss relations between TPP and TPP^z. In section 3 we describe the main algorithm (for TPP^z) and analyze its asymptotic performance bound. In each of the next three sections we describe an algorithm for a special instance of TPP^z and prove results on its asymptotic performance bound.

2. Notation and basic results. Given a list of boxes $L = (b_1, \dots, b_n)$ to be packed into a box $B = (l, w, \infty)$, we assume that each box b_i is of the form $b_i = (x_i, y_i, z_i)$, with $x_i \leq l$ and $y_i \leq w$ or $x_i \leq w$ and $y_i \leq l$ (that is, each box b_i can be packed into B in some orientation). We also assume throughout this paper that the list L consists of boxes with height bounded by a constant Z. In all algorithms mentioned here, unless otherwise stated, the input box B is assumed to be of the form $B = (l, w, \infty)$.

Given a triplet $t = (a, b, c)$, we also refer to each of its elements a, b, and c as $x(t)$, $y(t)$, and $z(t)$, respectively. For each box $b_i = (x_i, y_i, z_i)$, we denote by $\rho(b_i)$ the box consisting of the triplet (y_i, x_i, z_i) and we set $\Gamma(L) = \{(c_1, c_2, \dots, c_n) : c_i \in \{b_i, \rho(b_i)\}\}$. Given a real function $f : C \to \mathbb{R}$ and a subset $C' \subseteq C$, we denote by $f(C')$ the sum $\sum_{e \in C'} f(e)$.

Although a list is given as an ordered n-tuple of boxes, when the order of the boxes is irrelevant, the corresponding list may be viewed as a set.

Note that, by using a three-dimensional coordinate system, the box $B = (l, w, \infty)$ can be seen as the region $[0, l) \times [0, w) \times [0, \infty)$; and we may define a *$z$-oriented packing* $\mathcal{P}$ of a list of boxes L into B as a mapping $\mathcal{P} : L' = (b_1, \dots, b_n) \to [0, l) \times [0, w) \times [0, \infty)$, such that

$$L' \in \Gamma(L), \qquad \mathcal{P}^x(b_i) + x_i \leq l \text{ and } \mathcal{P}^y(b_i) + y_i \leq w \ ,$$

where $\mathcal{P}(b_i) = (\mathcal{P}^x(b_i), \mathcal{P}^y(b_i), \mathcal{P}^z(b_i))$, $i = 1, \dots, n$.

Furthermore, if $\mathcal{R}(b_i)$ is defined as

$$\mathcal{R}(b_i) = [\mathcal{P}^x(b_i), \mathcal{P}^x(b_i) + x_i) \times [\mathcal{P}^y(b_i), \mathcal{P}^y(b_i) + y_i) \times [\mathcal{P}^z(b_i), \mathcal{P}^z(b_i) + z_i),$$

then the following must hold:

$$\mathcal{R}(b_i) \cap \mathcal{R}(b_j) = \emptyset \quad \forall i, j,\ 1 \leq i \neq j \leq n \ .$$

If in the above definition we replace $L' \in \Gamma(L)$ by $L' = L$, then we have the concept of *oriented packing* (note that the condition $L' = L$ means that the boxes in L may not be rotated around the z-axis).

In what follows, we may use the term *packing* to refer to both the z-oriented and the oriented packing. To be precise, sometimes we should refer to a *z-oriented packing* (when some boxes are being rotated), but we simply say *packing* as this will be clear from the context. When this may cause confusion we specify which packing we are referring to.

Given a packing $\mathcal{P}$ of L, we denote by $H(\mathcal{P})$ the *height* of the packing $\mathcal{P}$, i.e., $H(\mathcal{P}) := \max\{\mathcal{P}^z(b) + z(b) :\ b \in L\}$.

If $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_v$ are packings of disjoint lists $L_1, L_2, \dots, L_v$, respectively, we define the *concatenation* of these packings as a packing $\mathcal{P} = \mathcal{P}_1 \| \mathcal{P}_2 \| \cdots \| \mathcal{P}_v$ of $L = L_1 \cup L_2 \cup \cdots \cup L_v$, where $\mathcal{P}(b) = (\mathcal{P}_i^x(b), \mathcal{P}_i^y(b), \sum_{j=1}^{i-1} H(\mathcal{P}_j) + \mathcal{P}_i^z(b))$, for all $b \in L_i$, $1 \leq i \leq v$. If each list $L_i = (b_1^i, b_2^i, \dots, b_{n_i}^i)$, $i = 1, \dots, v$, then the *concatenation* of these lists, denoted by $L_1 \| L_2 \| \cdots \| L_v$, is the list $(b_1^1, \dots, b_{n_1}^1, b_1^2, \dots, b_{n_2}^2, \dots, b_1^v, \dots, b_{n_v}^v)$.

The following notation is used to consider sublists of the list L.

- $\mathcal{C} := \{b_i = (x_i, y_i, z_i) : 0 \leq x_i \leq l, \quad 0 \leq y_i \leq w, \quad z_i > 0\}$;
- $\mathcal{C}[p'', p' \ ;\ q'', q'] := \{b_i = (x_i, y_i, z_i) :\ p'' \cdot l < x_i \leq p' \cdot l,\ q'' \cdot w < y_i \leq q' \cdot w\}$, for $0 \leq p'' < p' \leq 1, \quad 0 \leq q'' < q' \leq 1$;

- $\mathcal{Q}[p'', p' \ ; \ q'', q'] := \{b_i = (x_i, y_i, z_i) : \ b_i \in \mathcal{C}[p'', p' \ ; \ q'', q'] \ \text{ and } \ x_i = y_i\}$;
- $\mathcal{X} := \{b_i = (x_i, y_i, z_i) : \ y_i < x_i\}$, $\mathcal{Y} := \{b_i = (x_i, y_i, z_i) : \ y_i \geq x_i\}$;
- $\mathcal{C}_m := \mathcal{C}[0, \frac{1}{m} \ ; \ 0, \frac{1}{m}]$, $\quad \mathcal{Q}_m := \mathcal{Q}[0, \frac{1}{m} \ ; \ 0, \frac{1}{m}]$, for $m > 0$;
- $\mathcal{R}_1 := \mathcal{C}[0, \frac{1}{2} \ ; \ 0, \frac{1}{2}]$, $\mathcal{R}_2 := \mathcal{C}[0, \frac{1}{2} \ ; \ \frac{1}{2}, 1]$, $\mathcal{R}_3 := \mathcal{C}[\frac{1}{2}, 1 \ ; \ 0, \frac{1}{2}]$, $\mathcal{R}_4 := \mathcal{C}[\frac{1}{2}, 1 \ ; \ \frac{1}{2}, 1]$.

If $\mathcal{R}$ is a set of boxes, then we say that a box b is of *type* $\mathcal{R}$ if $b \in \mathcal{R}$ or $\rho(b) \in \mathcal{R}$.

We denote by $S(b)$ and $V(b)$ the *bottom area* (i.e., $S(b) := x(b)y(b)$) and the *volume* of the box b, respectively.

A *level* N in a packing $\mathcal{P}$ is a region $[0, l) \times [0, w) \times [Z_1, Z_2)$ in which there is a set L' of boxes such that for all $b \in L'$, $\mathcal{P}^z(b) = Z_1$ and $Z_2 - Z_1 = \max\{z(b) : \ b \in L'\}$. Sometimes we shall consider the level N as a packing of the list L'; we denote by $S(N)$ the sum $\sum_{b \in L'} S(b)$.

A *layer* (in the x-axis direction) in a level is a region $[0, l) \times [Y_1, Y_2) \times [Z_1, Z_2)$ in which there is a set L' of boxes such that, for all $b \in L'$, $\mathcal{P}^y(b) = Y_1$ and $\mathcal{P}^z(b) = Z_1$; and moreover, $Y_2 - Y_1 = \max\{y(b) : \ b \in L'\}$ and $Z_2 - Z_1 = \max\{z(b) : \ b \in L'\}$.

Relations between TPP and TPP$^{\boldsymbol{z}}$. One way to solve TPP^z is to adapt algorithms for TPP. A simple approach is to generate for each instance $L = (b_1, b_2, \ldots, b_n)$ a new instance $\phi(L) \in \Gamma(L)$, such that $\phi(L) = (d_1, d_2, \ldots, d_n)$, where

$$d_i = \begin{cases} b_i & \text{if } \ x_i \leq l \ \text{ and } \ y_i \leq w, \\ \rho(b_i) & \text{otherwise,} \end{cases}$$

and then apply an algorithm for TPP on the list $\phi(L)$.

For each algorithm $\mathcal{A}$ for TPP, let us denote by $\widehat{\mathcal{A}}$ the corresponding algorithm for TPP^z, as described above. That is, for every instance L of TPP^z, algorithm $\widehat{\mathcal{A}}$ applies algorithm $\mathcal{A}$ on the list $\phi(L)$. It is easy to see that the algorithm $\widehat{\mathcal{A}}$ may not preserve the asymptotic performance of the original algorithm $\mathcal{A}$.

The next result shows that there is no algorithm $\widehat{\mathcal{A}}$ for TPP^z, obtained from an algorithm $\mathcal{A}$ for TPP, as described previously, that has an asymptotic performance bound less than 3.

PROPOSITION 2.1. *If $\widehat{\mathcal{A}}$ is an algorithm for* TPP^z *obtained from an algorithm $\mathcal{A}$ for* TPP, *as described above, then the asymptotic performance bound of $\widehat{\mathcal{A}}$ is at least* 3.

Proof. Let $L = (b_1, b_2, \ldots, b_{3k})$ and $B = (3 + 3\epsilon, 2, \infty)$ be an instance of of TPP^z, where $b_1 = b_2 = \cdots = b_{3k} = (2, 1 + \epsilon, 1)$, k is a positive integer, and ϵ is a positive small number.

First, observe that it is possible to pack L in k levels, that is, $\mathrm{OPT}(L) \leq k$. For that, initially rotate each box in L and generate a packing putting three boxes per level. Now it suffices to note that, since $L = \phi(L)$, any algorithm $\mathcal{A}$ for TPP is such that $\widehat{\mathcal{A}}(L) \geq 3k$ (as the algorithm $\widehat{\mathcal{A}}$ packs only one box per level). ∎

Now suppose we have an algorithm $\mathcal{A}$ for TPP^z. There is a natural way to adapt it to an algorithm, say $\mathcal{A}'$, for TPP. The question is what can we say about the performance of $\mathcal{A}'$. The next result gives the answer.

THEOREM 2.2. *There is a polynomial reduction of* TPP *to* TPP^z *that preserves the approximability. Moreover, this reduction also preserves the additive constant β. That is, if $\mathcal{A}$ is a polynomial algorithm for* TPP^z, *such that $\mathcal{A}(L) \leq \alpha \cdot \mathrm{OPT}(L) + \beta \cdot Z$ then there exists a polynomial algorithm $\mathcal{A}'$ for* TPP *such that $\mathcal{A}'(L) \leq \alpha \cdot \mathrm{OPT}'(L) + \beta \cdot Z$, where* $\mathrm{OPT}'(L)$ *is the height of an optimum oriented packing of L.*

Proof. Let L, as described, and $B = (l, w, \infty)$ be an instance of TPP. Consider the following algorithm $\mathcal{A}'$. First scale B to $B' = (l', w', \infty)$ and L to L' in the same proportion in such a way that $\min\{x(b) : b \in L'\} > w$; then apply algorithm $\mathcal{A}$ to pack L' into the box B', obtaining a packing $\mathcal{P}'$. Finally, rescale $\mathcal{P}'$ back, obtaining a packing of the original list L (into B). It is clear that $\mathcal{A}'(L) \leq \alpha \cdot \mathrm{OPT}'(L) + \beta Z$. ∎

For all algorithms presented in the next sections, we consider, without loss of generality, that $L = \phi(L)$. That is, we may assume that the boxes in L need not be rotated to fit in the box B.

Before we present the algorithms for TPP^z, let us mention some algorithms used as subroutines and also the related results that are needed.

We denote by NFDH the next fit decreasing height algorithm for TPP, presented by Li and Cheng in [3]. For the description of this algorithm the reader may refer to [3] or [6]. This algorithm has two variants: NFDH^x and NFDH^y. The notation NFDH is used to refer to any of these variants.

Li and Cheng [3] proved the following result.

LEMMA 2.3. *If* $L \subset \mathcal{C}[\frac{1}{m+1}, \frac{1}{m} \;;\; 0, \frac{1}{m}]$, *then* $\mathrm{NFDH}^y(L) \leq (\frac{m+1}{m-1})\frac{V(L)}{l \cdot w} + Z$. *The same result also holds for the algorithm* NFDH^x *when applied to a list* $L \subset \mathcal{C}[0, \frac{1}{m} \;;\; \frac{1}{m+1}, \frac{1}{m}]$.

The following result is more general and gives as a corollary the result above [6].

LEMMA 2.4. *Let L be an instance of* TPP *and $\mathcal{P}$ be a packing of L consisting of levels $N_1, \ldots, N_v$ such that* $\min\{z(b) : b \in N_i\} \geq \max\{z(b) : b \in N_{i+1}\}$, *and* $S(N_i) \geq s \cdot l \cdot w$ *for a given constant $s > 0$, $i = 1, \ldots, v-1$. Then* $H(\mathcal{P}) \leq \frac{1}{s}\frac{V(L)}{l \cdot w} + Z$.

The constant s mentioned in the above lemma is called an *area guarantee* of the packing $\mathcal{P}$.

Li and Cheng presented in [4] an algorithm called LL for instances $L \subset \mathcal{C}_m$, $m \geq 3$. We write $\mathrm{LL}(L, m)$ to indicate that we are applying the algorithm LL to a list $L \subset \mathcal{C}_m$. They proved that the following result holds for this algorithm.

LEMMA 2.5. *Let* $L \subset \mathcal{C}[0, \frac{1}{m} \;;\; 0, \frac{1}{m}]$ *be an instance of* TPP *and $\mathcal{P}$ be a packing of L obtained by applying the algorithm* LL. *Then* $H(\mathcal{P}) \leq (\frac{m}{m-2})\frac{V(L)}{l \cdot w} + Z$.

We give an idea of the algorithm $\mathrm{LL}(L, m)$, as we need to refer to it in the proof of Lemma 2.6. Initially, it sorts the boxes in L in nonincreasing order of their height. Then it divides L into sublists $L_1, \ldots, L_v$, such that $L = L_1 \| L_2 \| \cdots \| L_v$, each sublist preserving the (nonincreasing) order of the boxes, and

$$\begin{aligned} &S(L_i) \leq \left[\left(\tfrac{m-2}{m}\right) + \left(\tfrac{1}{m}\right)^2\right] lw && \text{for } i = 1, \ldots, v\ , \\ &S(L_i) + S(first(L_{i+1})) > \left[\left(\tfrac{m-2}{m}\right) + \left(\tfrac{1}{m}\right)^2\right] lw && \text{for } i = 1, \ldots, v-1\ , \end{aligned}$$

where *first*(L') is the first box in L'. Then, the algorithm LL uses a two-dimensional packing algorithm to pack each list L_i in only one level, say, N_i. The final packing is the concatenation of each of these levels.

The next lemma is used to prove lower bounds for the asymptotic performance bound of some algorithms shown here.

LEMMA 2.6. *Let $\mathcal{A}$ be an algorithm for* TPP (TPP^z) *that partitions the input list L into two sublists $L_1 \subset \mathcal{R}_4$ and $L_2 \subset \mathcal{Q}_m$, $m \geq 3$, and generates a packing $\mathcal{P} = \mathcal{P}_1 || \mathcal{P}_2$, where $\mathcal{P}_1$ is any packing of L_1 and $\mathcal{P}_2$ is a packing of L_2 using the algorithm* LL. *Then the asymptotic performance bound $r(\mathcal{A})$ of $\mathcal{A}$ is such that* $r(\mathcal{A}) \geq \frac{7m-8}{4m-8}$.

Proof. Consider a box $B = (1, 1, \infty)$. Let L be a list of boxes, $L = L_1 \cup L_2$, with $L_1 \subset \mathcal{R}_4$ and $L_2 \subset \mathcal{Q}_m$, $m \geq 3$. Let $L_1 = (b'_1, \ldots, b'_{N'})$ and $L_2 = (b''_1, \ldots, b''_{M \cdot N''})$, where

$$b'_i = \left(\frac{1}{2}+\frac{1}{k}, \frac{1}{2}+\frac{1}{k}, 1\right) \quad \text{and} \quad b''_i = \begin{cases} \left(\frac{1}{m}, \frac{1}{m}, 1-(i-1)\xi\right) & \text{if } i \bmod M = 1, \\ \left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right) & \text{otherwise.} \end{cases}$$

Recall that the algorithm LL groups the first boxes with total bottom area no greater than $\left(\frac{m-2}{m}\right)+\left(\frac{1}{m}\right)^2$. This instance was chosen in such a way that, in the packing generated by the algorithm LL, each level has M boxes whose bottom area is $\left(\frac{m-2}{m}\right)+\left(\frac{1}{k}\right)^2$.

Note that the algorithm LL divides the list L_2 into N'' sublists, each sublist consisting of one box of the form $\left(\frac{1}{m}, \frac{1}{m}, 1-(i-1)\xi\right)$ and $M-1$ boxes of the form $\left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right)$.

The strategy is to take the instance $L = L_1 \cup L_2$ in such a way that the optimum packing consists of N' levels, each level containing one box of L_1 and the remaining space (in each level) almost filled with boxes of L_2. Taking N' and k as very large integers, with $N'' = \lceil \frac{3}{4} N' \frac{m}{m-2} \rceil$ and k a multiple of $2m$, we may choose M appropriately so that $r(\mathcal{A})$ can be made as close to $\frac{7m-8}{4m-8}$ as desired. $\square$

Another algorithm that plays an important role for the algorithms presented here is the algorithm COMBINE. This algorithm is a slightly modified version of the algorithm COLUMN presented in [6]. This algorithm generates a partial packing of a list L. The packing consists of several stacks of boxes, referred to as *columns*. Each column is built by putting one box on top of the other, and each column consists only of boxes of type either $\mathcal{T}^1$ or $\mathcal{T}^2$.

The algorithm COMBINE is called with the parameters $(L, \mathcal{T}^1, p^1, \mathcal{T}^2, p^2)$, where $p^1 = [p^1_1, p^1_2, \dots, p^1_{n_1}]$ consists of the positions in the bottom of box B where the columns of boxes of type $\mathcal{T}^1$ should start and $p^2 = [p^2_1, p^2_2, \dots, p^2_{n_2}]$ consists of the positions in the bottom of box B where the columns of boxes of type $\mathcal{T}^2$ should start. Each point $p^i_j = (x^i_j, y^i_j)$ represents the x-axis and the y-axis coordinates where the first box (if any) of each column of the respective type must be packed. Note that the z-axis coordinate need not be specified since it may always be assumed to be 0 (corresponding to the bottom of box B). Here we are assuming that the positions p^1, p^2 and the types $\mathcal{T}^1$, $\mathcal{T}^2$ are chosen in such a way that the defined packing can always be performed.

We call *height of a column* the sum of the height of all boxes in that column.

Initially, all $n_1 + n_2$ columns are empty, starting at the bottom of box B. At each iteration, the algorithm chooses a column with the smallest height, say a column given by the position p^i_j, and packs the next box b of type $\mathcal{T}^i$, updating the list L after each iteration. If there is no such box b, then the algorithm halts returning the partial packing $\mathcal{P}$ of L. We also say that $\mathcal{P}$ *combines* the lists of types $\mathcal{T}^1$ and $\mathcal{T}^2$.

If each box of type $\mathcal{T}^i$ has bottom area at least $s_i \cdot l \cdot w$, then $(n_1 s_1 + n_2 s_2) \cdot l \cdot w$ is called the *combined area* of the packing generated by the algorithm COMBINE.

The following result about this algorithm holds. The proof is analogous to the one given in [6] for the algorithm COLUMN.

LEMMA 2.7. *Let $\mathcal{P}$ be the packing of $L' \subseteq L$ generated by the algorithm* COMBINE *when applied to lists of types $\mathcal{T}^1$ and $\mathcal{T}^2$ and list of positions $p^i_1, p^i_2, \dots, p^i_{n_i}$, $i = 1, 2$. If $S(b) \geq s_i \cdot l \cdot w$ for all boxes b in $\mathcal{T}^i$ $(i = 1, 2)$, then*

$$H(\mathcal{P}) \leq \frac{1}{s_1 n_1 + s_2 n_2} \frac{V(L')}{l \cdot w} + Z.$$

To simplify the notation, given two lists L_1 and L_2, we denote by COLUMN(L_1, p_1, L_2, p_2) the algorithm COMBINE called with parameters $(L_1||L_2, L_1, p_1, L_2, p_2)$ and assume that it returns a pair $(\mathcal{P}', L')$ where $\mathcal{P}'$ is the partial packing of $L_1||L_2$ and L' is the set of boxes packed in $\mathcal{P}'$.

Another simple algorithm that we use is the algorithm OC (one column). Given a list of boxes, say $L = (b_1, \ldots, b_n)$, this algorithm packs each box b_{i+1} on top of the box b_i for $i = 1, \ldots, n-1$. It is easy to verify the following results.

Lemma 2.8. *If $\mathcal{P}$ is the packing generated by the algorithm* OC *when applied to a list L and s is a constant such that $S(b) \geq s \cdot l \cdot w$ for all boxes b in L, then $H(\mathcal{P}) \leq \frac{V(L)}{s \cdot l \cdot w}$.*

Lemma 2.9. *If $\mathcal{P}$ is the packing generated by the algorithm* OC *when applied to a list L of boxes b such that $b \in \mathcal{R}_4$ and ($\rho(b) \in \mathcal{R}_4$ or $\rho(b) \notin \mathcal{C}$), then $H(\mathcal{P}) = \mathrm{OPT}(L)$.*

We use two other algorithms, UD^x and UD^y, described in [6]. These algorithms are based on the algorithm UD, developed by Baker, Brown, and Kattseff [1] for the strip packing problem. The following results hold for these algorithms [6].

Lemma 2.10. *Let L be an instance for* TPP *such that $b \in \mathcal{C}[\frac{1}{2}, 1\ ;\ 0, 1]$ (resp., $b \in \mathcal{C}[0, 1\ ;\ \frac{1}{2}, 1]$) for all boxes b in L. Then the packing $\mathcal{P}$ generated by the algorithm* UD^x *(resp.,* UD^y*) is such that $H(\mathcal{P}) \leq \frac{5}{4}\mathrm{OPT}(L) + \frac{53}{8}Z$.*

Lemma 2.11. *Let L be an instance for* TPP^z *consisting of boxes b such that $b \in \mathcal{C}[\frac{1}{2}, 1\ ;\ 0, 1]$ (resp., $b \in \mathcal{C}[0, 1\ ;\ \frac{1}{2}, 1]$), and whenever $x(b) > y(b)$ (resp., $y(b) > x(b)$) then $\rho(b) \notin \mathcal{C}$. That is, no two boxes of L can be packed side by side in the x-direction (resp., y-direction). Then the packing $\mathcal{P}$ generated by the algorithm* UD^x *(resp.,* UD^y*) is such that $H(\mathcal{P}) \leq \frac{5}{4}\mathrm{OPT}(L) + \frac{53}{8}Z$.*

Proof. This result follows directly from the previous lemma and the fact that no two boxes can be packed side by side in the x-direction (resp., y-direction), even if rotations are allowed. □

3. The algorithm $\mathbf{R}_k$.

In [6] we presented an algorithm for TPP, called $\mathcal{A}_k$, that has an asymptotic performance bound less than 2.67. In this section we present an algorithm for TPP^z, called R_k, that is based on the algorithm $\mathcal{A}_k$. The algorithm depends on a parameter k, an integer that is assumed to be greater than 5.

Before we give the description of the algorithm we define some numbers which are used to define sublists, called critical sets.

Definition 3.1. *Let $r_1^{(k)}, r_2^{(k)}, \ldots, r_{k+15}^{(k)}$ and $s_1^{(k)}, s_2^{(k)}, \ldots, s_{k+14}^{(k)}$ be real numbers defined as follows:*

- *$r_1^{(k)}, r_2^{(k)}, \ldots, r_k^{(k)}$ are such that* $r_1^{(k)}\frac{1}{2} = r_2^{(k)}(1 - r_1^{(k)}) = r_3^{(k)}(1 - r_2^{(k)}) = \cdots = r_k^{(k)}(1 - r_{k-1}^{(k)}) = \frac{1}{3}(1 - r_k^{(k)})$ *and* $r_1^{(k)} < \frac{4}{9}$*;*
- $r_{k+1}^{(k)} = \frac{1}{3},\ r_{k+2}^{(k)} = \frac{1}{4},\ \ldots,\ r_{k+15}^{(k)} = \frac{1}{17}$*;*
- $s_i^{(k)} = 1 - r_i^{(k)}$ *for* $i = 1, \ldots, k$*;*
- $s_{k+i}^{(k)} = 1 - (\frac{2i+4-\lfloor\frac{i+2}{3}\rfloor}{4i+10})$ *for* $i = 1, \ldots, 14$.

The following result can be proved using a continuity argument.

Claim 3.1. *The numbers $r_1^{(k)}, r_2^{(k)}, \ldots, r_k^{(k)}$ are such that $r_1^{(k)} > r_2^{(k)} > \cdots > r_k^{(k)} > \frac{1}{3}$ and $r_1^{(k)} \to \frac{4}{9}$ as $k \to \infty$.*

For simplicity we omit the superscripts $^{(k)}$ of the notation $r_i^{(k)}, s_i^{(k)}$ when k is clear from the context.

Using the numbers in Definition 3.1, we define the following critical sets.

$$\mathcal{C}_i^A = \mathcal{C}[r_{i+1}, r_i \ ; \ \frac{1}{2}, s_i], \quad \mathcal{C}_i^B = \mathcal{C}[\frac{1}{2}, s_i \ ; \ r_{i+1}, r_i],$$

$$\mathcal{C}^A = \bigcup_{i=1}^{k+14} \mathcal{C}_i^A, \quad \mathcal{C}^B = \bigcup_{i=1}^{k+14} \mathcal{C}_i^B, \quad \mathcal{C}_{[1-k]}^A = \bigcup_{i=1}^{k} \mathcal{C}_i^A, \quad \mathcal{C}_{[1-k]}^B = \bigcup_{i=1}^{k} \mathcal{C}_i^B.$$

The next result refers to a list of positions $p_{i,j}, q_{i,j}, p'_j, q'_j, p''_j$ and q''_j to be considered when applying the algorithm COMBINE. In [6] we give such a list of positions, defined for a box $B = (1, 1, \infty)$. To use in this context, we have to consider a proportional reparameterization for a box $B = (l, w, \infty)$. For completeness, we define here these positions (only for $i < j$, since the case $i > j$ is symmetric). See Figure 3.1(a).

Positions to combine sublists of $\mathcal{C}_i^A$ and $\mathcal{C}_j^B$. For simplicity, we denote by A_i the list of boxes of type $\mathcal{C}_i^A$, and by B_j the list of boxes of type $\mathcal{C}_j^B$.

- To combine the lists A_i $(1 \le i \le k)$ and B_j $(i \le j \le k)$, take
  $p_{i,j} = \left[(0,0), \left(\frac{1}{2}, 0\right)\right]$ and $q_{i,j} = [(0, s_i)]$.
  In this case we have an area guarantee of at least $\frac{1}{2}$.
- To combine the list $A_{[1-k]} = A_1 \cup \cdots \cup A_k$ with B_j $(k+1 \le j \le k+14)$, we consider two phases. We divide $A_{[1-k]}$ into A' and A'', taking $A' = \{b \in A_{[1-k]} : \ x(b) \le 1 - s_j\}$ and $A'' = A_{[1-k]} \setminus A'$.
  - ⋆ To combine A' with B_j, take
    $p'_j = [(s_j, 0)]$ and
    $q'_j = \left[(0,0), \left(0, \frac{1}{j-k+2}\right), \left(0, \frac{2}{j-k+2}\right), \ldots, \left(0, \frac{j-k+1}{j-k+2}\right)\right]$.
    In this case we have an area guarantee of at least $\frac{13}{24}$. This minimum is attained when $j = k+1$.
  - ⋆ To combine A'' with B_j, take
    $p''_j = \left[(0,0), \left(\frac{1}{2}, 0\right)\right]$ and
    $q''_j = \left[\left(0, \frac{2}{3}\right), \left(0, \frac{2}{3} + \frac{1}{j-k+2}\right), \left(0, \frac{2}{3} + \frac{2}{j-k+2}\right), \ldots, \left(0, \frac{2}{3} + \left(\lfloor \frac{j-k+2}{3} \rfloor - 1\right) \frac{1}{j-k+2}\right)\right]$.
    Here we obtain an area guarantee of at least $\frac{27}{56}$.
- To combine the lists A_i $(k+1 \le i \le k+14)$ and B_j $(i \le j \le k+14)$, take
  $p_{i,j} = \left[(s_j, 0), \left(s_j + \frac{1}{i-k+2}, 0\right), \left(s_j + \frac{2}{i-k+2}, 0\right), \ldots, \left(s_j + (\lfloor (1-s_j) \cdot (i-k+2) \rfloor - 1) \frac{1}{i-k+2}, 0\right)\right]$ and
  $q_{i,j} = \left[(0,0), \left(0, \frac{1}{j-k+2}\right), \left(0, \frac{2}{j-k+2}\right), \ldots, \left(0, \frac{j-k+1}{j-k+2}\right)\right]$.
  In this case we also obtain an area guarantee of at least $\frac{27}{56}$.

LEMMA 3.2. *The following statements are valid for the list of positions $p_{i,j}$, $q_{i,j}$, p'_j, q'_j, p''_j, and q''_j:*

(a) *If $\mathcal{P}$ is a packing generated by the algorithm COMBINE with parameters $(L, \mathcal{C}_i^A, p_{i,j}, \mathcal{C}_j^B, q_{i,j})$, $1 \le i, j \le k$ or $k+1 \le i, j \le k+14$, then we have that $H(\mathcal{P}) \le \frac{56}{27} \frac{V(\mathcal{P})}{l \cdot w} + Z$.*

(b) *There is a partition of $\mathcal{C}_{[1-k]}^A$ into sets $\mathcal{C}'_{A,j}$ and $\mathcal{C}''_{A,j}$ such that a packing $\mathcal{P}'$ generated by the algorithm* COMBINE *with parameters $(L, \mathcal{C}'_{A,j}, p'_j, \mathcal{C}_j^B, q'_j)$, $k+1 \le j \le k+14$, is such that $H(\mathcal{P}') \le \frac{56}{27} \frac{V(\mathcal{P}'')}{l \cdot w} + Z$ and a packing $\mathcal{P}''$*

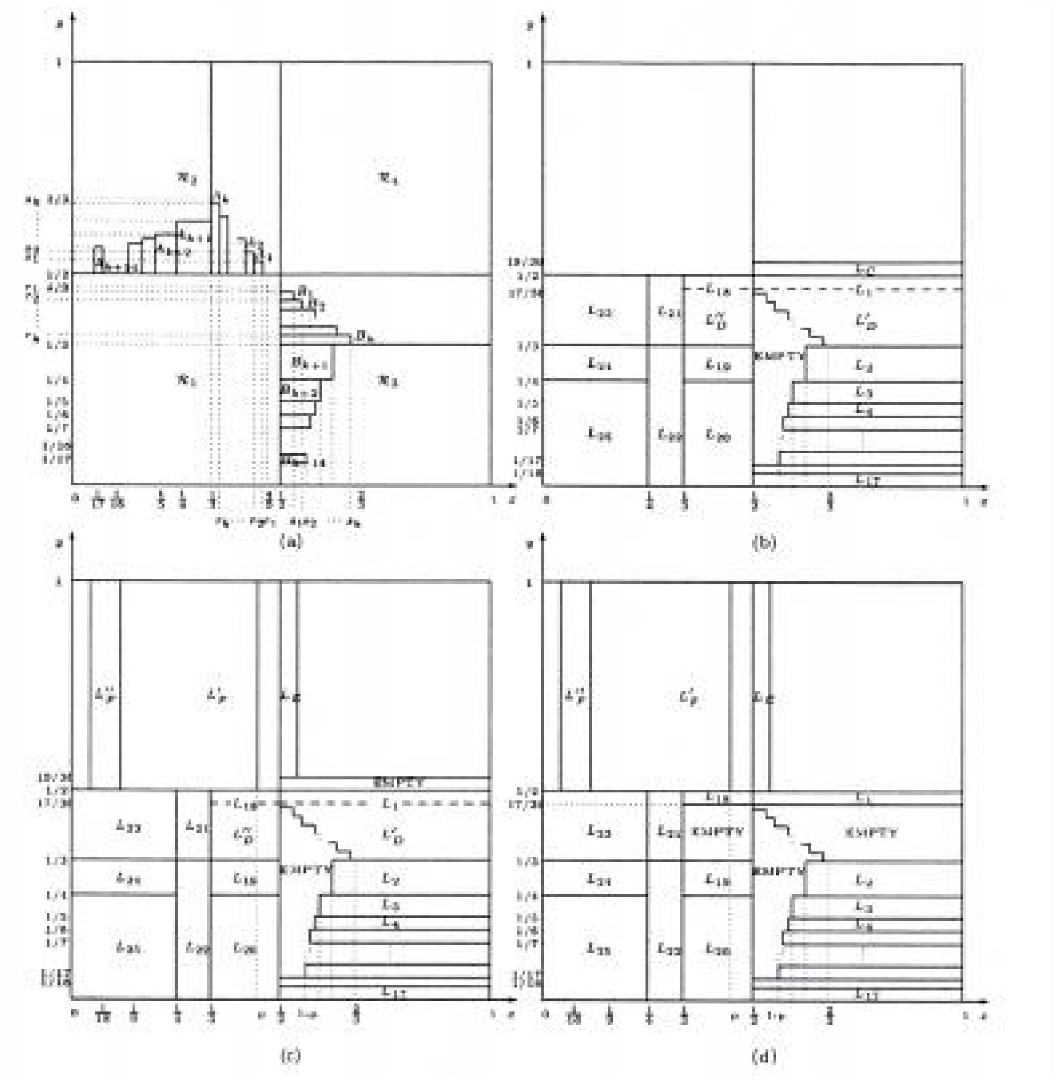


FIG. 3.1. *Partition of list L for algorithm R_k. The sets A_i and B_i in* (a) *correspond to the sets $\mathcal{C}_i^A$ and $\mathcal{C}_i^B$, resp.*

generated by the algorithm COMBINE *with parameters* $(L, \mathcal{C}''_{A,j}, p''_j, \mathcal{C}^B_j, q''_j)$, $k+1 \leq j \leq k+14$, *is such that* $H(\mathcal{P}'') \leq \frac{56}{27}\frac{V(\mathcal{P}'')}{l \cdot w} + Z$.

(c) *Defining positions symmetric to* $p_{i,j}, q_{i,j}, p'_j, q'_j, p''_j$ *and* q''_j, *analogous results hold when the letter A and B are exchanged in the items above.*

The algorithm R_k is inspired by the algorithm $\mathcal{A}_k$ presented in [6]. The reader may compare both algorithms to see where they differ; it should be noted that now there are steps where rotations are performed. This is done because otherwise we may not obtain valid inequalities with respect to the optimum packing.

ALGORITHM R_k

Input: List of boxes L.

```
    Output: Packing $\mathcal{P}$ of $L$ into $B = (l, w, \infty)$.
1 Rotate all boxes $b$ that are in $\mathcal{R}_4$ such that $\rho(b) \in \mathcal{R}_2 \cup \mathcal{R}_3$.
        /* i.e., Let $T \leftarrow \{b \in L \cap \mathcal{R}_4 : \rho(b) \in \mathcal{R}_2 \cup \mathcal{R}_3\}$. $L \leftarrow (L \setminus T) \bigcup \rho(T)$. */
2 Rotate all boxes $b$ of $L$ that are in $\mathcal{R}_2 \cup \mathcal{R}_3$ such that $\rho(b) \in \mathcal{R}_1$.
3 Let $p_{i,j}, q_{i,j}$, $1 \leq i, j \leq k+14$, and $p'_j, p''_j, q'_j, q''_j$, $k+1 \leq j \leq k+14$, be as defined
    above.
4 Combine boxes of types $\mathcal{C}^A$ and $\mathcal{C}^B$ of $L$ as follows (see Figure 3.1(a)).
    4.1 $i \leftarrow 1$; $j \leftarrow 1$; $\mathcal{P}_{AB} \leftarrow \emptyset$.
    4.2 While ($i \leq k$ and $j \leq k$) do
        $\mathcal{P}_{i,j} \leftarrow$ COMBINE($L, \mathcal{C}_i^A, p_{i,j}, \mathcal{C}_j^B, q_{i,j}$) .
        $\mathcal{P}_{AB} \leftarrow \mathcal{P}_{AB} \| \mathcal{P}_{i,j}$ .
        Update the list $L$ removing the packed boxes.
        If all boxes of type $\mathcal{C}_i^A$ have been packed, then increment $i$; else incre-
            ment $j$.
    4.3 If all boxes of type $\mathcal{C}^B_{[1-k]}$ have been packed
        4.3.1 Then
            While ($j \leq k+14$ and there is a box of type $\mathcal{C}^A_[1-k]$) do
                Let $\mathcal{C}'_{A,j}$ and $\mathcal{C}''_{A,j}$ be a partition of $\mathcal{C}^A_{[1-k]}$, as in Lemma 3.2.
                $\tilde{\mathcal{P}}'_j \leftarrow$ COMBINE($L, \mathcal{C}'_{A,j}, p'_j, \mathcal{C}_j^B, q'_j$). Update $L$ removing the
                    packed boxes.
                $\tilde{\mathcal{P}}''_j \leftarrow$ COMBINE($L, \mathcal{C}'_{A,j}, p'_j, \mathcal{C}_j^B, q'_j$). Update $L$ removing the
                    packed boxes.
                $\mathcal{P}_{AB} \leftarrow \mathcal{P}_{AB} \| \tilde{\mathcal{P}}'_j \| \tilde{\mathcal{P}}''_j$.
                if $B_j = \emptyset$, then $j \leftarrow j+1$.
            $i \leftarrow k+1$
        4.3.2 Else /* All boxes of types $\mathcal{C}^A_{[1-k]}$ have been packed */
            Perform steps symmetric to the ones given in the case 4.3.1.
    4.4 While ($i \leq k+14$ and $j \leq k+14$) do
        $\mathcal{P}_{i,j} \leftarrow$ COMBINE($L, \mathcal{C}_i^A, p_{i,j}, \mathcal{C}_j^B, q_{i,j}$). Update $L$ removing the packed
            boxes.
        $\mathcal{P}_{AB} \leftarrow \mathcal{P}_{AB} \| \mathcal{P}_{i,j}$ .
        If all boxes of type $\mathcal{C}_i^A$ have being packed, then increment $i$; else incre-
            ment $j$.
5 If all boxes of type $\mathcal{C}^B$ have been packed, then
    5.1 Rotate the boxes of $L \cap \mathcal{R}_2$ that fit in $\mathcal{R}_3$.
    5.2 Rotate the boxes of $L \cap (\mathcal{R}_2 \cup \mathcal{R}_4)$ such that if $b \in L \cap (\mathcal{R}_2 \cup \mathcal{R}_4)$, then
        $x(b) \leq y(b)$ or $\rho(b) \notin \mathcal{C}$.
    5.3 Subdivide the list $L$ into sublists $L_1, \ldots, L_{25}$ as follows (see Figure 3.1(b)).
```

$$
\begin{array}{ll}
L_i = L \bigcap \mathcal{C}[\frac{1}{2}, 1 \ ; \ \frac{1}{i+2}, \frac{1}{i+1}], \text{ for } i = 1, \ldots, 16 & L_{17} = L \bigcap \mathcal{C}[\frac{1}{2}, 1 \ ; \ 0, \frac{1}{18}], \\
L_{18} = L \bigcap \mathcal{C}[\frac{1}{3}, \frac{1}{2} \ ; \ \frac{1}{3}, \frac{1}{2}], & L_{19} = L \bigcap \mathcal{C}[\frac{1}{3}, \frac{1}{2} \ ; \ \frac{1}{4}, \frac{1}{3}], \\
L_{20} = L \bigcap \mathcal{C}[\frac{1}{3}, \frac{1}{2} \ ; \ 0, \frac{1}{4}], & L_{21} = L \bigcap \mathcal{C}[\frac{1}{4}, \frac{1}{3} \ ; \ \frac{1}{3}, \frac{1}{2}], \\
L_{22} = L \bigcap \mathcal{C}[\frac{1}{4}, \frac{1}{3} \ ; \ 0, \frac{1}{3}], & L_{23} = L \bigcap \mathcal{C}[0, \frac{1}{4} \ ; \ \frac{1}{3}, \frac{1}{2}], \\
L_{24} = L \bigcap \mathcal{C}[0, \frac{1}{4} \ ; \ \frac{1}{4}, \frac{1}{3}], & L_{25} = L \bigcap \mathcal{C}_4, \\
L_C = L \bigcap \mathcal{C}[\frac{1}{2}, 1 \ ; \ \frac{1}{2}, \frac{19}{36}] & L'_D = L_1 \bigcap \mathcal{C}[0, \frac{17}{36} \ ; \ 0, 1], \\
L''_D = L_{18} \bigcap \mathcal{C}[0, \frac{17}{36} \ ; \ 0, 1] & L_D = L'_D \bigcup L''_D.
\end{array}
$$

```
    5.4 Generate packing $\mathcal{P}_{CD}$ as follows.
```

```
          $(\mathcal{P}_{CD'}, L_{CD'}) \leftarrow$ COLUMN$(L_C, [(0,0)], L'_D, [(0, \frac{19}{36})])$.
          $(\mathcal{P}_{CD''}, L_{CD''}) \leftarrow$ COLUMN$(L_C \setminus L_{CD'}, [(0,0)], L''_D, [(0, \frac{19}{36}), (\frac{1}{2}, \frac{19}{36})])$.
          $\mathcal{P}_{CD} \leftarrow \mathcal{P}_{CD'} \| \mathcal{P}_{CD''}$.
          $L_{CD} \leftarrow L_{CD'} \bigcup L_{CD''}$.  $L_1 \leftarrow L_1 \setminus L_{CD}$.  $L_{18} \leftarrow L_{18} \setminus L_{CD}$.
     5.5 Generate packings $\mathcal{P}_1, \ldots, \mathcal{P}_{25}$ as follows.
          $\mathcal{P}_i \leftarrow$ NFDH$^y(L_i)$ for $i = 1, \ldots, 22$.
          $\mathcal{P}_i \leftarrow$ NFDH$^x(L_i)$ for $i = 23, 24$.
          $\mathcal{P}_{25} \leftarrow$ LL$(L_{25}, 4)$.
     5.6 Update $L$ removing the packed boxes. Note that $L \subseteq \mathcal{R}_2 \cup \mathcal{R}_4$.
     5.7 If $L_C \subseteq L_{CD}$
          then /* (Case 1) */
               $p \leftarrow \frac{\sqrt{199145}-195}{570} = 0.440\ldots$.  /* $L_C$ is totally packed (see Figure
                  3.1(c)) */
          else /* (Case 2) $L_D \subseteq L_{CD}$ */
               $p \leftarrow \frac{\sqrt{23401}-71}{180} = 0.455\ldots$.  /* $L_D$ is totally packed (see Figure
                  3.1(d)) */
     5.8  $L_E \leftarrow L \cap \mathcal{C}[\frac{1}{2}, 1-p\ ;\ \frac{1}{2}, 1]$.   $L'_F \leftarrow L \cap \mathcal{C}[\frac{1}{9}, p\ ;\ \frac{1}{2}, 1]$.
          $L''_F \leftarrow L \cap \mathcal{C}[\frac{1}{18}, \frac{1}{9}\ ;\ \frac{1}{2}, 1]$.        $L_F \leftarrow L'_F \cup L''_F$.
     5.9  $(\mathcal{P}_{EF'}, L_{EF'}) \leftarrow$ COLUMN$(L_E, [(0,0)], L'_F, [(1-p, 0)])$.
          $(\mathcal{P}_{EF''}, L_{EF''}) \leftarrow$  COLUMN$(L_E \setminus L_{EF'}, [(0,0)], L''_F, [(0, 1-p),$
                                $(0, 1-p+\frac{1}{9}), \ldots, (0, 1-p+(\lfloor 9p \rfloor - 1)\frac{1}{9})])$.
          $\mathcal{P}_{EF} \leftarrow \mathcal{P}_{EF'} \| \mathcal{P}_{EF''}$.
          $L_{EF} \leftarrow L_{EF'} \cup L_{EF''}$.
     5.10 If $L_E \subseteq L_{EF}$ /* (Subcase 1) $L_E$ is totally packed */
          then
               $\mathcal{P}_{\mathrm{UD}} \leftarrow$ UD$^x(L)$.
               $\mathcal{P}_{OC} \leftarrow$ OC$((L \setminus L_{EF}) \cap \mathcal{R}_4)$.
               $\mathcal{P}_{2e} \leftarrow$ NFDH$^x((L \setminus L_{EF}) \cap \mathcal{C}[0, \frac{1}{3}\ ;\ 0, 1])$.
               $\mathcal{P}_{2d} \leftarrow$ NFDH$^x((L \setminus L_{EF}) \cap \mathcal{C}[p, \frac{1}{2}\ ;\ 0, 1])$.
               $\mathcal{P}' \leftarrow \mathcal{P}_{OC} \| \mathcal{P}_{2e} \| \mathcal{P}_{2d} \| \mathcal{P}_{EF}$.
               $\mathcal{P}'' \leftarrow \{\mathcal{P} \in \{\mathcal{P}_{\mathrm{UD}}, \mathcal{P}'\} :\ H(\mathcal{P})$ is minimum $\}$.
               $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{AB} \| \mathcal{P}_{CD} \| \mathcal{P}_1 \| \ldots \| \mathcal{P}_{25}$.
               Let $L''$ and $L_{aux}$ be the lists of boxes packed in $\mathcal{P}''$ and $\mathcal{P}_{aux}$, resp.
               $\mathcal{P} \leftarrow \mathcal{P}_{aux} \| \mathcal{P}''$.
     5.11 If $L_F \subseteq L_{EF}$ /* (Subcase 2) $L_F$ is totally packed */
          then
               $\mathcal{P}_{OC} \leftarrow$ OC$((L \setminus L_{EF}) \cap \mathcal{R}_4)$.
               $\mathcal{P}_{2e} \leftarrow$ NFDH$^x((L \setminus L_{EF}) \cap \mathcal{C}[0, \frac{1}{18}\ ;\ \frac{1}{2}, 1])$.
               $\mathcal{P}_{2d} \leftarrow$ NFDH$^x((L \setminus L_{EF}) \cap \mathcal{C}[p, 1\ ;\ \frac{1}{2}, 1])$.
               $\mathcal{P}' \leftarrow \mathcal{P}_{OC} \| \mathcal{P}_{EF}$.
               $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{AB} \| \mathcal{P}_{CD} \| \mathcal{P}_{2e} \| \mathcal{P}_{2d} \| \mathcal{P}_1 \| \ldots \| \mathcal{P}_{25}$.
               Let $L'$ and $L_{aux}$ be the lists of boxes packed in $\mathcal{P}'$ and $\mathcal{P}_{aux}$, resp.
               $\mathcal{P} \leftarrow \mathcal{P}_{aux} \| \mathcal{P}'$.
6 If all boxes of type $\mathcal{C}^A$ have been packed then generate a packing $\mathcal{P}$ of $L$ as in step
     5 (in a symmetric way).
7 Return $\mathcal{P}$.
end algorithm.
```

The next theorem gives an asymptotic performance bound of the algorithm R_k when $k \to \infty$.

THEOREM 3.3. *For any instance L of* TPP^z *we have*

$$R_k(L) \leq \alpha_k \cdot \mathrm{OPT}(L) + \left(2k + \frac{597}{8}\right) Z,$$

where $\alpha_k \to \frac{579+\sqrt{199145}}{384} = 2.669\ldots$ *as* $k \to \infty$.

Proof. We present the proof for the case all boxes of type $\mathcal{C}^{\mathcal{B}}$ have been packed (see step 5). The proof of the other case (step 6) is analogous. We consider 4 cases, according to step 5.7 ($L_C \subseteq L_{CD}$), step 5.10 ($L_E \subseteq L_{EF}$), and step 5.11 ($L_F \subseteq L_{EF}$).

As many steps of the algorithm R_k are similar to the ones of the algorithm $\mathcal{A}_k$ for TPP, many of the inequalities obtained in the analysis of $\mathcal{A}_k$ are valid in these cases. We only mention them in the four claims A, B, C, and D below (see [6]).

Case 1.1. ($L_C \subseteq L_{CD}$) and ($L_E \subseteq L_{EF}$).

CLAIM A.

$$H(\mathcal{P}'') \leq \frac{1}{(1-p)\frac{19}{36}} \frac{V(L'')}{l \cdot w} + 4Z \quad \text{and} \quad H(\mathcal{P}_{aux}) \leq \frac{1}{r_1} \frac{V(L_{aux})}{l \cdot w} + (2k+68)Z.$$

Let $\mathcal{H}_1 := H(\mathcal{P}'') - \frac{53}{8}Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - (2k+68)Z$.

Using the definition of $\mathcal{H}_1$ and $\mathcal{H}_2$ in the two inequalities above we obtain

$$\mathrm{OPT}(L) \geq \frac{V(L)}{l \cdot w} = \frac{V(L'')}{l \cdot w} + \frac{V(L_{aux})}{l \cdot w} \geq \frac{(1-p)19}{36}\mathcal{H}_1 + r_1\mathcal{H}_2,$$

that is,

$$\mathrm{OPT}(L) \geq \frac{(1-p)19}{36}\mathcal{H}_1 + r_1\mathcal{H}_2 \ . \tag{3.1}$$

Note that from steps 1, 2, 4, 5.1, and 5.2 the list L'' satisfies the condition of Lemma 2.11. Hence, we have

$$H(\mathcal{P}'') \leq \mathrm{UD}^x(L'') \leq \frac{5}{4}\mathrm{OPT}(L'') + \frac{53}{8}Z \leq \frac{5}{4}\mathrm{OPT}(L) + \frac{53}{8}Z \ ,$$

that is,

$$\mathrm{OPT}(L) \geq \frac{4}{5}\mathcal{H}_1 \ . \tag{3.2}$$

Combining inequalities (3.1) and (3.2), we have

$$\mathrm{OPT}(L) \geq \max\left\{\frac{4}{5}\mathcal{H}_1, (1-p)\frac{19}{36}\mathcal{H}_1 + r_1\mathcal{H}_2\right\}.$$

From the definition of $\mathcal{H}_1$ and $\mathcal{H}_2$, we obtain

$$H(\mathcal{P}) = H(\mathcal{P}'') + H(\mathcal{P}_{aux}) = \mathcal{H}_1 + \mathcal{H}_2 + \left(2k + \frac{597}{8}\right) Z.$$

Using the last inequality in the above equation, we have

$$H(\mathcal{P}) \leq \left(\frac{\mathcal{H}_1 + \mathcal{H}_2}{\max\{\frac{4}{5}\mathcal{H}_1, (1-p)\frac{19}{36}\mathcal{H}_1 + r_1\mathcal{H}_2\}}\right) \mathrm{OPT}(L) + \left(2k + \frac{597}{8}\right) Z.$$

Analyzing the two possibilities for the maximum, we can prove (see [6]) that

$$\alpha'_k(r_1) := \frac{49+95p+180r_1}{144r_1} \geq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\frac{4}{5}\mathcal{H}_1, (1-p)\frac{19}{36}\mathcal{H}_1 + r_1\mathcal{H}_2\}}.$$

Thus,

$$H(\mathcal{P}) \leq \alpha'_k(r_1)\cdot \mathrm{OPT}(L) + \left(2k+\frac{597}{8}\right)Z.$$

Since $r_1 \to \frac{4}{9}$ as $k\to\infty$, we can conclude that $\alpha'_k(r_1) \to \frac{579+\sqrt{199145}}{384}$ as $k\to\infty$.

Case 1.2. $(L_C \subseteq L_{CD})$ and $(L_F \subseteq L_{EF})$.

Claim B.

$$H(\mathcal{P}'') \leq \frac{72}{19}\frac{V(P')}{l\cdot w} + 2Z, \quad \text{and} \quad H(\mathcal{P}_{aux}) \leq \frac{1}{p}\frac{V(L_{aux})}{l\cdot w} + (2k+70)Z.$$

Let $\mathcal{H}_1 := H(\mathcal{P}') - 2Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - (2k+70)Z$.

Then we have

$$\mathrm{OPT}(L) \geq \frac{V(L')}{l\cdot w} + \frac{V(L_{aux})}{l\cdot w} \geq \frac{19}{72}\mathcal{H}_1 + p\mathcal{H}_2. \tag{3.3}$$

Note that each box in $L'\cap\mathcal{R}_4$ considered in step 5.11 cannot be rotated, or if it can be rotated, then it fits in $\mathcal{R}_4$ again. So we can conclude that

$$\mathrm{OPT}(L) \geq \mathrm{OPT}(L') \geq \frac{19}{72}\mathcal{H}_1. \tag{3.4}$$

Proceeding as in Case 1.1, using inequalities (3.3) and (3.4), we have

$$H(\mathcal{P}) \leq \alpha''_k\cdot \mathrm{OPT}(L) + (2k+72)Z,$$

where $\alpha''_k = \frac{53+72p}{72p}$.

Thus, from the analysis of subcases 1.1 and 1.2, we can conclude that

$$\mathcal{A}_k(L) \leq \alpha_k\cdot\mathrm{OPT}(L) + \left(2k+\frac{597}{8}\right)Z,$$

where $\alpha_k \to \alpha'_k(\frac{4}{9}) = \alpha''_k = \frac{\sqrt{199145}+579}{384} = 2.669\ldots$ as $k\to\infty$.

Case 2.1. $(L_D \subseteq L_{CD})$ and $(L_E \subseteq L_{EF})$.

Claim C.

$$H(\mathcal{P}'') \leq \frac{1}{(1-p)\frac{1}{2}}\frac{V(L'')}{l\cdot w} + \frac{53}{8}Z \quad \text{and} \quad H(\mathcal{P}_{aux}) \leq \frac{1}{\frac{1}{4}+\frac{r_1}{2}}\frac{V(L_{aux})}{l\cdot w} + (2k+68)Z.$$

Let $\mathcal{H}_1 := H(\mathcal{P}'') - \frac{53}{8}Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - (2k+68)Z$.

Then we have $\mathrm{OPT}(L) \geq \frac{V(L'')}{l\cdot w} + \frac{V(L_{aux})}{l\cdot w} \geq \frac{1-p}{2}\mathcal{H}_1 + \left(\frac{1}{4}+\frac{r_1}{2}\right)\mathcal{H}_2$.

Using the same idea used in Case 1.1, we have $\mathrm{OPT}(L) \geq \mathrm{OPT}(L'') \geq \frac{4}{5}\mathcal{H}_1$ and thus we obtain $H(\mathcal{P}) \leq \beta'_k(r_1)\mathrm{OPT}(L) + \left(2k+\frac{597}{8}\right)Z$, where $\beta'_k(r_1) = \frac{11+10p+10r_1}{4+8r_1}$.

Case 2.2. $(L_D \subseteq L_{CD})$ and $(L_F \subseteq L_{EF})$.

Claim D.

$$H(\mathcal{P}') \leq 4\frac{V(P')}{l\cdot w} + 2Z, \quad \text{and} \quad H(\mathcal{P}_{aux}) \leq \frac{1}{p}\frac{V(L_{aux})}{l\cdot w} + (2k+70)Z.$$

Let $\mathcal{H}_1 := H(\mathcal{P}') - 2Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - (2k + 70)Z$.

In this case we have $\mathrm{OPT}(L) \geq \frac{V(L')}{l\cdot w} + \frac{V(L_{aux})}{l\cdot w} \geq \frac{1}{4}\mathcal{H}_1 + p\mathcal{H}_2$ and $\mathrm{OPT}(L) \geq \mathcal{H}_1$. Thus, $H(\mathcal{P}) \leq \beta''_k \cdot \mathrm{OPT}(L) + (2k + 72)Z$, where $\beta''_k = \frac{3+4p}{4p}$.

Furthermore, for the given value of p, as in the previous cases, we can conclude that

$$H(\mathcal{P}) \leq \beta_k \cdot \mathrm{OPT}(L) + \left(2k + \frac{597}{8}\right) Z,$$

where $\beta_k \to \beta'_k(\frac{4}{9}) = \beta''_k = \frac{\sqrt{23401}+207}{136} = 2.64\ldots$ as $k \to \infty$.

The theorem follows from the conclusions obtained in all cases analyzed. $\square$

The following result proved in [6] is also valid for this algorithm and can be proved analogously. It shows that for relatively small value of k ($k = 13$) the algorithm R_k has already an asymptotic performance bound that is very close to the value shown for $k \to \infty$.

COROLLARY 3.4. *For any instance L of* TPP^z *and $k \geq 13$ we have*

$$R_k(L) \leq \gamma_k \cdot \mathrm{OPT}(L) + \left(2k + \frac{597}{8}\right) Z,$$

where $\gamma_k = \frac{99+1080r_1^{(k)}+\sqrt{199145}}{864r_1^{(k)}} < 2.67$.

PROPOSITION 3.5. *The asymptotic performance bound of the algorithm R_k, $k \geq 13$, is between* 2.5 *and* 2.67.

Proof. The proof follows directly from Corollary 3.4 and Lemma 2.6 (using $m = 4$). $\square$

4. The Algorithm LS: Boxes in L have square bottoms.

In this section and in the following sections we apply the idea used in algorithm R_k to generate algorithms for particular instances of TPP^z. Here we consider the case in which the list L consists of boxes with square bottoms.

Without loss of generality, we consider that the box B has dimensions $(1, w, \infty)$, $w \geq 1$.

Given a list of boxes $L = (b_1, \ldots, b_n)$, consider the list of points given by the set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. Note that all points lay down in a line on the xy-plane that goes through $(0, 0)$ and $(1, 1)$. We call it a *box-line* (see Figure 4.1). The algorithm consider two cases, according to the position in the x-axis, where the box-line crosses the line $y = \frac{1}{2}$ (that is, in the position $xw = \left(\frac{1}{2w}\right) w$).

ALGORITHM LS.

Input: List of boxes $L \subset \mathcal{Q}[0, 1\ ;\ 0, 1]$.

Output: Packing $\mathcal{P}$ of L into $B = (1, w, \infty)$.

```
1 Take $p := 0.4791964$ and subdivide the list $L$ into sublists $L_1, \ldots, L_7, L_A, L_B, L_C$
      as follows (see Figures 4.1 and 4.2).
```

$$\begin{aligned}
&L_1 = L \bigcap \mathcal{C}[\tfrac{1}{2}, 1\ ;\ \tfrac{1}{2}, 1], \quad &&L_2 = L \bigcap \mathcal{C}[\tfrac{1}{3}, \tfrac{1}{2}\ ;\ \tfrac{1}{2}, 1], \quad &&L_3 = L \bigcap \mathcal{C}[0, \tfrac{1}{3}\ ;\ \tfrac{1}{2}, 1],\\
&L_4 = L \bigcap \mathcal{C}[\tfrac{1}{3}, \tfrac{1}{2}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], &&L_5 = L \bigcap \mathcal{C}[\tfrac{1}{4}, \tfrac{1}{3}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], &&L_6 = L \bigcap \mathcal{C}[0, \tfrac{1}{4}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}],\\
&L_7 = L \bigcap \mathcal{C}[0, \tfrac{1}{3}\ ;\ \tfrac{1}{4}, \tfrac{1}{3}], &&L_8 = L \bigcap \mathcal{C}[0, \tfrac{1}{4}\ ;\ 0, \tfrac{1}{4}], &&L_A = L_2 \bigcap \mathcal{C}[0, 1\ ;\ 0, \tfrac{5}{8}],\\
&L_B = L_3 \bigcap \mathcal{C}[0, 1\ ;\ 0, \tfrac{3}{8}], &&L_C = L_1 \bigcap \mathcal{C}[0, 1\ ;\ 0, \tfrac{5}{8}].
\end{aligned}$$

```
2 Let $x \leftarrow \frac{1}{2w}$.
3 if $x \leq \frac{2}{5}$
```

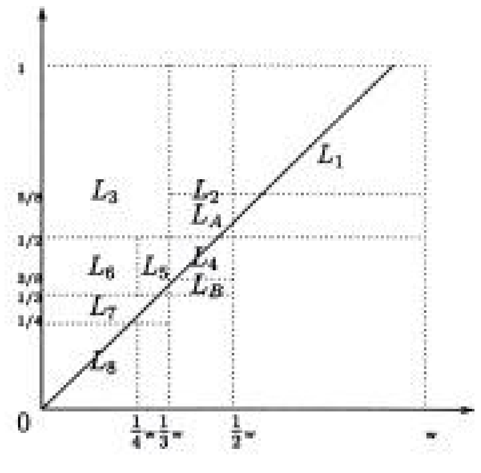


FIG. 4.1. *Partition of L into sublists.*

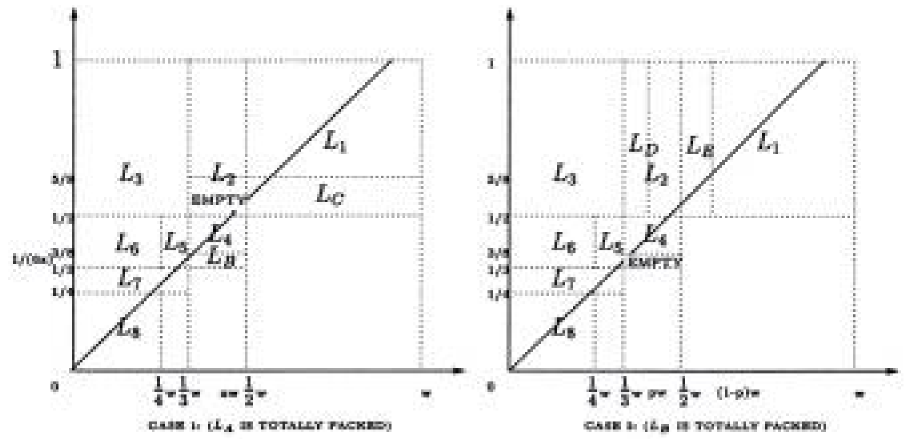


FIG. 4.2. *Combination of sublists L_A and L_B.*

```
then /* (Case 1) this means that there is no box in $L_C$ */
        $\mathcal{P}'_{1,2,3} \leftarrow \text{OC}(L_1)\|\text{NFDH}^x(L_2)\|\text{NFDH}^x(L_3)$.
        $\mathcal{P}''_{1,2,3} \leftarrow \text{UD}(L_1 \cup L_2 \cup L_3)$.
        $\mathcal{P}'' \leftarrow \big(\mathcal{P} \in \{\mathcal{P}'_{1,2,3}, \mathcal{P}''_{1,2,3}\} : \ H(\mathcal{P}) \text{ is minimum}\big)$.
        $\mathcal{P}_{aux} \leftarrow \text{NFDH}^x(L_4)\| \dots \|\text{NFDH}^x(L_7)\|LL(L_8, 4)$.
        $\mathcal{P} \leftarrow \mathcal{P}''\|\mathcal{P}_{aux}$.
        return $\mathcal{P}$.
else /* (Case 2) this means that there is no box in $L_2 \setminus L_A$ */
    4 $(\mathcal{P}_{AB}, L_{AB}) \leftarrow \text{COLUMN}(L_A, [(0,0), (\frac{1}{2}, 0)], L_B, [(0, \frac{5}{8}), (\frac{1}{2}, \frac{5}{8})])$.
    5 $L_4 \leftarrow L_4 \setminus L_{AB}$.      $L_B \leftarrow L_B \setminus L_{AB}$.
    6 (Case 2.1) if $L_A \subseteq L_{AB}$ /* $L_A$ is totally packed. */
          $(\mathcal{P}_{BC}, L_{BC}) \leftarrow \text{COLUMN}(L_C, [(0,0)], L_B, [(0, \frac{5}{8}), (\frac{1}{2}, \frac{5}{8})]$.
          $L_1 \leftarrow L_1 \setminus L_{BC}$. $L_4 \leftarrow L_4 \setminus L_{BC}$.
        (Subcase 2.1.1) $L_B \subseteq L_{BC}$.
```

```
            $\mathcal{P}' \leftarrow \mathrm{OC}(L_1)\|\mathcal{P}_{BC}$.
            $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{AB}\|\mathrm{NFDH}^x(L_4)\|\dots\|\mathrm{NFDH}^x(L_7)\|\mathrm{LL}(L_8,4)$.
        **(Subcase 2.1.2)** $L_C \subseteq L_{BC}$.
            $\mathcal{P}' \leftarrow \mathrm{OC}(L_1)\|\mathcal{P}_{BC}$.
            $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{BC}\|\mathrm{NFDH}^x(L_2)\|\dots\|\mathrm{NFDH}^x(L_7)\|\mathrm{LL}(L_8,4)$.
        $\mathcal{P} \leftarrow \mathcal{P}'\|\mathcal{P}_{aux}$.
        Let $L'$ and $L_{aux}$ be the lists of boxes packed in $\mathcal{P}'$ and $\mathcal{P}_{aux}$, resp.
    **7 (Case 2.2) if** $L_B \subseteq L_{AB}$ /* $L_B$ is totally packed. */
        /* Define two new sublists ($L_D$ and $L_E$) as follows. */
        $L_D = L_2 \bigcap \mathcal{C}[0,p\ ;\ 0,1]$ and $L_E = L_1 \bigcap \mathcal{C}[0,1-p\ ;\ 0,1]$.
        $(\mathcal{P}_{DE}, L_{DE}) \leftarrow \mathrm{COLUMN}(L_D, [(0,0)], L_E, [(p,0)])$.
        $L_1 \leftarrow L_1 \setminus L_{DE}$. $L_2 \leftarrow L_2 \setminus L_{DE}$.
        /* We have two subcases considering the result of this packing. */
        **(Subcase 2.2.1)** if $L_D \subseteq L_{DE}$ or $x \geq p$, $x = \frac{1}{2w} \in \left(p, \frac{1}{2}\right]$ then
            /* Note that when $x \geq p$, $L_D = \emptyset$. */
            $\mathcal{P}' \leftarrow \mathrm{OC}(L_1)\|\mathcal{P}_{DE}$.
            $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{AB}\|\mathrm{NFDH}^x(L_2)\|\dots\|\mathrm{NFDH}^x(L_7)\|\mathrm{LL}(L_8,4)$.
            Let $L'$ and $L_{aux}$ be the lists of boxes packed in $\mathcal{P}'$ and $\mathcal{P}_{aux}$, resp.
            $\mathcal{P} \leftarrow \mathcal{P}'\|\mathcal{P}_{aux}$.
        **(Subcase 2.2.2)** if $L_E \subseteq L_{DE}$ then
            $\mathcal{P}'_{1,2} \leftarrow \mathrm{OC}(L_1)\|\mathrm{NFDH}^x(L_2)\|\mathcal{P}_{DE}$.
            $\mathcal{P}''_{1,2} \leftarrow \mathrm{UD}(L_1 \cup L_2 \cup L_{DE})$.
            $\mathcal{P}'' \leftarrow \left(\mathcal{P} \in \{\mathcal{P}'_{1,2}, \mathcal{P}''_{1,2}\} :\ H(\mathcal{P}) \text{ is minimum}\right)$.
            $\mathcal{P}_{aux} \leftarrow \mathcal{P}_{AB}\|\mathrm{NFDH}^x(L_4)\|\dots\|\mathrm{NFDH}^x(L_7)\|\mathrm{LL}(L_8,4)$.
            Let $L''$ and $L_{aux}$ be the lists of boxes packed in $\mathcal{P}''$ and $\mathcal{P}_{aux}$, resp.
            $\mathcal{P} \leftarrow \mathcal{P}''\|\mathcal{P}_{aux}$.
8 Return $\mathcal{P}$.
**end algorithm.**
```

THEOREM 4.1. *For any instance of* TPP^z *consisting of a list L of boxes with square bottoms, we have*

$$\mathrm{LS}(L) \leq 2.543 \cdot \mathrm{OPT}(L) + \frac{101}{8}Z.$$

Proof. As the proof technique is analogous to the previous one, we only give the inequalities that are valid in each case. We suggest that the reader follow the analysis of each case, together with the corresponding case in the description of the algorithm. Throughout this proof $l = 1$, as we are considering that $B = (1, w, \infty)$.

Case 1. In this case we obtain the following inequalities:

$$H(\mathcal{P}'') \leq \frac{16}{5}\frac{V(L'')}{l \cdot w} + \frac{53}{8}Z,$$
$$H(\mathcal{P}'') \leq \frac{5}{4}\mathrm{OPT}(L) + \frac{53}{8},$$
$$H(\mathcal{P}_{aux}) \leq 2\frac{V(L_{aux})}{l \cdot w} + 5Z.$$

Defining $\mathcal{H}_1 := H(\mathcal{P}'') - \frac{53}{8}Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - 4Z$, we have $\mathrm{OPT}(L) \geq \max\{\frac{4}{5}\mathcal{H}_1, \frac{5}{16}\mathcal{H}_1 + \frac{1}{2}\mathcal{H}_2\}$ and therefore, proceeding as before, we obtain

$$H(\mathcal{P}) \leq \alpha_1 \cdot \mathrm{OPT}(L) + \frac{53}{8}Z,$$

where $\alpha_1 \le \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\frac{4}{5}\mathcal{H}_1, \frac{5}{16}\mathcal{H}_1+\frac{1}{2}\mathcal{H}_2\}} \le 2.5$.

Note that for the remaing cases, the lists L_3 and $L_2 \setminus L_A$ are empty and the box-line crosses the region L_C.

Subcase 2.1.1. Note that in this case, $L_A \cup L_B$ is totally packed in $\mathcal{P}_{AB} \| \mathcal{P}_{BC}$. Note also that the boxes of L_C in $\mathcal{P}_{BC}$ are of type $\mathcal{R}_4$ and therefore we obtain the following inequality:

$$H(\mathcal{P}') \le 4\frac{V(L')}{l\cdot w} + Z.$$

Since

$$H(\mathcal{P}_{aux}) \le 2\frac{V(L_{aux})}{l\cdot w} + 7Z,$$
$$H(\mathcal{P}') \le \mathrm{OPT}(L) + Z,$$

using the above inequalities and defining $\mathcal{H}_1 := H(\mathcal{P}') - Z$ and $\mathcal{H}_2 := H(\mathcal{P}_{aux}) - 7Z$, we have

$$H(\mathcal{P}) \le \alpha_{2,1,1} \cdot \mathrm{OPT}(L) + 8Z,$$

where $\alpha_{2,1,1} \le \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{1}{4}\mathcal{H}_1+\frac{1}{2}\mathcal{H}_2\}} \le 2.5$.

Subcase 2.1.2. In this case the boxes of $L_A \cup L_C$ are totally packed in $\mathcal{P}_{AB} \| \mathcal{P}_{BC}$. Furthermore, we have $x = \frac{1}{2w}$ and $x \in (\frac{2}{5}, \frac{1}{2}]$ (note that $x \cdot w$ is the position in the x-axis where the box-line crosses the line $y = \frac{1}{2}$). In this case we have the following inequalities with respect to x:

$$H(\mathcal{P}') \le \frac{32}{25x}\frac{V(L')}{l\cdot w} + Z,$$
$$H(\mathcal{P}_{aux}) \le \frac{1}{\min\{\frac{2}{9x}, \frac{1}{2}\}}\frac{V(L_{aux})}{l\cdot w} + 8Z,$$
$$H(\mathcal{P}') \le \mathrm{OPT}(L) + Z,$$

and therefore,

$$H(\mathcal{P}) \le \alpha_{2,1,2} \cdot \mathrm{OPT}(L) + 9Z \ ,$$

where $\alpha_{2,1,2} \le \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{25x}{32}\mathcal{H}_1+\min\{\frac{2}{9x}, \frac{1}{2}\}\mathcal{H}_2\}}$. Evaluating the value of $\alpha_{2,1,2}$, when $x \ge \frac{4}{9}$ and when $x < \frac{4}{9}$, we obtain that $\alpha_{2,1,2} \le 2.5$.

Subcase 2.2.1. In this case $L_B \cup L_D$ is totally packed in $\mathcal{P}_{AB} \| \mathcal{P}_{DE}$. Recall that $x = \frac{1}{2w}$. We divide the analysis in two cases. We consider first $x \in (p, \frac{1}{2}]$. Here we have

$$H(\mathcal{P}') \le 8x\frac{V(L')}{l\cdot w} + Z,$$
$$H(\mathcal{P}_{aux}) \le \frac{1}{x}\frac{V(L_{aux})}{l\cdot w} + 7Z,$$
$$H(\mathcal{P}') \le \mathrm{OPT}(L) + Z,$$

and therefore,

$$H(\mathcal{P}) \le \alpha_{2,2,1} \cdot \mathrm{OPT}(L) + 8Z \ ,$$

where $\alpha_{2,2,1} \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{1}{8x}\mathcal{H}_1+x\mathcal{H}_2\}} \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{1}{8p}\mathcal{H}_1+p\mathcal{H}_2\}} \leq 2.543$.

If $x \in (\frac{2}{5}, p]$, the analysis is similar and is omitted.

Subcase 2.2.2. Here $L_B \cup L_E$ is totally packed in $\mathcal{P}_{AB} \| \mathcal{P}_{DE}$. Let $x = \frac{1}{2w}$, $x \in (\frac{2}{5}, p]$. In this case,

$$H(\mathcal{P}'') \leq \frac{2x}{(1-p)^2}\frac{V(L')}{l \cdot w} + \frac{53}{8}Z,$$
$$H(\mathcal{P}_{aux}) \leq 2\frac{V(L_{aux})}{l \cdot w} + 5Z,$$
$$H(\mathcal{P}'') \leq \frac{7}{5}4\mathrm{OPT}(L) + \frac{53}{8}Z,$$

and so,

$$H(\mathcal{P}) \leq \alpha_{2,2,2} \cdot \mathrm{OPT}(L) + \frac{93}{8}Z,$$

where $\alpha_{2,2,2} \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\frac{4}{5}\mathcal{H}_1, \frac{(1-p)^2}{2x}\mathcal{H}_1+\frac{1}{2}\mathcal{H}_2\}} \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\frac{4}{5}\mathcal{H}_1, \frac{(1-p)^2}{2p}\mathcal{H}_1+\frac{1}{2}\mathcal{H}_2\}} \leq 2.543$.

In fact, the value of p was taken in such a manner that the two subcases above (2.2.1 and 2.2.2) lead to the same bound.

The theorem follows considering the cases analyzed above. □

PROPOSITION 4.2. *The asymptotic performance bound of the algorithm LS is between* 2.5 *and* 2.5425.

Proof. The proof follows directly from Theorem 4.1 and Lemma 2.6 (when we use $m = 4$). □

5. The Algorithm BS: Box *B* has a square bottom.

We now consider the special case of TPP^z where B has a square bottom. Without loss of generality, we consider $B = (1, 1, \infty)$.

First, we present an algorithm called NFDH$_p^{xy}$, $0 < p < 1$, that is used as a subroutine. This algorithm packs the boxes of a list L in the following way. Initially, it sorts L in a nonincreasing order of height, then generates a packing divided into levels. Each level is divided into two parts; the boxes are packed first in the region $[0, 1) \times [0, p)$ and then in the region $[0, 1) \times [1 - p, 1)$. The boxes are packed in the region $[0, 1) \times [0, p)$ using the algorithm NFDH^x until a box b_i cannot be packed in the same level; then NFDH$_p^{xy}$ uses the algorithm NFDH^y to pack boxes $\rho(b_i), \rho(b_{i+1}), \ldots$ in the region $[0, 1) \times [1 - p, 1)$ until a box b_k cannot be packed in the same level. At this point, the algorithm NFDH$_p^{xy}$ considers the two parts as only one level and continues to pack the box b_k in a new level. The process continues until all boxes in L have been packed.

Another variant of the above algorithm is called NFDH$_p^{yx}$. This algorithm is similar to the NFDH$_p^{xy}$ algorithm, except that NFDH$_p^{yx}$ first packs boxes b with $x(b) \leq p$, in the y-axis direction, and then packs the next boxes in the x-axis direction.

The following notation is used in the description of the algorithm:

$$\mathcal{X}' := \{b_i = (x_i, y_i, z_i) : \ y_i \leq 1 - x_i\}.$$

ALGORITHM BS.

Input: List of boxes L.

Output: Packing $\mathcal{P}$ of L into $B = (1, 1, \infty)$.

```
1 Rotate the boxes of $L$ in such a way that for each box $b$, $x(b) \leq y(b)$.
```

```
   Take $p = 0.43322958$ and $q = 1 - p$.
2  Divide $L$ into sublists $L'_1, L'_2, L'_3, L_A, L_B, L_C, L_4, \dots, L_{14}$ as follows (see Figure 5.1).
```

$$
\begin{array}{lll}
L'_1 = L \bigcap \mathcal{C}[q, 1\ ;\ q, 1], & L_A = L \bigcap \mathcal{C}[\tfrac{1}{2}, q\ ;\ \tfrac{1}{2}, 1], & L'_2 = L \bigcap \mathcal{C}[p, \tfrac{1}{2}\ ;\ \tfrac{1}{2}, 1] \backslash \mathcal{X}', \\
L_B = L \bigcap \mathcal{C}[\tfrac{1}{3}, p\ ;\ \tfrac{1}{2}, 1] \backslash \mathcal{X}', & L'_3 = L \bigcap \mathcal{C}[p, \tfrac{1}{2}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], & L_C = L \bigcap \mathcal{C}[\tfrac{1}{3}, p\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], \\
L_4 = L \bigcap \mathcal{C}[\tfrac{1}{4}, \tfrac{1}{3}\ ;\ \tfrac{2}{3}, 1], & L_5 = L \bigcap \mathcal{C}[\tfrac{1}{5}, \tfrac{1}{4}\ ;\ \tfrac{2}{3}, 1], & L_6 = L \bigcap \mathcal{C}[0, \tfrac{1}{5}\ ;\ \tfrac{8}{13}, 1], \\
L_7 = L \bigcap \mathcal{C}[\tfrac{1}{3}, \tfrac{1}{2}\ ;\ \tfrac{1}{2}, \tfrac{2}{3}] \bigcap \mathcal{X}', & L_8 = L \bigcap \mathcal{C}[\tfrac{1}{4}, \tfrac{1}{3}\ ;\ \tfrac{1}{2}, \tfrac{2}{3}], & L_9 = L \bigcap \mathcal{C}[\tfrac{1}{5}, \tfrac{1}{4}\ ;\ \tfrac{1}{2}, \tfrac{2}{3}], \\
L_{10} = L \bigcap \mathcal{C}[0, \tfrac{1}{5}\ ;\ \tfrac{1}{2}, \tfrac{8}{13}], & L_{11} = L \bigcap \mathcal{C}[\tfrac{1}{4}, \tfrac{1}{3}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], & L_{12} = L \bigcap \mathcal{C}[0, \tfrac{1}{4}\ ;\ \tfrac{1}{3}, \tfrac{1}{2}], \\
L_{13} = L \bigcap \mathcal{C}[0, \tfrac{1}{3}\ ;\ \tfrac{1}{4}, \tfrac{1}{3}], & L_{14} = L \bigcap \mathcal{C}[0, \tfrac{1}{4}\ ;\ 0, \tfrac{1}{4}] &
\end{array}
$$

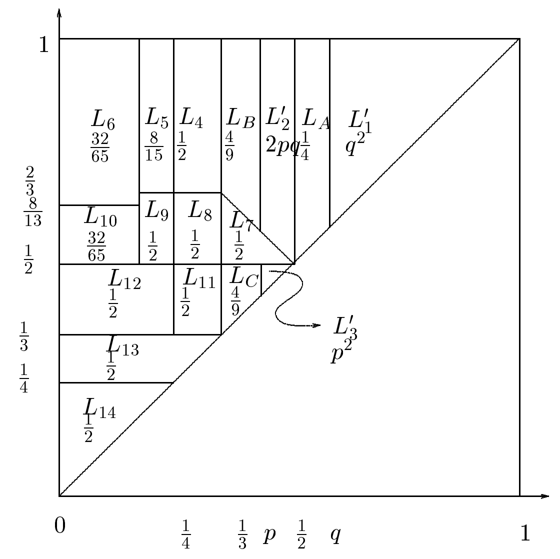


FIG. 5.1. *Partition of list L done by algorithm* BS.

```
3  $(\mathcal{P}_{AB}, L_{AB}) \leftarrow$ COLUMN$(L_A, [(0,0)], L_B, [(q,0)])$.
4  $(\mathcal{P}_{AC}, L_{AC}) \leftarrow$ COLUMN$(L_A \setminus L_{AB}, [(0,0)], L_C, [(q,0), (q, \frac{1}{2})])$.
5  $L_1 \leftarrow (L'_1 \cup L_A) \setminus (L_{AB} \cup L_{AC})$.  $L_2 \leftarrow (L'_2 \cup L_B) \setminus L_{AB}$.  $L_3 \leftarrow (L'_3 \cup L_C) \setminus L_{AC}$.
6  Let $\mathcal{P}_7$ be a packing of $L_7$ obtained as follows.
       6.1  Sort $L_7$ in a nonincreasing order of height.
       6.2  Construct a partition of $L_7$ given by $L_7^1, L_7^2, \dots, L_7^{n_7}$ such that
```

$$
\begin{cases}
L_7 & = L_7^1 \| L_7^2 \| \dots \| L_7^{n_7}, \\
|L_7^i| & = 3, \quad i = 1, \dots, n_7 - 1, \\
|L_7^{n_7}| & \leq 3.
\end{cases}
$$

```
       6.3  Generate a packing $\mathcal{P}_7^i$ of $L_7^i$, $i = 1, \dots, n_7$ as follows.
            6.3.1  Choose $b \in L_7^i$, such that $x(b)$ is minimum.
            6.3.2  Pack $\rho(b)$ in the position $(0, 1 - x(b))$ and the boxes in $L_7^i \setminus \{b\}$
                   in the positions $(0,0)$ and $(\frac{1}{2}, 0)$.
       6.4  $\mathcal{P} \leftarrow \mathcal{P}_7^1 \| \dots \| \mathcal{P}_7^{n_7}$.
7  $\mathcal{P}' \leftarrow$ OC$(L_1) \| \mathcal{P}_{AB} \| \mathcal{P}_{AC}$.
```

```
8  $\mathcal{P}_{aux} \leftarrow \mathrm{NFDH}^x(L_2)\| \dots \|\mathrm{NFDH}^x(L_6)\|\mathcal{P}_7\|\mathrm{NFDH}^{xy}_{\frac{2}{3}}(L_8)\|\mathrm{NFDH}^{xy}_{\frac{2}{3}}(L_9)\|$
       $\mathrm{NFDH}^{xy}_{\frac{8}{13}}(L_{10})\|\mathrm{NFDH}^x(L_{11})\| \dots \|\mathrm{NFDH}^x(L_{13})\|\mathrm{LL}(L_{14})$.
9  $\mathcal{P} \leftarrow \mathcal{P}'\|\mathcal{P}_{aux}$.
10 Return $\mathcal{P}$.
end algorithm.
```

THEOREM 5.1. *For any list L for the* TPP^z*, where B has a square bottom,*

$$\mathrm{BS}(L) \leq 2.528 \cdot \mathrm{OPT}(L) + 15Z\ .$$

Proof. From steps 3 and 4, we can conclude that either L_A is totally packed or $L_B \cup L_C$ is totally packed in $\mathcal{P}_{AB}\|\mathcal{P}_{BC}$. So we divide the proof into these two subcases.

Case 1. L_A is totally packed in $\mathcal{P}_{AB}\|\mathcal{P}_{BC}$.

Here we have

$$H(\mathcal{P}') \leq \frac{1}{q^2}V(L') + 2Z,$$
$$H(\mathcal{P}_{aux}) \leq \frac{9}{4}V(L_{aux}) + 13Z,$$
$$H(\mathcal{P}') \leq \mathrm{OPT}(L) + 2Z.$$

As before, we have $H(\mathcal{P}) \leq \alpha_1 \cdot \mathrm{OPT}(L) + 15Z$, where $\alpha_1 \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, q^2\mathcal{H}_1+\frac{4}{9}\mathcal{H}_2\}} \leq 2.528$.

Case 2. $(L_B \cup L_C)$ is totally packed in $\mathcal{P}_{AB}\|\mathcal{P}_{BC}$.

Here we have

$$H(\mathcal{P}') \leq 4V(L') + 2Z,$$
$$H(\mathcal{P}_{aux}) \leq \frac{1}{2pq}V(L_{aux}) + 13Z,$$
$$H(\mathcal{P}') \leq \mathrm{OPT}(L) + 2Z.$$

Analogously, we have $H(\mathcal{P}) \leq \alpha_2 \cdot \mathrm{OPT}(L) + 15Z$, where $\alpha_2 \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{1}{4}\mathcal{H}_1+2pq\mathcal{H}_2\}} \leq 2.528$.

From the two cases above, the theorem follows. □

PROPOSITION 5.2. *The asymptotic performance bound of the algorithm* BS *is between* 2.5 *and* 2.528.

Proof. It follows directly from Theorem 5.1 and Lemma 2.6 (using $m = 4$). □

6. The Algorithm SS: Boxes in L and box B have square bottoms.

Now we consider the special case of TPP^z, where all boxes in L and box B have square bottoms. Without loss of generality, we take $B = (1, 1, \infty)$.

In 1990, Li and Cheng [3] presented an algorithm for this problem with asymptotic performance bound 2.6875. The algorithm we present here, called SS, has an asymptotic performance bound of 2.361.

Here we need an algorithm, called GQ$_m$, described in [3], to pack boxes in $\mathcal{Q}_m$.

The algorithm GQ$_m$ works in the same way as algorithm LL. It sorts the boxes in $L \subset \mathcal{Q}_m$ in nonincreasing order of their height, divides L into sublists $L_1, \dots, L_v$, and uses the same two-dimensional packing algorithm to pack each sublist L_i in a level. The only place where algorithm GQ$_m$ differs from LL is the bottom area size used to subdivide L into sublists L_i. This is because the two-dimensional packing algorithm used by algorithm LL can guarantee a better area if all boxes have square bottoms. In this case the sublists L_i satisfy the following inequalities:

$$S(L_i) \le \left[\left(\tfrac{m-1}{m}\right)^2 + \left(\tfrac{1}{m}\right)^2\right] lw \qquad \text{for } i = 1, \dots, v,$$
$$S(L_i) + S(first(L_{i+1})) > \left[\left(\tfrac{m-1}{m}\right)^2 + \left(\tfrac{1}{m}\right)^2\right] lw \qquad \text{for } i = 1, \dots, v-1.$$

The following result is proved in [3].

LEMMA 6.1. *Let $L \subset \mathcal{Q}_m$, $m \ge 2$. Then, $GQ_m(L) \le (\frac{m}{m-1})^2 \frac{V(L)}{lw} + Z$.*

Analogously to Lemma 2.6 with algorithm LL, we can show that the following result holds for algorithm GQ$_m$.

LEMMA 6.2. *Let $\mathcal{A}$ be an algorithm for* TPP^z *to pack a list $L \subset \mathcal{Q}[0,1\ ;\ 0,1]$ into a box $B = (1,1,\infty)$. If $\mathcal{A}$ subdivides the input list L into two sublists $L_1 \subset \mathcal{R}_4$ and $L_2 \subset \mathcal{Q}_m$, $m \ge 2$, and applies algorithm* GQ$_m$ *to pack L_2, then the asymptotic performance bound of $\mathcal{A}$ is at least $\frac{4(m-1)^2+3m^2}{4(m-1)^2}$.*

Proof. The proof is similar to the proof of Lemma 2.6, now using the value $\left(\frac{m-1}{m}\right)^2$ instead of $\left(\frac{m-2}{m}\right)$. ∎

```
ALGORITHM SS.
  Input: List of boxes $L \subset \mathcal{Q}[0,1\ ;\ 0,1]$.
  Output: Packing $\mathcal{P}$ of $L$ into $B = (1,1,\infty)$.
1 Take $p = 0.37123918$ and $q = 1 - p$.
  Divide $L$ into sublists, $L_1', L_A, L_2', L_B, L_3$, and $L_4$ (see Figure 6.1),
      $L_1' = L \bigcap \mathcal{Q}[\frac{1}{2},1\ ;\ \frac{1}{2},1]$,   $L_A = L \bigcap \mathcal{Q}[\frac{1}{2},q\ ;\ \frac{1}{2},q]$,
      $L_2' = L \bigcap \mathcal{Q}[\frac{1}{3},\frac{1}{2}\ ;\ \frac{1}{3},\frac{1}{2}]$,   $L_B = L \bigcap \mathcal{Q}[\frac{1}{3},p\ ;\ \frac{1}{3},p]$,
      $L_3 = L \bigcap \mathcal{Q}[\frac{1}{4},\frac{1}{3}\ ;\ \frac{1}{4},\frac{1}{3}]$,   $L_4 = L \bigcap \mathcal{Q}[0,\frac{1}{4}\ ;\ 0,\frac{1}{4}]$.
2 $(\mathcal{P}_{AB}, L_{AB}) \leftarrow$ COLUMN$(L_A, [(0,0)], L_B, [(0,q),(q,q),(q,0)])$.
3 $L_i \leftarrow L_i' \setminus L_{AB}$ for $i = 1,2$.
4 $\mathcal{P}_1 \leftarrow$ OC$(L_1) \| \mathcal{P}_{AB}$.
5 $\mathcal{P}_{aux} \leftarrow$ NFDH$^x(L_2) \|$NFDH$^x(L_3) \| GQ_4(L_4)$.
6 $\mathcal{P} = \mathcal{P}_1 \| \mathcal{P}_{aux}$.
7 Return $\mathcal{P}$.
end algorithm.
```

THEOREM 6.3. *For any list L for* TPP^z*, where all boxes have square bottoms,*

$$\text{SS}(L) \le 2.361 \cdot \text{OPT}^z(L) + 4Z\ .$$

Proof. Again we analyze two cases, considering the packing generated in step 2.

Case 1. L_A is totally packed in $\mathcal{P}_{AB}$.

$$H(\mathcal{P}_1) \le \frac{1}{q^2} V(L') + Z,$$
$$H(\mathcal{P}_{aux}) \le \frac{9}{4} V(L_{aux}) + 3Z,$$
$$H(\mathcal{P}_1) \le \text{OPT}(L) + Z.$$

Proceeding as before, we have $H(\mathcal{P}) \le \alpha_1 \cdot \text{OPT}(L) + 4Z$, where

$$\alpha_1 \le \frac{\mathcal{H}_1 + \mathcal{H}_2}{\max\{\mathcal{H}_1, q^2\mathcal{H}_1 + \frac{4}{9}\mathcal{H}_2\}} \le 2.361.$$

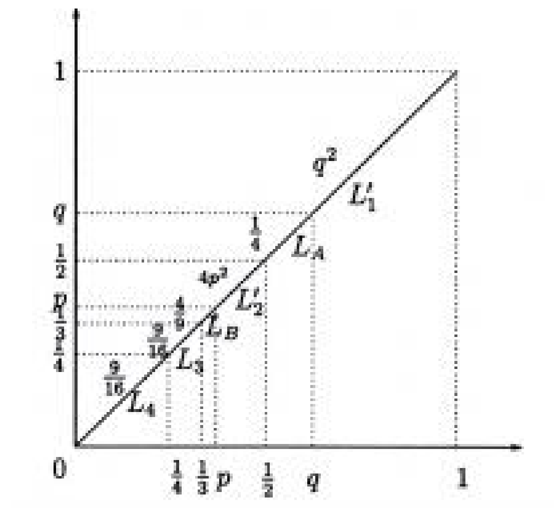


FIG. 6.1. *Partition of list L done by Algorithm* SS.

Case 2. L_B is totally packed in $\mathcal{P}_{AB}$.

Here we have

$$H(\mathcal{P}_1) \leq 4V(L') + Z,$$
$$H(\mathcal{P}_{aux}) \leq \frac{1}{4p^2}V(L_{aux}) + 3Z,$$
$$H(\mathcal{P}_1) \leq \mathrm{OPT}(L) + Z.$$

Thus, $H(\mathcal{P}) \leq \alpha_2 \cdot \mathrm{OPT}(L) + 4Z$, where $\alpha_2 \leq \frac{\mathcal{H}_1+\mathcal{H}_2}{\max\{\mathcal{H}_1, \frac{1}{4}\mathcal{H}_1+4p^2\mathcal{H}_2\}} \leq 2.361$.

From the two cases above, the theorem follows. □

PROPOSITION 6.4. *The asymptotic performance bound of Algorithm* SS *is between* 2.333 *and* 2.361.

Proof. It follows directly from Theorem 6.3 and Lemma 6.2 (with $m = 4$). □

REFERENCES

[1] B. S. BAKER, D. J. BROWN, AND H. P. KATSEFF, *A $\frac{5}{4}$ algorithm for two-dimensional packing*, J. Algorithms, 2 (1981), pp. 348–368.

[2] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*, Freeman, San Francisco, 1979.

[3] K. LI AND K.-H. CHENG, *On three-dimensional packing*, SIAM J. Comput., 19 (1990), pp. 847–867.

[4] K. LI AND K.-H. CHENG, *Static job scheduling in partitionable mesh connected systems*, J. Parallel and Distributed Computing, 10 (1990), pp. 152–159.

[5] K. LI AND K.-H. CHENG, *Heuristic algorithms for on-line packing in three dimensions*, J. Algorithms, 13 (1992), pp. 589–605.

[6] F. K. MIYAZAWA AND Y. WAKABAYASHI, *An algorithm for the three-dimensional packing problem with asymptotic performance analysis*, Algorithmica, 18 (1997), pp. 122–144.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 3, pp. 1030–1049

EXAMINING COMPUTATIONAL GEOMETRY, VAN EMDE BOAS TREES, AND HASHING FROM THE PERSPECTIVE OF THE FUSION TREE*

DAN E. WILLARD†

Abstract. This article illustrates several examples of computer science problems whose performance can be improved with the use of either the fusion trees [Fredman and Willard, *J. Comput. System Sci.*, 47 (1993), pp. 424–436; Fredman and Willard, *J. Comput. System Sci.*, 48 (1994), pp. 533–551] or one of several recent improvements to this data structure. It is likely that many other data structures can also have their performance improved with fusion trees. The examples here are only illustrative.

Key words. sorting, searching, hashing, computational geometry, multidimensional retrieval

AMS subject classifications. 68O20, 68O25, 68P05, 68P10

PII. S0097539797322425

1. Introduction.

The fusion tree [25] and its several variants [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 26, 36, 37, 38, 40, 41, 43, 44, 45, 46] are data structures that perform sorting operations in better than $O(N \ \log N)$ worst-case time and execute dynamic search operations in faster than $O(\log N)$ time in a very natural model of complexity. Since many classic algorithms were designed under the assumption that further improvements for sorting and searching were impossible, one would intuitively anticipate many of the classic search procedures to permit $O(\log\log N)$ or better time improvements when their use of conventional dynamic search trees is simply replaced by the faster underlying data structures made theoretically possible by new advances in data structure theory.

This article will illustrate several examples of such improvements. Most of these improvements will use a data structure, called the q-heap, as an intermediate device to speed up the search methodology. Q-heaps were introduced in [26] as a vehicle for solving the minimum spanning tree problem in linear time. However, they also have many other applications. Q-heaps will be shown in this article to improve many other classic searching problems.

The particular examples and new results we establish are listed at the bottom of this paragraph. None of the so-called improvements in this list are practical improvements because the coefficients associated with their asymptotic changes are undesirably large. However, item 1 below (dynamic universal hashing) is a foundational problem in computer science. All the other topics already appear (in some form) in volumes 1 and 3 of Mehlhorn's textbook [33, 34] as well as in many other textbooks [23, 27, 39]. Thus our improvements are very germane to a two-semester course on general algorithms and computational geometry. The six topics below are only intended as a representative sample to illustrate what can be done theoretically (albeit

*Received by the editors June 9, 1997; accepted for publication (in revised form) March 24, 1998; published electronically January 11, 2000. A preliminary version of this paper [53] appeared as *Applications of the fusion tree method to computational geometry and searching*, in Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, PA, 1992, pp. 386–395. This work was partially supported by NSF grants CCR 9302920 and CCR 9902726.
http://www.siam.org/journals/sicomp/29-3/32242.html

†Computer Science Department, University at Albany-SUNY, Albany, NY 12222 (dew@cs.albany.edu).

not practically) when fusion trees and q-heaps are applied to several problems:

1. A variant of dynamic hashing where each lookup, insertion, and deletion respects an $O(1)$ time bound with a probability exceeding $1 - o(N^{-(\log N)^K})$, for any arbitrary constant $K > 0$. (This result is a significant improvement over Dietzfelbinger and Meyer auf der Heide's universal hash scheme [22]. It will probably be the most noteworthy result of this paper because hashing is one of the basic operations studied in computer science. The improvement is based on combining the fusion-tree formalism with some of Siegel's theorems [42] about universal hashing.)
2. An improved version of McCreight's priority-search trees [35] that reduces the worst-case dynamic time performance to $O(\log N / \log \log)$ can search a set of N rectangles and output all K intersections among this set in time $O(K + N \log N / \log \log N)$. See Mehlhorn's textbook [34] for a concise description of priority-search trees.)
3. A related $\log \log N$ improvement of Chazelle's $O(N \log^{d-1} N / \log \log N)$ space data structure [16] that performs the reporting version of d-dimensional orthogonal range queries in time $O(\log^{d-1} N / \log \log N + K)$, where K is the number of elements outputted.
4. For any dimension $d \geq 2$ and any $p \geq d-1$, it will be possible to construct an $O(N \log^p N)$ space structure that supports an $O\{ (\log N / \log \log N)^{d-1} \}$ time for executing the aggregate version of orthogonal range queries. (Section 6 will explain how there are at least certain perspectives where this combination of time and space can be viewed as optimal.)
5. An improved variant of Van Emde Boas trees [48, 47, 28] that has a somewhat improved memory space.
6. A $\log \log N$ speedup of the Bentley–Shamos linear space method [10, 14] that calculates a set of N d-dimensional ECDF statistics in a time asymptote of $O(N \log^{d-1} N / \log \log N)$. (The acronym ECDF is an often-used abbreviation for "empirical cumulative distribution function.")

Each of the results above will employ the q-heap data structure from [26]. Section 9 will briefly illustrate several further examples of algorithmic problems whose performance can be improved with merely a faster sorting algorithm. In particular, such problems can have their runtimes improved by either the Fredman–Willard sorting procedure or by the further subsequent improvements of this procedure that have been proposed by Andersson [4], Andersson et al. [5], and Thorup [44, 46].

We do not believe that any of the six sample topics mentioned in the preceding paragraph are the main point. Rather the key question is how many other well-known results in computer science can undergo a similar theoretical (albeit perhaps impractical) asymptotic improvement when some type of application of fusion trees, q-heaps and their generalizations are employed [25, 26]? For instance, Thorup recently discovered a worst-case linear-time algorithm for solving Dijkstra's single source shortest path (SSSP) problem with a method that uses some of the constructs of [26] in one of its main interim steps. In the present article, we focus mostly on a moderately narrow bandwidth of problems, drawn from computational geometry and information retrieval theory, simply because such problems reflect the author's particular expertise and knowledge. However, it will become apparent to the reader who examines our six sample problems in the context of the expanding literature [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46] that there are surely many other problems that can undergo theoretical (although often not practical) improvements when some type of variant of fusion tree is present.

Thus our goal in this article will not be exclusively to address the six sample problems mentioned. Sections 2, 3, and 4 are written so that they can be understood by a reader who is unacquainted with both fusion trees and computational geometry. Their goal will be to provide such a reader with an intuitive feel for this subject, ending with a very curious example about hashing. Our more specialized discussion appears in sections 4 through 9. They focus mostly around computational geometry and Van Emde Boas trees.

2. Literature survey. We will use largely the same notation and computational model that appeared in the articles [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46]. A word is assumed to consist of b bits, and each key shall be assumed to be a fixed point integer fitting into one word. The instruction set available to the computer will be arithmetic, bitwise logical, and comparison operations on b-bit words. The integer N will denote the size of the database we search. It will be assumed that $b \geq \log_2 N$ since otherwise our main memory search structures would not even have a sufficiently large word length to store the addresses of the N objects that are stored in the computer's main memory bank. The runtime of our algorithm will be viewed as a quantity $F(b, N)$. Usually it has not been done, but most of the algorithms of [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46] can be trivially generalized to common floating point representations of real numbers.

The first surprising article about fast dynamic tree operations was Van Emde Boas's data structure [48], which had an $O(\log\log U)$ worst-case retrieval, insertion, and deletion time for a data structure using $O(\ U \ \log\log\ U\ )$ space to represent a set of elements from the universe $0, 1, 2, \ldots, U$. The space was further compressed by Van Emde Boas to $O(\mathrm{U})$ in [47]. (One can also find a further discussion of the implications of Van Emde Boas trees and their variants in, for instance, [28, 29, 30, 31, 49, 50].) Since $b = \log U$, one can think of Van Emde Boas's algorithm as having an $O(\ \log\ b\ )$ time. Unless b is a very large number, this time is better than the conventional $O(\log N)$ balance tree time.

The first contribution of Fredman and Willard's fusion trees [25] was that they established a sorting time $F(b, N)$ with a worst-case bound $O(N \log N/ \log\log N)$ for an $O(N)$ data space structure (regardless of b's and N's values). This was the first method to obtain an $o(N \log N)$ time for sorting using what Andersson et al. [5] later called a "conservative method" for measuring computing costs. Fusion trees also provided an $O(\log N/ \log\log N)$ worst-case and $O(\sqrt{\log\ N})$ randomized time cost for performing standard balance-tree and search-and-update operations in an $O(N)$ space structure. A cousin of the fusion tree, called the q-heap, can bring dynamic balance-tree search, insert, and delete worst-case times down to an $O(1)$ asymptote provided the relevant set has a PolyLog(N) size and one has access to an $o(N)$-sized lookup table. Q-heaps were introduced in [26] as a vehicle for devising a worst-case linear time algorithm for the minimum spanning tree problem.

One open question raised by Fredman and Willard's original paper [25] was whether or not one could achieve an $O(\sqrt{\log\ N})$ worst-case time cost for performing standard balance-tree search-and-update operations in an $O(N)$ space structure. Such a question was naturally pressing because Fredman and Willard had indeed achieved an $O(\sqrt{\log\ N})$ worst-case time in the three cases where a static data structure had access to $O(N)$ space, a dynamic data structure was allowed more space, or where the environment was dynamic, the space was $O(N)$ but the time was now *randomized.* The final design of this revised and more idealized version of a fusion tree was done recently by Andersson [4]. He showed that $O(\sqrt{\log\ N})$ amortized and also that implicitly[1] worst-case times for performing standard balance-tree search-and-update operations were possible for strictly $O(N)$ space structures. His article explicitly uses all the functionality of the original fusion trees and adds to these a new memory compression method to achieve the $O(N)$ space.

Another open question raised by the discussion on fusion trees in [25] was whether the cost of sorting could be improved beyond the $O(N \log N/ \log\log N)$ worst-case and $O(N\sqrt{\log\ N})$ randomized times. This problem has been studied extensively by [1, 4, 5, 40, 44, 46], and the best combined results from this evolving literature are roughly that $O(N)$ space supports either a randomized time $O(N \log\log N)$ or a worst-case time $O[N(\log\log N)^2\ ]$ for sorting. In particular, the randomized problem was resolved by Andersson et al. [5] and Thorup [44], and the best worst-case sorting is due to Thorup [46].

Another recent direction of research has been the study of Dijkstra's SSSP problem. Thorup showed in [45] that it was possible to devise a fully $O(N)$ worst-case algorithm for constructing the SSSP. Most of Thorup's algorithm is unrelated to the prior Fredman–Willard fusion tree research, but he does use a q-heap data structure as one important subcomponent of his final data structure.

There are simply too many new ideas in the rapidly expanding literature [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 36, 37, 40, 41, 43, 44, 45, 46] for us to describe all these results in full detail. Albers and Hagerup [1] have discussed parallel analogues of fusion tree sorting. (An important aspect of their work is that some of its ideas are essential for implementing the *nonparallel sorting* algorithms of [5].) Many lower bounds relevant to fusion-like data structures have been developed by Andersson et al. [6], Brodnik, Mitlersen, and Munro [15], Beame and Fich [9], Ben-Amram and Galil [11], Miltersen [36, 37]. Andersson [3], and Andersson, Miltersen, and Thorup showed in [7] that the Fredman–Willard fusion tree results could be extended to a data model using exclusively AC^0 instructions. Raman [40] and Thorup [43] have developed some interesting new priority queue data structures, which ultimately led Thorup to announce a linear algorithm [45] for Dijkstra's SSSP problem and a worst-case $O[N(\log\log N)^2]$ algorithm [46] for sorting.

It should be noted that while many of these articles improve upon the prior Fredman–Willard results with better upper bounds, a large number of these papers (such as, for example, [4, 45]) mention using particular parts of the Fredman–Willard data structure as working subroutines of their yet more refined algorithms. In essence what we will do in sections 4 through 8 of this paper is also employ such subcomponents of fusion trees to examine six other algorithmic paradigms.

It also should be mentioned that it is not entirely true that all aspects of research related to fusion trees are fully divorced from practical application. For instance, the discussion of AC^0 circuits by Andersson [3] and Andersson, Miltersen, and Thorup [7] could result in practical hard-wired algorithms. Also some of the engineering-grade sorting algorithms of Bentley and McIlroy [13] and P. McIlroy, Bostic, and M. McIlroy [32] are partly related to fusion tree research in that one part of their procedures breaks a larger problem into locally smaller problems with tiny constants.

[1]The relevant "worst-case" cost algorithm did not actually appear in Andersson's paper [4], but we think it should be fairly credited to Andersson because it is an easy extension of his amortization techniques combined with standard methods for converting an amortized optimizing algorithm into a worst-case control procedure. For instance, one could use techniques roughly similar to either our proof of Lemma 3.3 (see section 10) or methodologies of [52, 53] to easily transform Andersson's amortized optimizing algorithm into a worst-case controlling procedure.

It also should be mentioned that some of the older subcomponent data structures employed by the original fusion trees, such as perfect hashing, Van Emde Boas tree and fast tries [22, 24, 42, 47, 48, 49, 50] can be potentially pragmatic when used in a context *perhaps different from* that utilized by the fusion tree.

As we noted in section 1, our goal in the present article will be to consider six sample problems, which illustrate some types of theoretically feasible (although not necessarily practical) improvements that can result when the Fredman–Willard q-heap data structures are applied in various settings. In order to simplify the presentation, we will not require the reader to examine [26] or any other article that was mentioned in the preceding paragraphs of this literature survey. All the reader is required to know is that there is a *black box software package,* called the q-heap, that was proven in [26] to have the following characteristics.

THEOREM 2.1. *Suppose S is a subset of cardinality $M < \sqrt[5]{\log N}$ lying in a larger database consisting of N elements. Then there exists a q-heap data structure for representing S such that the q-heap uses $O(M)$ space and enables insertions, deletions, member, and predecessor queries into the subset S to run in constant worst-case time, provided access is available to a precomputed lookup table of size $o(N)$.*

The lookup table for q-heaps can be constructed in $O(N^c)$ time (for some small constant $c < 1$). The $o(N)$ space and preprocessing costs accrued to such lookup tables are an acceptably small and minor expense in most computing applications because they will have a large number of different q-heaps share access to one common search table. In other words, a large number of different minisets $S_1, S_2, \ldots, S_j$ will typically each have different q-heap data structures $Q(S_1)$, $Q(S_2), \ldots, Q(S_j)$, but they will share use of the very expensive common lookup table. This notion of sharing an expensive lookup table, jointly developed by Fredman and us in the context of two slightly different lookup schemes [25, 26], is perhaps the main intuitive reason one can improve the performance of such a large number of different computing tasks (using either the fusion trees or much of the subsequent literature).

We emphasize that this article has been organized so that the reader does not need to know what algorithm is actually contained in the "black box" of software of Theorem 2.1. It will matter not whether the software inside this curious black box is manufactured by some corporate giant, such as IBM, Microsoft, or NetScape, or by some financially stumbling computer company, called perhaps the NanoSoft Corporation$^{\text{TM}}$. Regardless of such details, our improvements over the prior literature's six sample problems will be easily comprehensible to the reader, as a theoretical family of algorithms, without knowledge of what lies inside the curious black box, inserted by the engineering staff of NanoSoft.

Indeed if I may drop the mild humor from the preceding paragraph, it makes a serious point. It is that one does not need to understand the details of the fusion procedures of [25, 26] to grasp the nature of our six sample problems. Thus, it is sufficient to view Theorem 2.1 as a black box of software. Our six sample problems are intended to motivate curiosity into the subject matter of [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46] by illustrating the breadth of the improvements they make feasible.

Moreover, despite the perhaps impractical coefficients associated with most fusion-like algorithms, one enticing aspect of this subject is the long list of potential problems which can have some facet of their capacity undergo at least a theoretical mathematical improvement. The six sample problems examined in sections 4 through 9 of this paper and Thorup's recently announced linear solution to Dijkstra's SSSP problem

[45] are probably only a small sample of what can actually be done. For instance, the entire literature on fusion trees [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46] seems to be relevant to lower bound theory because it shows that certain conjectured mathematical lower bounds, that had once looked very plausible, are in fact theoretically fallacious under some quite natural models of computation.

3. Corollaries to Theorem 2.1. This section will discuss two simple corollaries to Theorem 2.1, which will illustrate the main format in which we will apply Theorem 2.1. First we will introduce one useful lemma.

Lemma 3.1. *For simplicity, let us assume that $B > 16$. Consider a B-tree whose internal nodes have arity between $B/8$ and B, whose root has arity between 2 and B, and whose leaves store the data and each have the same depth. Suppose that searches, insertions, and deletions in such a tree of height h will have an $O(h)$ cost when no splits or merges occur, and the costs of splitting a node or merging two nodes is bounded by $O(B)$. Then regardless of the details of the structure of the node v, it is possible to devise an insertion and deletion algorithm for this tree that runs in amortized time $O(h)$, with the $O(h)$ asymptote using a constant that is independent of B.*

Our main interest in Lemma 3.1 will be when the q-heaps of Theorem 2.1 are the main organizing method for v's internal structure.

Proof. We will only briefly sketch the proof of Lemma 3.1. Consider the natural B-tree insertion/deletion algorithm that merges a nonroot internal node v with its sibling w if v's arity is less than $B/8$, that splits a node into two equal halves whenever its arity exceeds B, and that makes the child of a tree root into the new root if the preceding operations caused the old root to have only one child. (Sometimes a merge will immediately trigger a split into two equal-size halves because a node becomes too large after the merge operation.) It is easy to devise an accounting function that shows there will be only an amortized number of $O(1/\mathrm{B})$ splits and merges in a tree of height h. (This is essentially because nodes of height j will have an amortized frequency of $O[\ (8/B)^{j+1}\ ]$ of splitting and merging.) Hence the splits and merges will have an $O(1)$ amortized cost. This shows that the total cost of insertions and deletions is $O(h)$, since splitting and merging are the only costly operations outside a general $O(h)$ searching cost. □

Lemma 3.1 also holds if splits and merges have a cost proportional to the number of leaf descendants of a node. (Multibranching B-trees with such properties were presented in [51].) However, such trees are not relevant to our present discussion.

We will frequently employ versions of the B-tree structure of Lemma 3.1 where the branching factor $B = \sqrt[5]{\log N}$ and where a q-heap is used to format the structure of the individual internal nodes. This tree will have a height, search time, and update time proportional to $1 + \log M / \log\log N$ when it stores M elements. The term q*-heap will refer to such a modified B-tree form of q-heap. From the combination of Lemma 3.1 and Theorem 2.1, we thus have the following corollary.

Corollary 3.2. *Assume that in a database of N elements, we have available the use of precomputed tables of size $o(N)$. Then for sets of arbitrary cardinality $M \leq N$, it is possible to have available variants of q*-heaps using $O(M)$ space that have a worst-case time $O\{\ 1 + \log M\ /\ \log\log N\}$ for doing member, predecessor, and rank searches, and that support an amortized time $O\{\ 1 + \log M\ /\ \log\log N\}$ for insertions and deletions.*

The proof of Corollary 3.2 is an immediate consequence of Lemma 3.1 and Theorem 2.1 because the data structure of Corollary 3.2 simply consists of a B-tree storing M records whose internal nodes are q-heaps, and which has a branching factor

$B = \sqrt[5]{\log N}$.

LEMMA 3.3. *Consider again a B-tree data structure that satisfies the hypothesis of Lemma* 3.1. *Then it is possible to develop a more elaborate version of the insertion and deletion algorithms of Lemma* 3.1 *so that the insertion or deletion of a leaf-record has an $O(h)$ worst-case (rather than amortized) cost.*

The previous literature has illustrated many examples of amortized optimization algorithms which can also guarantee worst-case time, if their procedures are made somewhat more elaborate. A similar type of proof of Lemma 3.3 is sketched in the appendix (section 10). The reason we postpone the proof of Lemma 3.3 until the appendix is that its worst-case control procedure has a poor coefficient, and the techniques needed to transform the amortized time-optimizing algorithm of Lemma 3.1 into a worst-case controlled procedure is very similar to what often appears in the previous literature.

COROLLARY 3.4. *The data structure in Corollary* 3.2 *can be improved so that its predecessor-query, member-query, insertion, and deletion operations all have a worst-case complexity* $O\{ 1 + \log M / \log\log N\}$.

Once again, no proof is needed to verify Corollary 3.4. It is an immediate consequence of Lemma 3.3 and Theorem 2.1 because its data structure simply consists of a B-tree storing M records whose internal nodes are q-heaps and which has a branching factor $B = \sqrt[5]{\log N}$. We will use the term q*-heap to refer to the data structure of either Corollaries 3.2 or 3.4 because they are essentially the same concept, except that one uses an amortized-optimizing algorithm and the other employs a slightly more elaborate worst-case control. We close this section by emphasizing that the q*-heaps of Corollaries 3.2 and 3.4 are related to q-heaps and AF-heaps in [26]. The latter data structures, originating in our joint paper with Fredman, stimulated Corollaries 3.2 and 3.4.

Finally, we point out that there are roughly two types of applications of q*-heaps that are explored in this paper and the previous literature. One type of application is based on using the q*-heap to speed up an algorithm by essentially reducing the height of a tree. This method is feasible because the q*-heap can often allow us to traverse in $O(1)$ time tree nodes which have roughly PolyLog(N) arity. An alternate approach is to forgo making any use of the q*-heap until one has essentially reduced an initial problem of size N to one of roughly size $(\log N)^c$ for some fixed constant $c > 0$, at which time the final processing step can be made to run in $O(1)$ time. In essence, the first method is used by us in sections 5 through 7 of this paper, while the second was used by Fredman and Willard [26] and Thorup [45] and will be again used by us in section 4 of this paper.

4. Hashing.

Most of the sample problems discussed in this article come from computational geometry. However, this article will begin by considering universal hashing because our proposed improvement has a short description that contains one very pretty idea. It will be unnecessary for the reader to examine the prior literature on hashing before examining this section. Our discussion, although only an abbreviated outline, should be sufficiently self-contained for the reader inexperienced with universal hashing to still appreciate the gist.

Dietzfelbinger and Meyer auf der Heide [22] have developed a universal hashing scheme where each insertion, deletion, and lookup will always have a probability exceeding $1 - o(N^{-k})$ of running in constant time (for an arbitrary $k > 0$). This section will illustrate how q*-heaps enable us to develop a quite different alternate structure that provides a better probability $1 - o(N^{-(\log N)^k})$ for these three operations to run in constant time (where $k > 0$ is again an arbitrary constant, and it is assumed again that the universe size $U <$ Polynomial(N)).

The data structure for achieving this functionality rests on two concepts. First, it was noted by Corollary 3.4 that the q*-heap provides a formalism to perform constant time insertions, deletions, and retrievals on any set of PolyLog(N) cardinality. Now, consider a hash table, with N addressable buckets for representing a time-varying set of cardinality $\leq N$, which has no overflow mechanism, and which simply stores in a q*-heap all the records that are mapped into a common bucket address. This hash table will thus assure that a bucket can be searched in $O(1)$ time provided it stores no more than PolyLog(N) elements.

The pleasing point is that the Poisson probability distribution will assign each bucket a probability less than $o(N^{-(\log N)^{k+1}})$ for containing more than $(\log N)^{k+2}$ elements. Thus, there will be only this tiny probability that a single one of our N buckets will store no more than $(\log N)^{k+2}$ elements. Thus for an arbitrary constant k, there is the same probability greater than $1 - o(N^{-(\log N)^k})$ that a constant time bound on searches, insertions, and deletions will hold *within all N buckets simultaneously!*

Another pleasing point is that by applying some theorems of Siegel [42], we can strengthen in a straightforward manner the Poisson probability analysis (above) into a formal theorem about classes of universal hash functions. To do so, we assume that there is a prespecified constant N bounding the maximal size of the time-varying set stored in the hash table and that the universe size satisfies $U \leq$ Polynomial(N). Then Siegel's universal hash functions [42] will require only N^ϵ words to form a class satisfying his $(\log N)^k$-wise independence property, for any constant k. This fact immediately implies that the Poisson distribution is a sufficiently accurate probability predictor for this *universal hash class* to imply the same claimed $1 - o(N^{-\text{PolyLog}(N)})$ probability that each search and update operation will run in constant time. (We omit the further details because they are basically a routine hybridized application of Siegel's quite sophisticated formalism [42] in the context of the data structure outlined in the prior paragraph of this section. The basic point is thus that *universal hash classes* are assured by Siegel's analysis to operate in the same manner as the simpler analysis of randomized Poisson probability distributions from the prior paragraph of our discussion.)

The three-paragraph passage above is obviously more complicated than it appears because it cannot be fully formalized without duplicating both Siegel's full formalism [42] and the full proof [26] that a q-heap data structure satisfies the black box properties of Theorem 2.1. The reason the preceding discussion is significant is that it uses essentially the same computing model as Dietzfelbinger and Meyer auf der Heide [22] had used for their universal hashing scheme. Our probability of constant time operations is an $1 - o(N^{-\text{PolyLog}(N)})$ magnitude, which is better than the probability of the form found in [22] $1 - o(N^{-k})$.

5. Priority-search trees. This section will examine McCreight's priority-search tree in considerable detail. This search problem will offer an excellent case study illustrating how a data structure can undergo theoretical (although possibly impractical) improvements when q-heaps are used to streamline it. McCreight's priority-search trees are described by both his journal article [35] and by Mehlhorn's textbook [34]. We will assume that the reader has examined at least one of these sources when we present our $\log\log N$ improvements. The second half of this section will also explain how to produce similar theoretical (but again not practical) improvements upon one of Chazelle's orthogonal range query structures in [16].

Given a set S of N points on the xy-plane, define Query (a, b, c) to be a request for the subset of S that satisfies

$$a < x < b \;\wedge\; y > c. \tag{5.1}$$

The McCreight priority-search trees can perform this operation in $O(\log N + K)$ time (where K is the size of the output). It supports $\log N$ insert and delete operations and uses $O(N)$ space.

The fusion priority-search tree described here will be a variant of the B-tree, with a branching factor B essentially storing information about some subset of points whose x-value lies in some line segment $[A_v,\ B_v)$, whose "range" is implicitly defined by the B-tree's stored keys. This line segment will be denoted as Range(v), and Set(v) will denote the subset of points from S whose x-coordinate lies in this interval. (This notation thus implies that if $w_1, w_2, \ldots, w_n$ are the children of v then the union of Range(w_1), Range$(w_2), \ldots,$Range(w_n) will equal Range(v), and similarly the union of Set(w_1), Set$(w_2), \ldots,$Set(w_n) will equal Set(v).)

Following McCreight's example [35], each node v will store the particular ordered pair (x, y) from SET(v) which has maximal y-value but is not stored in any ancestor of v. This ordered pair is denoted as MAX(v). The unique aspect of the fusion priority-search tree is that it will contain the following three additional fields:

1. KEYS(v): This will be a q*-heap describing the keys separating the ranges of v's children.
2. TOPS(v): This will be a second q*-heap that stores the y-coordinates of the MAX elements belonging to v's children.
3. FUSEDTOPS(v): Let $y_1,\ y_2, \ldots, y_B$ denote the distinct y-values stored in TOPS(v) and r_i denote the rank of y_i in this subset. Then FUSEDTOPS(v) is the ordered tuple $(r_1,\ r_2, \ldots, r_B)$. Since FUSEDTOPS(v) can be encoded in $B\ \log\ B\ \leq\ O(\sqrt{\log\ N}\cdot\ \log\ \log\ N)$ bits, it can fit into one word of memory. (Recall that section 2 indicated that we always employ words which have a bit length $\geq \log N$.)

Some notation is helpful to describe the search algorithm for the fusion priority-search tree. Given the ordered triple (a, b, c) associated with the query (5.1), we say a node v is a

(i) *left borderline* node if Range(v) intersects a but not b.
(ii) *right borderline* node if Range(v) intersects b but not a.
(iii) *double borderline* node if Range(v) intersects both a and b.
(iv) *subsumed* node if Range(v) is contained within the interval (a, b)
(v) *pivotal* node if v is both subsumed and a child of a double borderline node.

Since a fusion priority-search tree will have a $O(\log\ N/\log\ \log\ N)$ height and a $\sqrt{\log\ N}$ branching factor, it will have no more than $O(\log\ N/\log\ \log\ N)$ borderline and pivotal nodes. (This is because there can never be more than $\sqrt{\log\ N}$ pivotal nodes in a tree with $\sqrt{\log\ N}$ branching factor.) All these pivotal and borderline nodes can be found in $O(\log\ N/\log\ \log\ N)$ time by a trivial application of the "q*-heap" search procedure of Corollary 3.2 (see footnote[2] for the formal details). The first step of our retrieval algorithm will consist of such a search for the borderline and pivotal nodes. The second step of the algorithm of query (5.1) will examine the MAX(v) elements of the nodes just visited to check whether they satisfy the query. It will output those elements which do.

Each remaining element in S that satisfies query (5.1) will be a MAX(v) element for some subsumed node v. Since we desire to make our algorithm run in time $O(\ \log N\ /\ \log\log N\ \ + K)$ (where K is the size of the output), it is necessary to make the last step of our search find these elements in time proportional to $O(\ \log N\ /\ \log\log N\ + K)$. (This is quite a tight constraint that we are required to satisfy. It will require some care to achieve this objective.)

To obtain this run time, the third step of our search will repeatedly invoke a subroutine called LOOKAHEAD. Upon visiting a node v, this procedure will determine in constant time precisely which of v's children w are *subsumed* nodes that *simultaneously* store an element in their MAX(w) fields that satisfies the query (5.1) *before it actually visits these elements!* (It is *extremely* delicate and tricky to do this search in precisely $O(1)$ time because query (5.1) is a query with three inequality constraints rather than the usual two inequalities associated with a conventional one-dimensional range query condition. Since the tree node v will have $\sqrt{\log\ N}$ children, the procedure LOOKAHEAD must make strong use of the q*-heap property to determine *in constant time* which of these $\sqrt{\log\ N}$ children contain data of interest *before they are even visited.)* The LOOKAHEAD procedure is easiest to describe if we let

(i) J_1 denote an integer indicating how many of v's children are subsumed,

(ii) J_2 denote an integer indicating how many y-values in TOPS(v) are greater than c,

(iii) J_3 denote an integer that equals zero if v is a left borderline node, one if it is a right borderline node, and two if it is subsumed.

Each of these three integers can be calculated in constant time, since J_1 can be discovered during a q*-heap search of the KEYS(v) field, J_2 discovered by a similar search through TOPS(v), and J_3's value will be known by the algorithm as soon as it enters the node v.

Now consider the 4-tuple $J^*\ =\ (J_1,\ J_2,\ J_3,\ \text{FUSEDTOPS}(v))$. This tuple can be encoded in $O\ (\sqrt{\log\ N}\ \cdot\ \log\ \log\ N)$ bits. Consequently, we can store all the possible values for this tuple in a precomputed lookup table of size $o(N)$, where each table entry shall essentially encode a list of which subsumed children w of v have MAX(w) data elements satisfying query (5.1). (More precisely, the particular entry in our table that is indexed by the 4-tuple $J^*\ =\ (J_1,\ J_2,\ J_3,\ \text{FUSEDTOPS}(v))$ can be thought of as containing a list of integers, $I_1,\ I_2,\ldots,I_t$, such that the next nodes that should be visited by our top-down search are v's I_1th, I_2th$,\ldots,I_t$th children.) Since the lookup table can be built in $o(N)$ preprocessing time and occupies $o(N)$ space, its presence does not increase the data structure's overall memory space and

[2]The key idea behind this fast search procedure is that the q*-heap property allows us to trivially organize a node v's data structure so that only $O(1)$ time is needed to find the children of v that are borderline when v is a borderline node. Therefore by a trivial repeated iteration of this functionality over the $O(\log N/\log\log N)$ different height levels, we can trivially find all the $O(\log N/\log\log N)$ borderline nodes in $O(\log N/\log\log N)$ time. Moreover, since the fusion priority-search tree has a $\sqrt{\log\ N}$ branching factor and since all its pivotal nodes will be the children of that particular double-borderline node which has the greatest depth, the same procedure will obviously only need $\sqrt{\log\ N}$ time to find the $\sqrt{\log\ N}$ or fewer pivotal nodes that exist. Hence, we have displayed an $O(\log N/\log\log N)$ time procedure for finding all the pivotal and borderline nodes.

preprocessing time. But yet it enables LOOKAHEAD to determine in $O(1)$ time which of v's $O(\sqrt{\log N})$ children are subsumed nodes having MAX-values satisfying the query even before the time these elements are actually visited!

The key point about the preceding paragraph's lookup table is that its $o(N)$ space and preprocessing costs are insignificant quantities because all the nodes in the fusion priority-search tree will *use one common lookup table,* rendering its cost an acceptably small and trivial burden.

Define EXPLORE(p) to be a three-part procedure that first visits the node p, then invokes LOOKAHEAD to discover which children q of p are subsumed nodes with Max(q) satisfying (5.1), and finally recursively calls itself to explore these children. The final step of the search algorithm of query (5.1) will invoke EXPLORE to probe the subtrees descending from the pivotal and the left and right borderline nodes. Each subsumed node whose MAX(v) element satisfies (5.1) will lie in one such subtree, and all its ancestors in this subtree will also have their Max elements satisfy (5.1). These two conditions guarantee that EXPLORE will correctly find all the nodes whose Max elements satisfy (5.1). Moreover, EXPLORE requires only $O(\log N/\log\log N + K)$ time to process the K subsumed nodes it finds because of LOOKAHEAD's $O(1)$ search property, combined with the fact that there are only $O(\log N/\log\log N)$ borderline and $O(\sqrt{\log N})$ pivotal nodes generated by the query (5.1). Hence, we have shown that all three steps of the retrieval algorithm of query (5.1) run in time $O(\log N/\log\log N + K)$.

Finally, we will show how to execute insertions and deletions in the preceding data structure in amortized time $O(\log N/\log\log N)$. We assume $B = \sqrt{\log N}$ and employ a fusion priority-search tree where each node has arity between $B/8$ and B. We will use the algorithm of Lemma 3.1 for rebalancing the B-tree. It is immediate that constant time insertion and deletion operations are feasible in the KEYS(v) and TOPS(v) fields because they are q*-heaps with $O(1)$ height. The similar $O(1)$ algorithm to update FUSEDTOPS(v) consists of a two-step procedure that first searches TOPS(v) to determine in $O(1)$ time the rank of the particular element y_i (that is to be inserted or deleted) and then uses this rank, the integer i, and the old value of FUSEDTOPS(v) to determine FUSEDTOPS(v)'s new value via table lookup. (Once again, we note that the storage and preprocessing costs of the lookup tables are negligible expenses throughout this paper, since they are $o(N)$ costs.)

The natural algorithm for modifying a fusion priority-search tree of height h will perform $O(h)$ search and update operations into the KEYS(v), TOPS(v), and FUSEDTOPS(v) fields (plus some minor bookkeeping work) when an insertion or deletion does not trigger a node split or merge. It will thus consume $O(h)$ time when a split or merge does not occur. Each split and merge will consume $O(B)$ time. This data structure thus satisfies requirements of Lemma 3.1. Since $h < O(\log N/\log\log N)$, its amortized cost for insertions and deletions is thus $O(\log N/\log\log N)$, by Lemma 3.1. (Once again by Lemma 3.3, a strengthened version of this algorithm can also guarantee worst-case insertion and deletion time.)

This paragraph will explain how to use the fusion trees to improve one of Chazelle's data structures. His article [16] devised a data structure that occupies $O(N \log^{d-1} N/\log\log N)$ space and allows d-dimensional orthogonal reporting queries to run in time $O(\log^{d-1} N+K)$, where K is the number of elements reported [16]. The new fusion priority-search trees allow us to revise Chazelle's orthogonal range query data structure so that a theoretically faster (but impractical) $O(\log^{d-1} N/\log\log N+K)$ reporting time prevails in the same memory space. For the case of the dimension

$d = 2$, the new data structure and algorithm will be the same as Chazelle's except for the following four changes:

1. Wherever Chazelle's data structure had employed McCreight's priority-search trees, the new data structure will obviously utilize the new $\log \log N$ faster fusion priority-search trees.
2. The branching factor of the base tree nodes of [16] will be changed from $\log N$ to $\log N/\log\log N$, thereby enabling one to traverse the set of children of a particular node in $\log \log N$ faster time.
3. Wherever an $O(\log N)$ time fractional cascade was previously employed, the new data structure will use a revised fractional cascade that has a fusion tree rather than a binary search tree as its index, thereby speeding up the index search by a $\log \log N$ factor.
4. The $O(\log N)$ time top-down tree walk in the base tree of [16] will be speeded up by a $\log \log N$ factor by having a q*-heap index the $\log N/\log\log N$ children of each node in this tree. (This is possible to do because the base tree has height $O(\log N/\log\log N)$ and every node has an arity $\log N/\log\log N$. Thus, using the q*-heap data structure of Corollary 3.2, each such node can then be traversed in constant time. Therefore, the top-down tree walk will run in time $O(\log N/\log\log N)$.)

It is immediate that this revision of Chazelle's data structure supports two-dimensional orthogonal reporting queries in time $O(\log N/\log\log N \ + \ K)$. The d-dimensional reporting time of $O(\log^{d-1} N/\log\log N \ + \ K)$ in space $O(N \ \log^{d-1} N \ / \ \log \ \log \ N)$ follows from Bentley's method of reducing d-dimensional queries to two-dimensional ones [10]. (All these results are obviously theoretical but not practical modifications of the procedure of [16].)

The McCreight and Chazelle versions of priority-search trees have many uses in computer science [35]. The fusion priority-search trees pertain to all such problems. For example, one can search a set of N rectangles and output all the K intersections among this set in time $O(K \ + N \ \log N/\log \ \log \ N)$.

6. Aggregation queries. Let S denote a set of points in d-dimensional space. Each point-record r shall contain a special field denoted as VALUE(r). Then a d-dimensional aggregation query is defined as a d-tuple $(A_1,\, A_2, \ldots, A_d)$, representing a request to calculate $\sum$ VALUE(r) for those elements satisfying

$$KEY \cdot 1 \ < \ A_1 \ \wedge \ KEY \cdot 2 \ < \ A_2 \ \wedge \ \cdots \ \wedge \ KEY \cdot d \ < \ A_d. \tag{6.1}$$

This section will explain how to perform query (6.1) in time $O\,\{\,(\log \ N/\log \ \log \ N)^{d-1}\,\}$ when the data structure uses space $O(N \ \log^p N)$ with $p \ > \ d-1$. The last paragraph of this section will define a certain sense in which our proposed algorithm can be regarded as optimal.

Our data structure will be a hybridization of fusion trees with the overlapping K-ranges of [12] and fractional cascading [19]. For simplicity, we will restrict our attention to the case of the dimension $d = 2$. The data structure will be a variant of the range tree whose "base" section T will be a tree with $O(\log \ N/\log \ \log \ N)$ height, PolyLog (N) branching factor and which stores the elements of S at the tree's leaf level in order of increasing KEY·1 value. Each internal node v of T will contain B auxiliary fields, denoted as AUX$(v,1)$, AUX$(v,2),\ldots,$AUX(v,B). The field AUX(v,i) will describe the leaves descending from v's left-most i sons arranged in order by increasing KEY·2 value. Let $r_1 \ r_2 \ r_3 \ldots$ be the elements in AUX(v,i), so arranged in order *of increasing* KEY·2 value. Then AUX(v,i) will also store the aggregate quantities SUBTOTAL$(v,i,j) = \sum_{k=1}^{j}$ VALUE(r_k), for each $j \leq$ CARDINALITY(AUX(v,i)).

A two-dimensional orthogonal wedge query (a,b) is a request to retrieve $\sum$ VALUE(r) for the subset of elements from S satisfying

$$(6.2) \qquad KEY \cdot 1 < a \ \wedge \ KEY \cdot 2 < b.$$

It was implicit from the prior literature that the data structure from the previous paragraph could answer such a query by retrieving $O(\log N/\log \log N)$ subtotal counters. However, the difficulty was that there was no previously apparent computation method to find these counters in time $O(\log N/\log \log N)$.

This difficulty will be resolved by assigning a q*-heap to each node v in the base tree. The q*-heaps will enable a search algorithm to traverse each base tree node in constant time and locate in $O(\log N/\log \log N)$ time the $O(\log N/\log\log N)$ auxiliary fields that will need to be probed. The root's auxiliary field will be searched next in $O(\log N/\log \log N)$ time by using a fusion tree to index it. The remaining AUX-fields can be searched in constant time per field if we employ fractional cascading [19] to interconnect these fields. This algorithm easily generalizes to all dimensions d $O\{(\log N/\log\log N)^{d-1}\}$ search algorithm for doing query (6.1) in $O(N \cdot \log^p N)$ space, for any $p > d-1$.

Finally, we wish to define a partial sense in which the algorithms proposed in this section can be regarded as optimal. Let the notational symbol $L_d(N)$ denote the mathematical quantity of $(\log N/\log \log N)^{d-1}$. Chazelle and Yao have studied time-space tradeoffs for aggregate queries using what is called the semigroup model of computation. Their goal has been to determine how many semigroup addition operations are needed to perform a query similar to (6.1) or (6.2). If a d-dimensional data structure occupies $O(N \cdot \log^p N)$ space for any $p > d-1$, they have determined that the asymptote $L_d(N)$ constitutes both an upper and lower bound for the number of required semigroup addition operations. (Essentially, Yao published this result for the dimension $d = 2$ in 1985 [55], and a more recent 1990 *Journal of the ACM* article by Chazelle generalized Yao's result for higher dimensions [17].)

Our computational model for doing the queries (6.1) or (6.2) is similar to that used by Chazelle and Yao, *except that we wish to be more conservative* by charging one unit cost for *both* each semigroup addition operation *as well as* for each other type of standard machine-language computer operation *that is needed to locate these semigroup* aggregate counters. From the earlier work of Chazelle and Yao, we know that this model of computation certainly cannot calculate aggregates in a time better than their lower bound asymptotes of the form $L_d(N)$. The pleasing aspect of the algorithm outlined in this section is that the upper bound on its runtime will match the Chazelle and Yao lower bounds up to a constant factor when we assume the computer's machine language commands include the standard machine-language operations recognized by the literature on fusion trees.

7. Compressed Van Emde Boas trees. Van Emde Boas trees provide a facility to dynamically perform predecessor queries among a set S of N keys chosen from the universe $0, 1, 2, \ldots, U-1$ in worst-case time $O(\log \log U)$. The early variants of this data structure [48] had used $O(U)$ space, and Johnson [28] showed how the same result could also hold in $O(N \cdot U^{\epsilon})$ space. The y-fast tries [49] with their modifications implied by Dietzfelbinger et al. [21] establishes a $\log\log U$ time in $O(N)$ space, but they establish this speedup in an amortized expected rather than a strict worst-case model of time. This section will show that a $\log\log U$ combination of worst-case retrieval, insertion, and deletion times are possible in $O(N)$ space provided that $N \geq U/(\log\ U)^{c\cdot\log\ \log\ U} = U\cdot 2^{-c(\log\ \log\ U)^2}$, for any fixed constant c. Without the use of the q-heap, the same $O(\log\log U)$ time in $O(N)$ space would only be possible when $N \geq U/\text{PolyLog}(U))$.

Let us begin by explaining how Van Emde Boas would derive the result in the sentence above. There are two published variants of Van Emde Boas trees [48, 47]. The first variant [48] provides dynamic worst-case times $O(\log\log U)$ in space $O(U\log\log U)$. For some fixed constant K, the second variant of Van Emde Boas trees [47] divides the initial set S into U/K subsets. Each subset S_i of the larger set S is associated with an integer $i \leq U/K$, and S_i describes the subset of S whose key values lie between $(i-1)K$ and $iK-1$. Let Z denote the set of integers i whose S_i sets are nonempty. Van Emde Boas's second data structure [47] was a two-part structure whose upper fragment uses his earlier data structure to index the set Z and whose lower fragment is a forest of conventional balanced trees, where there is one tree representing each set S_i. Van Emde Boas [47] noted this structure would have $O(\log\log U + \log K)$ worst-case search, insertion, and deletion times, and it would use space $O[\ N + (U\log\log U)/K\ ]$ (where N is the cardinality of the set S). Such constraints allow one to obtain $O[\ \log\log U\ ]$ worst-case times for the basic retrieval and update operations over an $O(N)$ space structure when N satisfies $N \geq U/\text{PolyLog}(U))$ (assuming we set $K = U/N$).

Q*-heaps allow us to improve the result above. In particular the above retrieval and update times can be reduced to $O(\ \log\log U + \log K\ /\ \log\log N)$ balance trees in the preceding data structure. Then the revised Van Emde Boas tree will allow an $O[\ \log\log U\ ]$ worst-case time for the basic retrieval and update operations over an $O(N)$ space structure when N satisfies the much weaker constraint $N^{-c(\log\ \log\ U)^2}$ for any arbitrarily chosen constant c (assuming we again set $K = U/N$).

8. The ECDF calculation. The section will describe a new algorithm to calculate a set of N d-dimensional ECDF statistics [10, 14] in linear space and time $O(N\log^{d-1}N/\log\log N)$. This result is a $\log\log N$ improvement over the Bentley [10] and Bentley–Shamos ECDF algorithm [14]. The last paragraph of this brief section will explain why we conjecture our algorithm is optimal at least for the dimension $d = 2$.

Since our solution to the ECDF problem of [10, 14] will employ a data structure that naturally combines fusion trees with an earlier data structure devised by Dietz [20], we will provide only an abbreviated description of the new ECDF algorithm. Dietz developed an O(U) space data structure for representing any subset of the integers $0, 1, 2, \ldots, U$ which supports $\log U/\log\log U$ time for dynamic rank queries.

Consider a set S of N points in the plane. Let $x(p)$ and $y(p)$ denote the x and y coordinates of a point $p \in S$. Let $r(p)$ denote the rank of $x(p)$ among the set of x-coordinates. The first two steps of the new 2-dimensional ECDF algorithm will sort the set S twice (by x and y coordinate) and calculate the value of $r(p)$ for each $p \in S$. The third step will employ Dietz's data structure, where we now set the universe size $U = N$. It will take the sorted list that enumerates the elements of S *by increasing y-value* and insert the $r(p)$ attributes of these elements *one by one in order by their increasing y-value* into a Dietz data structure (whose search keys are its r-elements). Just before the insertion of $r(p)$, the algorithm will calculate the quantity $e(p)$ that indicates $r(p)$'s rank *relative to the previously inserted* r-elements. The three steps of this algorithm run in $O(N)$ space and $O(N\log N/\log\log N)$ time. The derived quantities $e(p)$ correspond to the 2-dimensional ECDF statistics of [10, 14].

For the case of the higher dimensions $d > 2$, essentially we will use Bentley's multidimensional divide-and-conquer method [10], adapted to Dietz's data structure and fusion trees, in a manner analogous to the preceding algorithm. That is, we will use Bentley's reduction method to reduce a d-dimensional search to a 2-dimensional ECDF search and then apply the faster 2-dimensional ECDF algorithm outlined in the preceding paragraph.

It is unknown whether the preceding algorithm is in *any sense* optimal for the dimension $d \geq 3$. Our algorithmic upper bounds *almost matches* one of Chazelle's lower bounds [18] when the dimension $d = 2$. The preceding sentence used the phrase "almost matches" because the computational model in [18] for batch-oriented lower bounds is similar but not identical to our model of computation for upper bounds. An interesting open question is whether or not Chazelle's lower bound model could be extended to show that the 2-dimensional ECDF algorithm in this section is optimal.

9. Further results and open questions. There are many examples of algorithms which have linear costs except for their use of sorting and searching. Fusion trees and/or the yet faster, more recent, and more refined sorting and searching algorithms of especially Andersson [4], Andersson et al. [5], and Thorup [44, 46] immediately imply speedups for such tasks. Some examples of these speedups include the construction of a convex hull, the testing for the equality, disjointedness, and containment relationships between two sets, the determination of the chromatic number of a permutation graph, etc. In particular, the algorithm of [5, 44] provides an $O(N \log\log N)$ randomized sorting time and the algorithm in [46] achieves a strict $O[N(\log\log N)^2]$ worst-case sorting time applicable for each of the four paradigm applications mentioned in this paragraph.

The main point is not the particular examples explored in the paragraph (above), or indeed any of the examples discussed in this article. Rather it is that there are so many other similar examples and problems that can somehow be theoretically improved with one of the methodologies of [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 25, 26, 36, 37, 40, 41, 43, 44, 45, 46]. It should be stressed that we have focused mostly on a moderately narrow bandwidth of problems, drawn from computational geometry and information retrieval theory because such problems reflect the author's particular expertise and knowledge. If one merely examines Thorup's linear time algorithm for resolving Dijkstra's SSSP problem [45], it is apparent that there will be many other problems which theoretically can be improved with the use of q-heaps and their many modifications.

One interesting open question is whether or not it is feasible to construct a Vornoi diagram in faster than $N \log N$ time. Since a convex hull can be constructed in faster time using especially the new faster variants of the fusion tree sorting algorithm [4, 5, 44, 46], it is natural to inquire whether the same may be somehow true for Vornoi diagram construction. Another interesting question is whether or not the original q-heaps of [26] can somehow have their $O(1)$ performance time characteristics extended beyond the minisets of PolyLlog(N) size, described by Corollaries 3.2 and 3.4.

It is best to close this article by reviewing what we have accomplished, as well as what has certainly *not been done.* Section 1 had deliberately used the phrase "so-called improvements" to stress the fact that our theoretical asymptotic improvements are not accompanied with coefficients of practically small size. On the other hand, for the sheer delight of intellectual exploration, it is curious that so many different problems in theoretical computer science can undergo asymptotic improvements after the application of fusion trees and q-heaps. (There are obviously many more of these counterintuitive upper bounds one can establish when investigating this subject further. Even *when their coefficients are poor*, such upper bounds are helpful in clarifying what types of conjectured lower bounds can or cannot feasibly hold under various different possible models of computation.) Moreover, although not practical, our even-theoretical discussion of universal hashing is significant because dynamic universal hashing is a foundational problem.

10. Appendix. The proof of Lemma 3.3.

The previous literature has illustrated many examples of amortized optimization algorithms which can also guarantee worst-case time, if their procedures are made somewhat more elaborate and complicated. We will use a similar approach here to transform the amortize-optimizing procedure of Lemma 3.1 into an algorithm that also controls its worst-case running time, as required by Lemma 3.3. Our algorithm in many respects will be analogous to [54]. The discussion in this appendix will therefore be brief. It may be helpful if the reader examines [54] at some juncture, if he wishes to see how one should fill in the precise details for the ideas that are intuitively sketched below.

Our B-tree will be quite conventional, in that all the data will be stored at the leaf level of the B-trees and all leaves will have the same depth. Say an internal node v is "safe" if its arity lies between $B/4$ and $3B/4$, and it is "legal" if its arity lies between $B/8$ and B. (Let us remember that Lemmas 3.1 and 3.3 assume that $B > 16$, and that the tree root is allowed to "legally" contain between 2 and B children.) Our insertion/deletion algorithms will then guarantee that each node has legal arity at all times, and it will attempt to make the arity safe as often as possible.

The following notation will help us describe the insertion and deletion algorithms of Lemma 3.3.

1. An insertion (or deletion) operation will be said to *involve* a node v if it either inserts or deletes a leaf-record L that is either a descendent of v or a descendent of one of the two "adjacent" siblings of v (that lie to its immediate left or right).
2. An *evolutionary merge process* will be a procedure that gradually merges two adjacent B-tree sibling nodes, v and w, into one new node x. We will not allow the merge process to merge the two nodes, v and w, during one single unified action because such an operation would require $O(B)$ time and possibly cause our algorithm to lose its desired worst-case running time $O(h)$. Rather, for some fixed prespecified constant K, each *action* of our merge process will take K new pointers from v's and w's set of children and put copies of these pointers into the new node x that is gradually being constructed. Once the command to "merge" v and w is initiated, one such "action" will be performed by our insertion-deletion algorithm during each insertion and deletion command that "involves" either v or w. Once the new node x is fully constructed, our merge algorithm will deallocate the nodes v and w and make x a new child of the former parent of v and w.
3. An *evolutionary split process* will be the exact reverse of an evolutionary merge process. It will be a procedure that gradually splits one node x into two equal-sized nodes v and w by doing $O(K)$ units of work for each insertion-deletion command that "involves" x. That is, each operation will take K pointers from x and put copies of them into the new evolving nodes v and w.

Our algorithm for inserting and deleting new leaf-records into the B-tree will employ the above three constructs. It will be called Alg(K). It will initiate an evolutionary split process whenever a node's arity becomes unsafely large by exceeding the arity $3B/4$. If a deletion causes a node v's arity to fall below $B/4$ then Alg(K) will do roughly the reverse action of the preceding sentence by essentially activating an evolutionary merge process.[3] Also, all conventional B-tree algorithms, including Alg(K), must be prepared for the possibility that the root r might possibly contain only one child s at the end of some deletion (or insertion) command. In this case, Alg(K) will simply make s the new root of the B-tree and deallocate the node r (similar to conventional B-tree algorithms).

For the sake of brevity, the preceding paragraph's description of Alg(K) was kept very short. The two natural questions a reader will ask about Alg(K) are

1. What is its runtime?
2. Is Alg(K) correct in the sense that it assures the B-tree is always legally balanced?

The punchline will be that it will be trivial to show that Alg(K) always satisfies its claimed $O(h)$ worst-case time bound in a tree of height h, but its correctness will depend on one further delicate observation.

In particular, the answer to question 1 is easy because each insertion or deletion of a leaf-record L in a tree of height h will "involve" no more than $3h$ nodes in the B-tree. (This is because only the ancestors y of L and y's two adjacent siblings will be "involved.") Thus the evolutionary split processes of Alg(K) will execute no more than $3h$ "actions," each of which requires approximately K units of work. Hence, it will consume no more than worst-case time $O(h)$, where the coefficient hidden within the O-notation is proportional to $3K$.

Now let us turn to question 2. Its full answer is complex because Alg(K) will certainly not have the power to assure a B-tree is legally balanced if the prespecified constant K has too small an initial value. However, it is basically trivial to verify that if the prespecified constant K is sufficiently large (i.e., say $K > 100$), then the evolutionary processes used by Alg(K) will be quick enough to repair a temporarily "unsafe" node v and its adjacent siblings (when necessary) to assure that the tree's balance never becomes so much worse as to be "illegal." The latter is all we need to prove Lemma 3.3.

We will not delve into further details or describe the tedious formal encoding of the procedure Alg(K) because many other papers have appeared in the prior literature about how an amortized-optimizing algorithm can be transformed into a worst-case controlling procedure when one manipulates the algorithm's constant coefficient factor K with prudence (see, for example, [54, 52]). Our goal in this abbreviated appendix was only to sketch the intuition behind Lemma 3.3 very briefly. The reader can find much more sophisticated and interesting examples of amortized-to-worst-case transformations in [54].

[3]Deletions are conceptually similar to insertions, but their formal algorithm is painfully more complicated because there are several different subcases. The problem is that one might wish a tiny node v to merge with its sibling w but the latter cannot be done immediately because either w is currently too large or w is in the midst of another evolutionary split or merge that was previously invoked. If w is too large (say its arity is of size larger than $5B/8$), then v will initiate an evolutionary split to break w into two equal-size halves (before v merges with one of the newly produced halves of w). Similarly, if w is previously in the midst of another evolutionary split or merge when v wishes to merge with it, then v's merge must wait until the latter is completed. (The reason our definition of "involvement" made mention of the "adjacent siblings" of v is that a process where an unsafe node v cannot repair itself until its sibling is fixed will work correctly only if we make certain that every insertion or deletion among the leaves descending from v causes K units of evolutionary repair to be done for *both* v and its two adjacent siblings.)

Acknowledgments. I thank both referees for their very helpful comments.

REFERENCES

[1] S. Albers and T. Hagerup, *Improved parallel integer sorting with concurrent writing*, Inform. and Comput., 136 (1997), pp. 25–52.

[2] S. Alstrup, D. Harel, P. Lauridsen, and M. Thorup, *Dominators in Linear Time*, SIAM J. Comput., 28 (1999), pp. 2117–2132.

[3] A. Andersson, *Sublogarithmic searching without multiplication*, J. Comput. System Sci., to appear.

[4] A. Andersson, *Faster deterministic sorting and searching in linear space*, in Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, 1996, IEEE Computer Society, Los Alamitos, CA, 1996, pp. 135–141.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, *Sorting in linear time?*, J. Comput. System Sci., 57 (1998), pp. 74–93.

[6] A. Andersson, P. Miltersen, S. Riis, and M. Thorup, *Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{Logn\ /\ LogLogn})$ is necessary and sufficient*, in Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, 1996, IEEE Computer Society, Los Alamitos, CA, 1996, pp. 441–450.

[7] A. Andersson, P. Miltersen, and M. Thorup, *Fusion Trees Can Be Implemented with AC^0 Instructions Only*, Brics TR 96-30, University of Copenhagen, Copgenhagen, Denmark, 1996.

[8] A. Andersson and K. Swanson, *On the difficulty of range searching*, in Proceedings of 1995 Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, pp. 473–481.

[9] P. Beame and F. Fich, *Optimal bound for the predecessor problem*, in Proceedings of the 31st ACM Symposium on the Theory of Computing, Atlanta, Georgia, 1999, pp. 622–631.

[10] J. Bentley, *Multidimensional divide-and-conquer*, Commun. ACM, 23 (1980), pp. 214–228.

[11] A. Ben-Amram and Z. Galil, *When can we sort in o(NlogN) time?*, in Proceedings of the 34th IEEE Symposium on Foundations of Computer Science, 1993, IEEE Computer Society, Los Alamitos, CA, 1993, pp. 538–546.

[12] J. Bentley and H. Mauer, *Efficient worst-case data structures for range searching*, Acta Inform., 13 (1980), pp. 155–168.

[13] J. Bentley and M. McIlroy, *Engineering a sort function*, Software—Practice and Experience, 23 (1993), pp. 1249–1265.

[14] J. Bentley and M. Shamos, *A problem in multi-variate statistics: Algorithm, data structure and applications*, in 15th Allerton Conf. on Comm., Contr., and Comp., University of Illinois, Champaign, IL, 1977, pp. 193–201.

[15] A. Brodnik, P. Miltersen, and J. I. Munro, *Transdichotomous algorithms without multiplication—some upper and lower bounds*, in Proceedings of WADS-1997, Lecture Notes in Comput. Sci. 1272, Springer-Verlag, Berlin, 1997, pp. 426–439.

[16] B. Chazelle, *Filtering search: A new approach to query-answering*, SIAM J. Comput., 15 (1986), pp. 703–724.

[17] B. Chazelle, *Lower bounds for orthogonal range searching* II—*the arithmetic model*, J. ACM, 37 (1990), pp. 439–463.

[18] B. Chazelle, *Lower bounds for off-line range searching*, Proceedings of the 27th ACM Symposium on Theory of Computing, ACM, New York, 1995, pp. 733–740.

[19] B. Chazelle and L. Guibas, *Fractional cascading: A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[20] P. Dietz, *Optimal algorithms for list indexing and subset rank*, in Proceedings of WADS, Lecture Notes in Comput. Sci. 382, Springer-Verlag, Berlin, 1989, pp. 39–46.

[21] M. Dietzfelbinger, A. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, in Proceedings of the 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1988, pp. 524–531.

[22] M. Dietzfelbinger and F. Meyer auf der Heide, *A new universal class of hash functions and dynamic hashing in real time*, in Automata, Languages and Programming, Lecture Notes in Comput. Sci. 443, Springer-Verlag, New York, 1990, pp. 6–19.

[23] H. Edelsbrunner, *Algorithms in Computational Geometry*, Springer-Verlag, Berlin, 1987.

[24] M. Fredman, J. Komlos, and E. Szemerdi, *Storing a sparse table with $O(1)$ worst-case access time*, J. ACM, 31 (1984) pp. 539–544.

[25] M. Fredman and D. Willard, *Surpassing the information theoretic barrier with fusion trees*, J. Comput. System Sci., 47 (1993) pp. 424–436.
[26] M. Fredman and D. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. System Sci., 48 (1994) pp. 533–551.
[27] G. Gonnet, *Handbook of Algorithms and Data Structures*, Addison–Wesley, Reading, MA, 1983.
[28] D. Johnson, *A priority queue in which initialization and queue operations take $O(log\ log\ D)$ time*, Math. Systems Theory, 15 (1982), pp. 295–309.
[29] R. Karlsson and J. I. Munro, *Proximity on a grid*, in Proceedings of the Second Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, 1985, pp. 187–196.
[30] R. Karlsson, J. I. Munro, and E. Robertson, *The nearest neighbor problem on bounded domains*, in Automata, Languages and Programming, Lecture Notes in Comput. Sci. 194, Springer-Verlag, Berlin, 1985, pp. 318–327.
[31] D. Kirkpatrick and S. Reich, *Upper bounds for sorting integers on random access machines*, Theoret. Comput. Sci., 28 (1984) pp. 263–276.
[32] P. McIlroy, K. Bostic, and M. McIlroy, *Engineering radix sort*, Computer Systems, 6 (1993), pp. 5–27.
[33] K. Mehlhorn, *Data Structures and Algorithms* 1*: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
[34] K. Mehlhorn, *Data Structures and Algorithms* 3*: MultiDimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.
[35] E. McCreight, *Priority search trees*, SIAM J. Comput., 14 (1985), pp. 257–276.
[36] P. Miltlersen, *Lower bounds for union split-split-find related problems*, in Proceedings of 26th ACM Symposium on the Theory of Computing, ACM, New York, 1994, pp. 625–634.
[37] P. Miltersen, *Lower bounds on static dictionaries with bit operations but no multiplication*, in Automata, Languages and Programming, Lecture Notes in Comput. Sci. 1099, Springer-Verlag, Berlin, 1996, pp. 213–225.
[38] W. Paul and J. Simon, *Decision trees and random access machines*, in Proceedings of the Symposium on Logic and Algorithmic, Zurich, 1980, Monogr. Enseign. Math. 30, University of Geneva, Geneva, 1982, pp. 331–340.
[39] F. Preperata and M. Shamos, *Introduction to Computational Geometry*, Springer-Verlag, Berlin, 1985.
[40] R. Raman, *Priority queues: Small monotone and trans-dichotomous*, Proceedings of ESA'96, Lecture Notes in Comput. Sci. 1136, Springer-Verlag, Berlin, 1996, pp. 121–137.
[41] R. Raman, *A Summary of Shortest Path Results*, Technical report 96-13, Kings College, London, 1996.
[42] A. Siegel, *On universal classes of fast high performance hash function, their time-space tradeoff, and their applications*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 20–25.
[43] M. Thorup, *On RAM priority queues*, in Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms, Atlanta, GA, 1996, pp. 59–67.
[44] M. Thorup, *Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift and bit-wise Boolean operations*, in Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, 1997, pp. 352–359.
[45] M. Thorup, *Undirected single source shortest path in linear time*, in Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1997, pp. 12–21.
[46] M. Thorup, *Faster deterministic sorting and priority queues in linear space*, in Proceedings of 9th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1998, pp. 550–555.
[47] P. Van Emde Boas, *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Process. Lett., 6 (1977), pp. 80–82.
[48] P. Van Emde Boas, R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue*, Math. Systems Theory 10, (1977), pp. 99–127.
[49] D. Willard, *Log-logarithmic worst case range queries are possible in space $O(N)$*, Inform. Process. Lett., 17 (1983), pp. 81–89.
[50] D. Willard, *New trie data structures which support very fast search operations*, J. Comput. System Sci., 28 (1984), pp. 379–394.
[51] D. Willard, *Reduced memory space for multi-dimensional search trees*, in Proceedings of the 2nd Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, 1985, pp. 363–374.

[52] D. Willard, *A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst case time*, Inform. and Comput., 97 (1992), pp. 150–204.

[53] D. Willard, *Applications of the fusion tree method to computational geometry and searching*, in Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 386–395.

[54] D. Willard and G. Lueker, *Adding range restriction capability to dynamic data structures*, J. ACM, 32 (1985) pp. 597–619.

[55] A. Yao, *On the complexity of maintaining partial sums*, SIAM J. Comput., 14 (1985), pp. 277–289.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1051–1082

NEAR-OPTIMAL PARALLEL PREFETCHING AND CACHING*

TRACY KIMBREL† AND ANNA R. KARLIN‡

Abstract. Recently there has been a great deal of interest in the operating systems research community in prefetching and caching data from parallel disks, as a technique for enabling serial applications to improve input–output (I/O) performance. In this paper, algorithms are considered for integrated prefetching and caching in a model with a fixed-size cache and any number of backing storage devices (disks). The integration of caching and prefetching with a single disk was previously considered by Cao, Felten, Karlin, and Li. Here, it is shown that the natural extension of their *aggressive* algorithm to the parallel disk case is suboptimal by a factor near the number of disks in the worst case. The main result is a new algorithm, *reverse aggressive*, with near-optimal performance for integrated prefetching and caching in the presence of multiple disks.

Key words. algorithms, prefetching, caching, file systems, operating systems

AMS subject classifications. 68Q25, 68M20, 68P20, 68Q20

PII. S0097539797326976

1. Introduction.

Recent advances in technology have made magnetic disks both cheaper and smaller. As a result, parallel disk arrays have become an attractive means for achieving high performance from storage devices at low cost. Multiple disks offer the advantages of both increased bandwidth and reduced contention. Nonetheless, there are many applications which do not benefit from this I/O parallelism as much as they could, and end up stalling for I/O a significant fraction of the time.

At the same time, it has been observed that many of these applications have largely predictable access patterns. This has enabled the use of prefetching and informed cache replacement (e.g., [11, 21, 28, 29]) as techniques for reducing I/O overhead in such systems. The two techniques are not independent, however, and can interact poorly if their interaction is not considered carefully [6, 28].

In this paper, we consider a theoretical model that captures the important characteristics of a system for prefetching and caching with multiple disks. We study the offline problem of constructing an optimal prefetching and caching schedule in this model for a given stream of requests for blocks of data residing on the disks. An optimal schedule minimizes the elapsed time required to serve the given request stream. Although complete information about future requests is usually not available, partial information *is* often available in the form of limited or even significant lookahead into the request stream. Empirically we have found that a limited-lookahead version of our algorithm outperforms other approaches in practice [17]. In addition, the design and analysis of the optimal offline algorithm is an important step towards understanding and evaluating more practical limited-lookahead algorithms. We can perhaps draw

*Received by the editors September 8, 1997; accepted for publication (in revised form) December 18, 1998; published electronically February 1, 2000. A preliminary and abbreviated version of this paper appeared in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Burlington, VT, 1996, pp. 540–549.
http://www.siam.org/journals/sicomp/29-4/32697.html

†IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 (kimbrel@watson.ibm.com). This work was performed while the author was a student at the University of Washington.

‡Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195 (karlin@cs.washington.edu).

an analogy with the impact of the optimal offline paging algorithm [3] on the design, implementation, and evaluation of online paging algorithms.

Surprisingly, even in the offline, single-disk situation, this is a challenging combinatorial problem. Recently, a polynomial-time exact solution for the single-disk case was found via a linear programming relaxation [2]. The difficulty comes from the fact that prefetching too soon can cause additional cache misses by replacing blocks that would remain in the cache if prefetching were done later or not at all: new and possibly better eviction opportunities arise as a program proceeds. Cao et al. [6] were able to show that a simple, practical algorithm called *aggressive*, which prefetches as early as is reasonable, has performance that is provably close to optimal in the single-disk case.

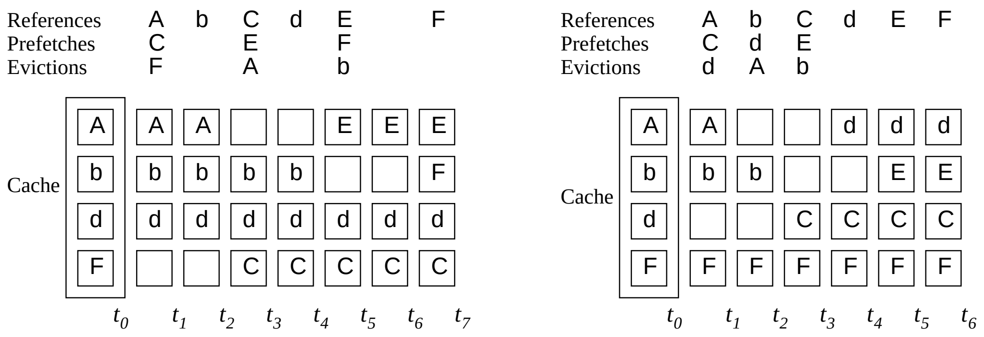
We show, however, that the natural extension of this algorithm to the multiple disk case has performance that is suboptimal by a factor nearly equal to the number of disks. The interaction between caching and prefetching is significantly more complicated in a system with multiple disks because a set of blocks can be prefetched in parallel only if they reside on different disks: each disk can serve only one prefetch at a time. The prefetching schedule and choice of cache evictions impact the potential for subsequent parallel prefetching in a complex way. Our main result is a new algorithm, *reverse aggressive*, with near-optimal performance for this problem.

1.1. An example. An example will serve to introduce our model and illustrate the challenge posed by the multidisk problem. An application program references one block per time unit. If the application wants to reference a block that is not present in the cache, the application must wait or *stall* until the block is present. Each disk can perform only one fetch at a time. If the cache is full, every fetch requires the eviction of some block from the cache. In a real system, it is not known in advance exactly how long a fetch will take (though in our theoretical model, the fetch time is constant); because of this, we assume the evicted block becomes unavailable at the moment the fetch starts. The goal is to minimize the total time spent by the application, or equivalently to minimize the stall time. In the following example, the cache holds four blocks, and it takes two time units to fetch a block from disk.

Suppose the application references blocks according to the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d, and F. Blocks A, C, E, and F reside on one disk, blocks b and d on a different disk. A straightforward approach is to use the *aggressive* algorithm [6]: always fetch the missing block that will be referenced soonest and evict the block whose next reference is furthest in the future, but do not fetch if the evicted block will be referenced before the fetched block.

Figure 1(a) shows the schedule of prefetches, evictions, and block service times produced by this algorithm. For example, initially the first missing block is C, and the block whose next reference is furthest in the future is F. Moreover, the reference to F is after the reference to C. Therefore, the *aggressive* algorithm immediately initiates a fetch for C, evicting F. Notice that this fetch is entirely overlapped with computation (the references to A and b). The schedule produced using this algorithm results in one unit of stall time (the sixth time unit). The entire sequence is served in seven time units.

Figure 1(b) shows another schedule that is faster by one time unit. On the first fetch, d is evicted rather than F, even though d is referenced earlier than F. This has the advantage of offloading one fetch from the heavily loaded disk to the otherwise idle disk. This change allows the fetches of C and d and of d and E to proceed in parallel, thus saving one time unit.



(a) A prefetching and caching schedule. (b) A better schedule.

FIG. 1. *An example of prefetching and caching with two disks. One disk holds blocks A, C, E, and F, and another disk holds blocks b and d. The cache size is $K = 4$ and the fetch time is $F = 2$.*

The example shows that it is helpful to take disk load into account when making fetching and eviction decisions. This is the factor that makes the multidisk problem more difficult than the single-disk problem.

1.2. Overview of results. Our model generalizes the previous example in the obvious way.

- Let D be the number of disks.
- Let B be a set of blocks. We will refer to the disk on which a block $b \in B$ resides as the *color* of b.
- There is a cache that contains at most K blocks in B at any time.
- A *reference sequence*, or *request sequence*, is an ordered sequence of references $R = r_1, r_2, \dots r_{|R|}$, where each $r_i \in B$.
- Fetching a block from a disk into the cache takes F time units.

We imagine that there is a *cursor* which, at any time during the servicing of the request sequence, points to the next request to be served. If this request is for a block that is in the cache, the cursor advances by one during the next time unit. If this request is for a block that is not in the cache, the cursor *stalls* until that block arrives in the cache (i.e., until the fetch for that block completes). Note that to the extent that the cursor is advancing, a prefetch can overlap the serving of requests. Also, prefetches can overlap each other provided that the prefetched blocks reside on different disks. We assume that each block resides on only a single disk.

The goal is to determine a schedule of prefetches and evictions such that the time required to serve the entire sequence is minimized. Since it requires one unit of time to serve each request, the elapsed time is equal to the length of the request sequence plus the total number of steps during which the cursor stalls.

We consider three algorithms for parallel prefetching in this paper: *conservative*, *aggressive*, and *reverse aggressive*. The first two are natural extensions of the two single-disk prefetching strategies described in [6]. They lie at opposite ends of the spectrum in terms of the total number of fetches performed: *conservative* performs the minimum possible number of fetches, at the expense of a worse elapsed time in the worst case; *aggressive* prefetches as aggressively as possible without being foolish.

We give nearly tight bounds on the performance of both of these algorithms. Unfortunately, for both of these algorithms there are reference patterns on which their performance is suboptimal by a factor of nearly D, for values of D, F, and K

that are typical in practice.

THEOREM 1. *On any reference string R, the elapsed time of* conservative *with D disks on R is at most $D+1$ times the elapsed time of the optimal prefetching strategy on R.*

This bound is nearly tight for $D \ll F \ll K$: There are arbitrarily long strings on which conservative *requires time $1 + D\frac{K-F}{K}\frac{F}{F+D}$ times the optimal elapsed time.*

THEOREM 2. *On any reference string R, the elapsed time of* aggressive *with D disks on R is at most $D(1+\frac{F+1}{K})$ times the elapsed time of the optimal prefetching strategy on R.*

This bound is nearly tight for $D \ll \sqrt{F}$: There are arbitrarily long strings on which aggressive *requires time $D - \frac{3D(D-1)}{F+3(D-1)}$ times the optimal elapsed time (within an additive constant that depends only on F and K).*

Our main result is the development and analysis of a new algorithm, called *reverse aggressive*, whose performance is provably close to optimal. Interestingly, it achieves this by constructing a prefetching schedule backwards, i.e., by considering the reference sequence in reverse order. For reasons that will be made clear, this causes it to avoid problems encountered by the (forward) *aggressive* algorithm. *Aggressive* suffers from load imbalance and an inability to keep lightly loaded disks from outpacing (prefetching far ahead of) heavily loaded disks. On real systems, DF/K is small,[1] so that the factor $1 + DF/K$ in the following theorem is not much greater than one (hence our claim of "near-optimality").

THEOREM 3. Reverse aggressive *requires at most $1 + DF/K$ times the optimal elapsed time to service any request sequence, plus an additive term DF independent of the length of the sequence.*

This bound is nearly tight for small D: There are arbitrarily long strings on which reverse aggressive *requires $(1 + (F-1)/K)$ times the elapsed time of the optimal prefetching strategy on R.*

1.3. Related work. Our problem is a generalization of (but significantly more complicated than) the classical paging problem. Indeed, one principle for prefetching (the *optimal eviction* rule described in section 2.1) is derived from Belady's optimal *longest forward distance* [3] paging algorithm. As we will see, however, the application of this rule alone is insufficient to guarantee good prefetching performance; the natural algorithm based on it is suboptimal by a factor of nearly $D+1$. (See Theorem 1.)

On the theoretical side, we know of no prior work on the integration of parallel prefetching and caching. There have been some interesting results on the use of data compression for the design of optimal prefetching strategies [20, 33] and work on prefetching strategies for external merging under a probabilistic model of request sequences [25]. However, these studies concentrated only on the problem of determining which blocks to fetch, and did not address the problem of determining which blocks to replace.

Our work builds on recent studies of the sequential (single-disk) version of this problem which showed [6, 5] that it is important to integrate prefetching, caching, and disk scheduling and that a properly integrated strategy can perform much better than a naive strategy, both theoretically and in practice.

In the systems community, caching and prefetching have been known techniques to improve the performance of storage hierarchies for many years [3, 12]. The breadth

[1] F/K is typically less than 0.02, and typical disk-arrays have at most 5 disks. Moreover, technological trends are such that F/K will only get smaller with time.

of application of these techniques has ranged from architecture [31] to database systems [9, 26, 10] to file systems [12, 22, 15, 24, 32, 4, 14, 7, 29] and beyond. A recent trend in this research is to use applications' knowledge about their access patterns to perform more effective caching and prefetching [4, 7, 28, 29].

Our practical motivation for this problem comes from file systems. In this domain, the most common prefetching approach is to perform sequential read-ahead, i.e., to detect when an application accesses a file sequentially and to prefetch the blocks of the files that are so used [12, 22, 23]. The obvious limitation of this approach is that it benefits only applications that make sequential references to large files. Another body of work has been on predicting future access patterns (without hints from the application) even when access patterns are more complicated [11, 32, 26, 10, 14].

Much research on parallel I/O has concentrated on techniques for "striping" and distributing error-correction codes among redundant disk arrays or other devices. These techniques are used to achieve high bandwidth by exploiting parallelism and to tolerate failures [16, 30, 8, 27, 13].

Our work complements these previous efforts. File access prediction (with or without application hints) can be used to provide the inputs to the algorithm described in this paper. Once future accesses are known, our algorithm determines a near-optimal prefetching schedule. Our algorithm achieves near-optimal performance for any given layout of disk blocks (such as striping). Its performance will only improve when a near-optimal layout is used.

Recently, caching and prefetching have also been empirically studied for parallel file systems [11, 21, 28, 29].

Finally, we have performed simulation studies of the performance of the algorithms described in this paper. Companion papers [17, 19] report on this empirical evaluation. A brief summary of the results is given in section 6 of this paper.

1.4. Organization of the paper.

In section 2, we describe several properties that can be assumed of optimal prefetching algorithms. These constrain the problem and by adhering to them, we can ensure that an algorithm's performance is not far from optimal. Also in section 2, we describe the algorithms in greater detail and give intuition on their performance. In section 3, we give overviews of the proofs of the results claimed in section 1.2. Detailed proofs are contained in section 4. In section 5 we consider the time required by the algorithms to determine prefetching and caching schedules. In section 6, we briefly describe our empirical studies of the performance of these algorithms. We conclude with open problems for further research.

2. Properties of optimal prefetching and caching schedules.

After the following definition, we describe several properties that can be assumed of optimal prefetching algorithms.

DEFINITION 4. *At any point in processing the sequence (i.e., for any given cache state and cursor position), a* hole *is a block that is not present in the cache. We will refer to the operation of fetching a disk block as "filling a hole." We will use the term "hole" to refer to both the missing block and its next occurrence in the request sequence; which of these is meant will be clear from the context. If the cache is full, there are K out of $|B|$ blocks in the cache and thus $|B| - K$ holes. After a block is requested for the last time, we consider the corresponding hole in the request sequence to be at position $|R| + 1$, i.e., greater than the index of any request, where R is the request sequence.*

2.1. Prefetching and caching with a single disk. Before proceeding, we review the results of Cao et al. [6] for prefetching and caching in the single-disk case. They described four properties that can be assumed of any optimal strategy in the single-disk case:

1. *optimal fetching*: when fetching, always fetch the missing block that will be referenced soonest;
2. *optimal eviction*: when fetching, always evict the block in the cache whose next reference is furthest in the future;
3. *do no harm*: never evict block A to fetch block B when A's next reference is before B's next reference;
4. *first opportunity*: never evict A to fetch B when the same thing could have been done one time unit earlier.

It is easy to show that any schedule for serving requests and performing fetch-and-evict operations that does not follow these rules can be transformed into one that does, with performance at least as good. The first two rules specify what to fetch and what to evict, once a decision to fetch has been made. The last two rules constrain the times at which a fetch can be initiated. However, these rules do not uniquely determine a prefetching schedule. In particular, they do not specify how to choose between an earlier prefetch with a correspondingly earlier eviction and a later prefetch with a correspondingly later eviction. The former helps prevent stalling on earlier holes, whereas the latter may help prevent the introduction of holes, and hence stalling at a later time.

Nonetheless, these rules do provide a fair amount of guidance in the design of a prefetching algorithm. Cao et al. considered two natural algorithms, *aggressive* and *conservative*, that follow these rules and lie at opposite ends of the spectrum of possibilities. *Aggressive* is the algorithm that initiates a prefetch whenever its disk is ready (i.e., is not in the middle of a prefetch) and the *do no harm* rule allows it. *Conservative* is the algorithm that refuses to fetch until it can evict the same block that would be evicted by the optimal *longest forward distance* [3] algorithm in the classical paging model. That is, *conservative* applies the rule *optimal eviction* as though the prefetch were to be initiated immediately before serving the request to the missing block, then applies the rule *first opportunity* to swap the chosen fetch/eviction pair as early as possible. *Conservative* makes the minimum number of total fetches, but it often declines opportunities to prefetch blocks.

Cao et al. showed that in the single-disk case, *conservative*'s elapsed time on any sequence is at most twice the optimal time, and that *aggressive*'s worst-case elapsed time is at most $\min(1+F/K,\ 2)$ times optimal, where F is the time required to fetch a block and K is the cache size measured in blocks. (They also showed that these bounds are tight.) On real systems, F/K is typically small, so *aggressive* is close to optimal.

2.2. The multidisk case. There is an obvious and natural extension of each of these algorithms to the multidisk case. For *aggressive*, it is the following: Whenever a disk is free, prefetch the first missing block of that disk's color, replacing the block (of any color) whose next reference is furthest in the future among all cached blocks. However, a fetch should be started only if the next access to the evicted block is after that of the block being fetched.

Unfortunately, as we shall see, this algorithm does not enjoy the same performance guarantee in the multidisk case as it achieved in the single-disk case. In fact, the four properties on which it was based in the single-disk case do not hold for optimal strategies in the multidisk case. As a result, it suffers from two problems in the multidisk case that did not exist in the single-disk case:

- The eviction decisions it makes are "color blind": It chooses evictions to make without consideration of the load on the disks. These choices can result in a situation where many of the holes at any time are of the same color, and therefore cannot subsequently be prefetched in parallel. (See Figure 1 for an example of this.)
- *Aggressive* is too aggressive. The result is that it can cause some disks to fetch too far ahead with respect to other disks. These fetches increase the share of the cache occupied by blocks belonging to the lightly loaded disk(s), creating even more holes for the heavily loaded disk(s) to fill.

Therefore, we are motivated to approach the multidisk prefetching problem in a way that will constrain the space of possibilities for the prefetching schedule in the same way that the four rules described above constrain the schedule in the single-disk case.

2.3. Properties of optimal parallel prefetching and caching. It is not hard to show that out of the four rules for optimal prefetching with one disk, only the last (*first opportunity*) holds when there are multiple disks. Finding a rule to replace *optimal fetching* is not much of a problem, however. The "colored" version of the rule can be used, i.e., for each disk *col*, the next block to fetch from *col* is the next missing block in the sequence that is colored *col*. Thus, as in the single-disk case, the question of which block to fetch reduces to the question of when to initiate a prefetch operation; this question needs to be answered for each disk, of course.

Optimal eviction is more troublesome. Suppose there are two disks, colored red and blue. If there are many red blocks missing in the sequence, then it may be that the best choice for eviction is a blue block even though the block whose next request is furthest in the future is red. This is because the relatively lightly-loaded blue disk can better handle the increased burden of another missing block than the red disk can. (See Figure 1.) Given that a blue block is to be evicted, it is true that the best choice is the blue block that is not requested for the longest time. That is, the colored version of this rule holds, but it does not tell us which color block to evict.

Even the seemingly obvious *do no harm* rule can be violated by the optimal prefetching strategy. This is because the loads on the disks can be imbalanced. If there are many red blocks missing from the sequence, say, but no blue blocks missing, it may be advantageous to buy time by evicting a blue block (and completing a fetch of a red block sooner than would be possible otherwise), and then bringing the blue block back into the cache after a request to some red block has been served (so that a new eviction opportunity has arisen).

2.4. Using the reverse sequence. An interesting twist allows us to convert multiple-disk prefetching to a more constrained, and hence easier to solve, problem. In particular, we consider the request sequence in reverse, in a sense we will describe momentarily. We will be able to show that of the four rules, all but one (*optimal eviction*) hold for optimal schedules serving the reverse sequence. Moreover, we will be able to replace this rule by a simple colored variant as we did with the *optimal fetching* rule for the forward sequence; this will be shown in section 4.1.

First, we return to the single-disk case, and observe that any prefetching schedule that serves the reverse sequence $reverse(R)$ in time T can be used to derive a schedule to serve R in time T as follows. If the schedule for serving $reverse(R)$ serves request r_i between times t and $t+1$, the derived schedule for R serves r_i between times $T-t-1$

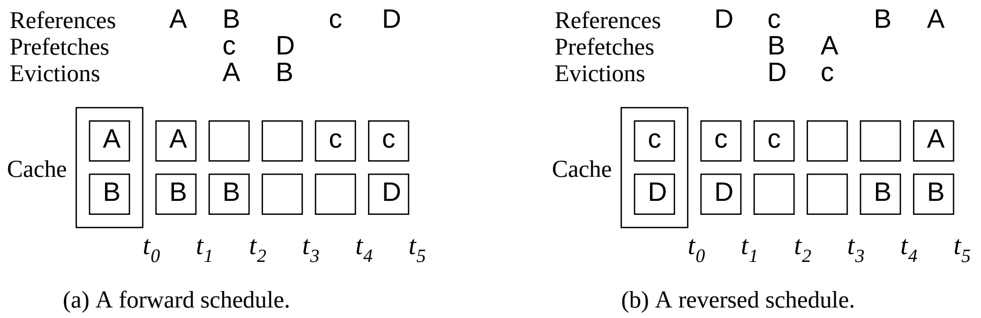


FIG. 2. *An example of reversing a schedule of prefetching and caching with two disks: a disk holding blocks A, B, and D, and another disk holding block c. The cache size is $K = 2$ and the fetch time is $F = 2$.*

and $T - t$. If the reverse schedule replaces a with b between times t and $t + F$, the derived schedule replaces b with a between times $T - t - F$ and $T - t$.[2] Applying this logic twice, we see that the optimal elapsed time for the reverse sequence is the same as the optimal elapsed time for the original sequence.

Reversal of the sequence is more complicated when multiple disks are considered. In the forward direction, the prefetching schedule is constrained to fetch at most one block at a time from each disk; eviction choices may be blocks of either color. Switching between the forward sequence and the reverse sequence, fetches become evictions and vice versa. To derive a useful schedule from a schedule serving the reverse sequence, then, requires that the schedule for the reverse sequence be constrained to *evict* at most one block of each color at a time. This is illustrated in the following example (see Figure 2):

Consider the request sequence "ABcD," where upper case letters denote red blocks and lower case letters denote blue blocks. Let $F = 2$ and $K = 2$. By assumption, at time 0, blocks A and B reside in the cache (for the execution of the sequence in the forward direction). At time 1, a fetch is initiated to bring c into the cache from the blue disk, evicting A. At time 2, a fetch of D from the red disk is initiated, evicting B from the cache. The schedule serves the request sequence in five units of time. See part (a) of Figure 2.

In the schedule for the reverse sequence, at time 1, D is evicted in order to start fetching B. Since c is blue and D is red, a fetch of A (evicting c) can be started at time 2, even though A and B are both red. See part (b) of Figure 2. This schedule can be transformed as described in the previous paragraphs into the valid schedule for the forward sequence of part (a), which is its mirror image.

As previously mentioned, all of the rules presented in section 2.1 except *optimal eviction* can be assumed of optimal prefetching schedules for the reverse sequence. *This fact makes it easier to find a schedule for the reverse sequence, then transform it into one for the original sequence, than to find a schedule for the original sequence directly.* The reason for this is that in the forward direction, any time a block is prefetched a decision must be made as to which color block to evict. In the reverse direction, this decision is made for us: the block to evict is the one not needed for the

[2]We assume that all schedules start with the cache containing the first K distinct requests in the sequence. Alternatively, all our results hold within an additive constant that accounts for differences in algorithms' transient cold-cache startup behaviors. We can assume without loss of generality that all schedules end with the last K distinct requests in the cache.

longest time whose color matches the color of the free disk (i.e., the colored version of the *optimal eviction* rule can be used). One might expect that fetch decisions are harder, but this is not the case. In the forward direction, the missing block to fetch is the one of the right color that is needed soonest. (This is the colored version of *optimal fetching* described earlier.) In the reverse direction, it is the one needed soonest, regardless of color.

2.5. The *reverse aggressive* algorithm. *Reverse aggressive* is a prefetching algorithm that performs aggressive prefetching on the reverse of its input sequence, then derives a schedule to serve the forward sequence as described in section 2.4. That is, on the reverse sequence, it behaves as follows. Whenever a disk is not in the middle of a prefetch, it determines which block in the cache is not needed for the longest time among those with the same color as the disk. If the index of the next request to that block is greater than the index of the first hole (of any color), the block identified for eviction is evicted, and the first hole is prefetched.

An intuitive explanation of *reverse aggressive*'s advantage over (forward) *aggressive* is the following:

- Whereas *aggressive* chooses evictions without considering the relative loads on the disks, *reverse aggressive* greedily evicts to as many disks as possible on the reverse sequence. In the forward direction, this translates to performing a maximal set of fetches in parallel. The fact that these are fetches in the forward direction means that at some point earlier in the sequence, corresponding blocks were evicted. Thus the eviction decisions of *reverse aggressive* on the forward sequence are based on the ability to prefetch the evicted blocks later on in parallel.
- Whereas *aggressive* can wastefully prefetch ahead on some of its disks, *reverse aggressive* is greedy in the reverse direction. Consequently, it is fetching pages in the forward direction just in time (to the extent possible) for them to be used. This results in performing close to the best evictions possible for those fetches, and exploiting parallelism as much as possible without creating load imbalance.

3. Results. In this section we give high-level descriptions of the main ideas used to derive our results. Full details are given in section 4.

3.1. Performance of *conservative* and *aggressive*. The key concept in the upper bound of Theorem 2 is the notion of *domination* from the work on prefetching in the single-disk case [6]. This allows us to bound the cost of *aggressive*'s prefetching schedule in terms of the progress of the optimal schedule at intermediate points during the processing of the request sequence.

DEFINITION 5. *Given two sets A and B of holes with $|A| \leq |B|$, A is said to* dominate *B if for all i, $1 \leq i \leq |A|$, the index of A's ith hole (ordered by increasing index) is no less than the index of B's ith hole. We will say that the ith hole in A is* matched *to the ith hole of B. Notice that domination is transitive.*

Let *opt* denote an optimal algorithm. For intuition, consider the following. If *aggressive*'s cursor is ahead of *opt*'s cursor, *aggressive*'s holes dominate *opt*'s holes, and both are initiating prefetches at the same times, then *opt*'s cursor cannot pass *aggressive*'s: while *aggressive* stalls on a hole, *opt*'s cursor cannot pass its matching hole. We show that *aggressive* is able to continually regain and maintain such an advantage (having its cursor ahead and its holes dominate) over *opt* at regular intervals, without losing too much time to *opt* in the process. *Aggressive* can lose its advantage,

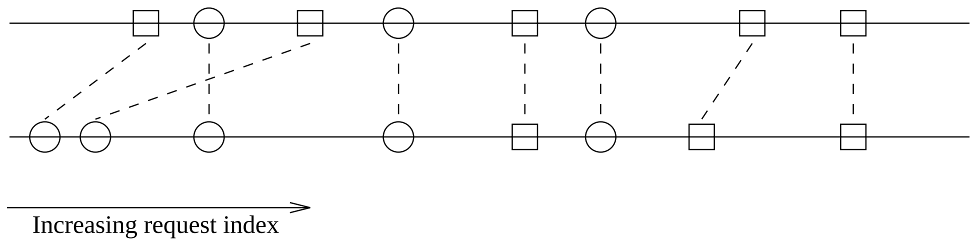


FIG. 3. *Strong domination example: the upper set of holes strongly dominates the lower one. Mismatched shapes represent excess holes. Edges show how strong domination implies ordinary domination. (See Lemma* 11 *and the discussion following.)*

and lose time to *opt*, by prefetching more aggressively than *opt*; this will become clear as the details are presented.

The lower bounds of nearly D in Theorems 1 and 2 come from the fact that an adversary can construct request sequences that cause both *conservative* and *aggressive* to always fetch blocks from only one disk (because they make poor eviction choices). The optimal algorithm *opt* can serve these same sequences at nearly D times the rate because of the parallelism of prefetching on D disks. The additive term of one for *conservative* (in both the upper and lower bounds) comes from *opt*'s ability to overlap prefetches with the serving of requests. In contrast, *conservative* may not be able to do so.

The factor of D in the upper bounds comes from the fact that D is also a limit to the parallelism available to *opt*. As in the single-disk case, the additive term $\frac{(D+1)F}{K}$ in the upper bound for *aggressive* comes from the fact that *aggressive*'s newly created holes are always at least K steps from the cursor. From this, it follows that *aggressive* prefetches too soon (creating extra holes) at most once every K requests.

3.2. Performance of reverse aggressive. The proof of Theorem 3 required several new ideas. The notion of domination from the proof of Theorem 2 is replaced by a stronger notion that we call *strong domination.*

DEFINITION 6. *Let A and B be sets of holes, possibly with different numbers of holes of each color, such that $|A| \leq |B|$. For each color col, let $N_{col}(A)$ (respectively, $N_{col}(B)$) be the number of holes of color col in A (respectively, B). Let $N_{col} = \min(N_{col}(A), N_{col}(B))$. If $N_{col}(A) > N_{col}(B)$, we say that col is an* excess color *of A; if $N_{col}(A) < N_{col}(B)$, col is an excess color of B; if $N_{col}(A) = N_{col}(B)$, col is not an excess color. Let $E_{col} = |N_{col}(A) - N_{col}(B)|$. If col is an excess color of A, we refer to A's first E_{col} holes of color col following the cursor as* excess *holes; excess holes of B are defined similarly. We say the set of holes A* strongly dominates *the set of holes B if*

- *for each col, A's last N_{col} holes of color col dominate B's last N_{col} holes of color col (i.e., A's nonexcess holes of color col dominate B's nonexcess holes of color col, whether col is an excess color of A or B or col is not an excess color), and*
- *all of B's excess holes precede the first hole in A of any color.*

This idea is illustrated in Figure 3, in which holes of different colors are depicted by different shapes.

DEFINITION 7. *For two sets A and B of holes, we say that A strongly dominates B up to index y, if the subset of holes in A that occur at or before index y in the request sequence strongly dominates the subset of holes in B that occur at or before*

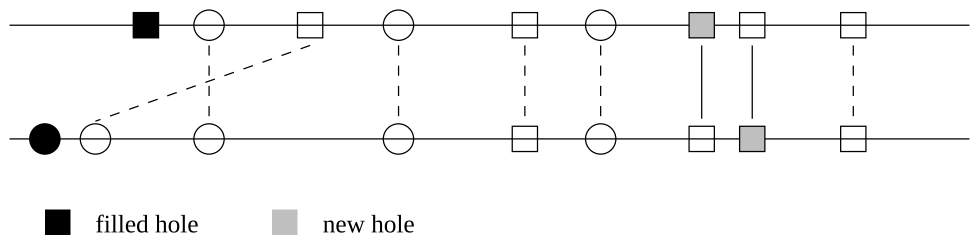


FIG. 4. *Domination lemma: the upper set of holes continues to strongly dominate the lower one.*

index y. When y is the end of the request sequence, we will simply use "strongly dominates" rather than "strongly dominates up to the end of the sequence."

DEFINITION 8. *Let $New(H,(cur,col))$ denote the new set of holes should a prefetch be initiated, if possible (i.e., if allowed by the* do no harm *principle), evicting a block of color col, when the cursor position is cur and the current set of holes is H. Note that $New(H,(cur,col))$ is uniquely determined by the optimal prefetching principles* optimal fetching *and* colored optimal eviction *described in section* 2.4*. If the* do no harm *principle prevents a prefetch, define $New(H,(cur,col)) = H$.*

The following crucial lemma is used to show that if *reverse aggressive* strongly dominates *opt*, and both have the opportunity to initiate a fetch replacing blocks of the same color, then *reverse aggressive* strongly dominates *opt* after the corresponding fetches complete.[3] For purposes of analysis, we consider any blocks that are currently being fetched to be in the cache, i.e., there is no corresponding hole in A or B, even though the corresponding request cannot be served until the F steps are over.

LEMMA 9 (strong domination lemma). *Let A and B be two sets of holes in a request sequence R, and let y, $cur_A < y$, and $cur_B < y$ be indices in R. If A strongly dominates B up to index y, then*

1. *for each color col, if $cur_A \geq cur_B$, $New(A,(cur_A,col))$ strongly dominates $New(B,(cur_B,col))$ up to y;*
2. *for each color col, $New(A,(cur_A,col))$ strongly dominates B up to y;*
3. *for each color col, if $cur_A \geq cur_B$ and every block of color col that is* not *a hole in A is requested after cur_A and before the first hole in A so that $New(A,(cur_A,col)) = A$ (i.e.,* do no harm *prevents a prefetch), A strongly dominates $New(B,(cur_B,col))$ up to y;*
4. *for each pair col_A and col_B of colors, if the best eviction choice of color col_A given the set of holes A and the cursor position cur_A is a block that is not requested between cur_A and y, $New(A,(cur_A,col_A))$ strongly dominates $New(B,(cur_B,col_B))$ up to y.*

Part 1 of Lemma 9 is illustrated in Figure 4.

Note that part 3 of Lemma 9 is a special case of part 1. We prove it separately because it is an important case and because it will aid understanding later, where the lemma is used.

It is not possible to show that *reverse aggressive* strongly dominates *opt* throughout the sequence. Instead, we show that by giving *reverse aggressive* a little more time to serve every subsequence of K requests, it will strongly dominate *opt* at these

[3]We are speaking here of the performance of *reverse aggressive* on the reverse sequence, compared to an optimal schedule for the reverse sequence. However, as described in section 2.4, the optimal elapsed time is the same in both directions, and from *reverse aggressive*'s schedule, we are able to derive a prefetching schedule for the forward sequence with the same elapsed time.

regular intervals. That is, *reverse aggressive* loses about DF steps by prefetching too soon, thereby generating extra holes to fill, only every K requests or so.

The difficulty in showing this is that, in fact, *reverse aggressive* may prefetch prematurely very often, *but with at most $D-1$ disks.* We show that it is able to compensate by consistently making good (distant from the cursor) evictions with the other ("good") disk. While *reverse aggressive* spends an extra F steps relative to *opt* filling the first extra hole created by one of the "bad" disks, the good disk fills one hole. This gives *reverse aggressive* a "one-hole lead" over *opt* with respect to the filling of holes. (Remember, each disk can fetch blocks of any color.) This provides a buffer against stalling on the (further) extra holes created by the bad disks, at least until an extra hole created by the good disk is reached. (The strong domination lemma is used to show that this invariant is maintained.) The good disk creates extra holes only once every K requests.

Formalizing these arguments is difficult; the details are presented in section 4.

4. Proofs.

The following definitions will be useful. Further definitions, specific to the particular proofs in which they are used, will be introduced later.

We divide the request sequence (or, when appropriate, its reverse) into *phases*, maximal-length subsequences of requests to K distinct blocks, as follows. The first phase begins with the first request. Each phase ends immediately before the first request to the $(K+1)$st distinct block since the beginning of the phase, and the next phase begins with that request.

If algorithm *alg* has fetches in progress at any time t, we denote *alg*'s holes before initiating those fetches by $H^-_{alg}(t)$ (i.e., $H^-_{alg}(t)$ contains the holes being filled, but not the ones being created), and *alg*'s holes after those fetches are complete (but ignoring any fetches that haven't begun by time t) by $H^+_{alg}(t)$.

In this section and the next, we assume all algorithms are working with the reverse sequence and denote the optimal algorithm for serving the reverse sequence by *opt*.

Under any algorithm that works on the forward sequence and follows the *optimal eviction* rule, no new holes will be created in a phase once the cursor enters the phase. For every hole in the phase, there is at least one block in the cache that is not requested for the remainder of the phase (since there are only K blocks requested in the phase, by definition, and the cache holds K blocks). In contrast, it is possible that *reverse aggressive* (and *opt* working on the reverse sequence, in fact) will create a new hole within a phase even after its cursor has entered the phase. Although it is true that for every hole in the phase there is a block in the cache that is not requested until after the end of the phase, it may be that all those blocks are the same color and that the best eviction choice of another color is a block that will be requested before the end of the phase. However, if *reverse aggressive* does create new holes in the phase containing the cursor, it will create such holes of at most $D-1$ colors. We refer to the other disk as the *busy* disk for the phase. (If there are two or more such disks, an arbitrary one is chosen.) As long as there are holes remaining in the phase, the busy disk will initiate a fetch to fill one of them every F steps, and will create new holes beyond the end of the current phase.

A fetch using the busy disk (and evicting a block of the same color as the busy disk; the block fetched may be any color) is referred to as a *busy-disk fetch*; fetches using other disks are referred to as *non-busy-disk fetches.*

4.1. Reverse aggressive: Upper bound.

We first give some preliminaries, proving the claims of section 2.4 and a simple lemma on combining subsets of dominating and dominated sets of holes. We next prove the strong domination lemma (Lemma 9).

The strong domination lemma is then used to bound *reverse aggressive*'s elapsed time for a single phase relative to *opt*'s elapsed time. Roughly speaking, if *reverse aggressive*'s holes dominate *opt*'s, *opt* cannot get ahead of *reverse aggressive* since *opt*'s first hole is at least as early in the request sequence as *reverse aggressive*'s. By allowing *reverse aggressive* a small amount of time to correct for mistakes it makes by prefetching sooner than *opt*, strong domination up to the end of the phase is maintained as an invariant until both algorithms reach the end of the phase. This step of the proof is complicated by the fact that the algorithms may fetch blocks using their respective disks in different orders. We must permute one sequence of fetches in order to make direct comparisons between the two algorithms' operations.

Finally, we show that by using a different permutation (and a correspondingly different matching of one algorithm's prefetch operations to the other's), the strong domination lemma implies that strong domination up to the end of the request sequence holds as an invariant as we compare the algorithms' progress from one phase to the next.

LEMMA 10. *Any prefetching schedule for the reverse sequence that does not satisfy the four rules described in section* 2.4 *can be transformed into one that does, with no increase in elapsed time.*

Proof.

1. *Optimal fetching* (fill the first hole): Suppose that at time t_1, a fetch is initiated to fill some hole h_2 other than the first hole h_1. h_1 must be filled before it can be served; say it is filled by a fetch initiated at time $t_2 > t_1$. Since the (later) reference to h_2 cannot be served until after the reference to h_1 is served, the schedule remains valid if h_1 is filled at time t_1 and h_2 at time t_2, and all other operations (prefetches, evictions, and cursor movements) are unchanged. Since we are working with the reverse sequence, this change can be made regardless of the colors of h_1 and h_2. If filling h_1 at time t_1 allows the cursor to advance sooner than it can if h_2 is filled at time t_1, then the eviction opportunities under this schedule are at least as good as those under the original schedule; i.e., the set of holes obtained strongly dominates that obtained under the original schedule. Thus the transformed schedule can be completed to derive a schedule with elapsed time no greater than that of the original.
2. *Colored optimal eviction* (evict the block not needed for the longest time among those colored the same as the free disk): Suppose that at time t_1, block b_1 is evicted, and block b_2 of the same color as b_1 is in the cache and is first referenced after the next reference to b_1. If b_2 is subsequently evicted before the next reference to b_1 is served, the effect is the same if b_2 is evicted first, then b_1. Otherwise, b_1 must be fetched back at some time $t_2 > t_1$ before the reference to it can be served. If b_2 is evicted at time t_1 instead of b_1, it can be fetched back at time t_2. By assumption, there are no intervening references of b_2 on which to stall; thus the transformed schedule stalls no more than the original.
3. *Do no harm* (do not evict b_1 to fetch b_2 if b_1 is needed sooner): Suppose b_1 is evicted to fetch b_2. b_1 must be fetched back before the reference to it can be served; this fetch evicts some other block b_3. Since fetches on any disk can be of any color, the fetch of b_1 can be replaced by a fetch of b_2 (evicting b_3). By

assumption, there are no intervening references of b_2 on which to stall; thus the transformed schedule stalls no more than the original.

4. *First opportunity* (perform each fetch/eviction pair as soon as possible): Suppose that the disk colored *col* is left idle at time t, a fetch of block b_1 is initiated at $t+1$ evicting block b_2 of color *col*, and that the block served at time t is not b_2. Then by initiating the fetch at time t rather than $t+1$, the hole (b_1) is filled one step sooner; certainly, no additional stall is incurred by this change. ∎

We assume without loss of generality that *opt* obeys these rules.

LEMMA 11. *Given two sets of holes $A = A_1 \cup A_2$ and $B = B_1 \cup B_2$ with $|A_1| \leq |B_1|$, $|A_2| \leq |B_2|$, $A_1 \cap A_2 = \emptyset$, and $B_1 \cap B_2 = \emptyset$, if A_1 dominates B_1 and A_2 dominates B_2, then A dominates B.*

Proof. Suppose the contrary. Let i be such that the ith member of A (ordered, as usual, by increasing index in the request sequence) has an index less than the ith member of B. Then A contains i holes with indices less than or equal to that of A's ith hole, and B contains only $i-1$ such holes. But because A_1 dominates B_1 and A_2 dominates B_2, for each member of A there is a distinct member of B with lesser or equal index. Thus we have a contradiction. ∎

Note that Lemma 11 extends to pairs of sets composed of more than two disjoint subsets each. Notice also that by Lemma 11, strong domination implies ordinary, color-blind domination. (Match nonexcess holes according to colors, and all of one set's excess holes to all of the other set's excess holes. See Figure 3 for an example illustrating this.)

LEMMA 12. *Strong domination is transitive.*

Proof. Suppose A strongly dominates B and B strongly dominates C. We show that A strongly dominates C. Fix a color *col*; for convenience (so it can be used as an adjective), suppose *col* is red. Define $N_{col}(\cdot)$ as before. For a collection S of sets of holes, let $N_{col}(S) = \min_{s \in S}(N_{col}(s))$. (We will drop the brackets when listing the members of S.) Let $N_{col} = N_{col}(A, B, C)$. We consider three cases, illustrated in Figure 5.

1. $N_{red} = N_{red}(A)$. A has N_{red} red holes, and these dominate the last N_{red} red holes in B. B's last $N_{red}(B, C)$ red holes dominate C's last $N_{red}(B, C)$ red holes, so B's last N_{red} red holes must dominate C's last N_{red} red holes. Since domination is transitive, A's N_{red} red holes dominate C's last N_{red} red holes. Suppose h is a red hole in C that is excess with respect to A. If h is matched to a red hole h' of B, h' is excess with respect to A and thus precedes A's first hole, so h must precede A's first hole as well. If h is excess with respect to B, it precedes B's first hole, which precedes or is the same as A's first hole, since strong domination implies ordinary domination.
2. $N_{red} = N_{red}(B)$. A's last N_{red} red holes dominate B's N_{red} red holes, which dominate C's last N_{red} red holes. Suppose h is a red hole in C that is excess with respect to B. h must precede B's first hole. h precedes A's first hole as well, since B's first hole precedes or is the same as A's first hole; again, this is because strong domination implies ordinary domination. If h is excess with respect to A, we are done. If h matches some hole h' of A, h surely does not occur after h'.
3. $N_{red} = N_{red}(C)$. A's last $N_{red}(A, B)$ red holes dominate B's last $N_{red}(A, B)$ red holes, so A's last N_{red} red holes must dominate B's last N_{red} red holes, which dominate C's N_{red} red holes. C has no excess red holes with respect

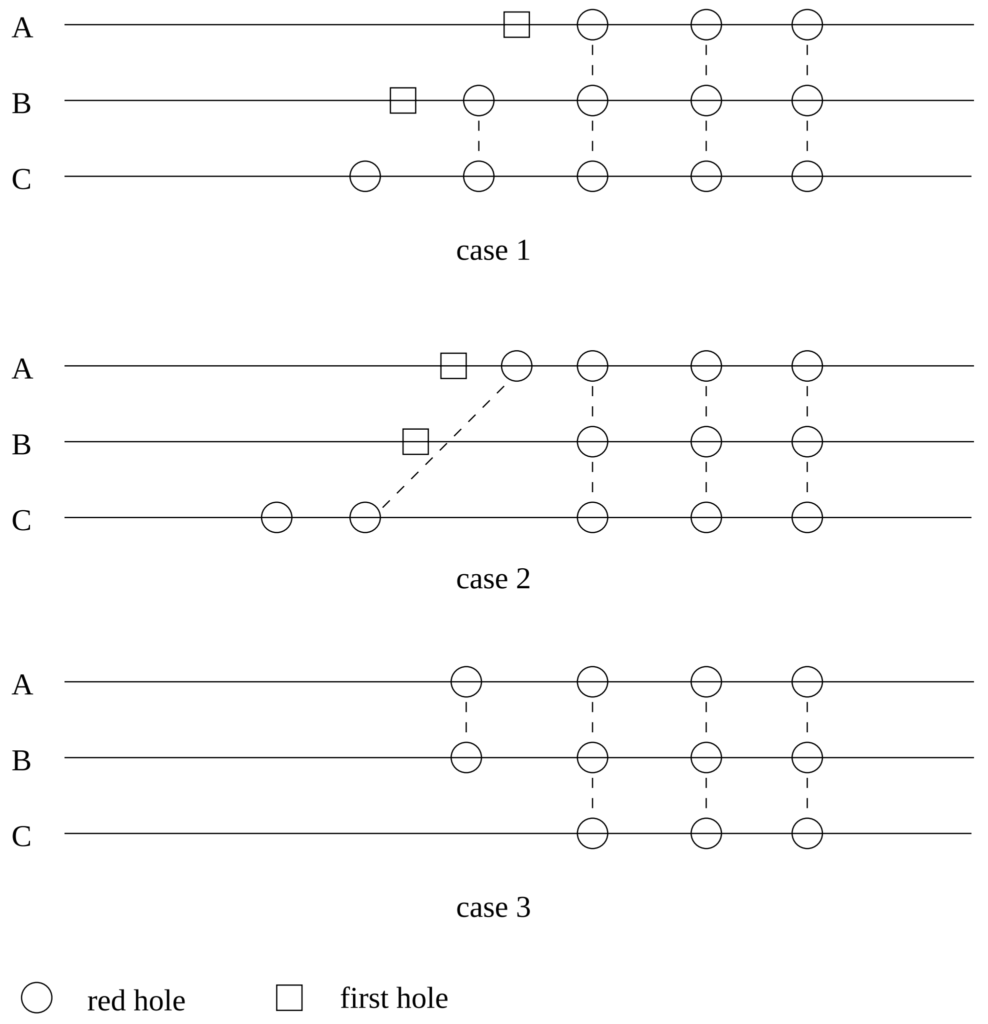


FIG. 5. *Strong domination is transitive.*

to B or A. ▯

We now prove Lemma 9 (the strong domination lemma).

Proof. Define $N_{col}(A)$, $N_{col}(B)$, and N_{col} as before.

We consider the individual changes to A and B in three steps:

1. A's first hole is removed (if necessary, i.e., if $New(A, (cur_A, col)) \neq A$).
2. B's new hole is added to B (if necessary) and A's new hole is added to A (if necessary).
3. B's first hole is removed (if necessary).

We will show that after each step, strong domination of A over B up to y is preserved.

For convenience, we will say that (a hole at) index i is "left" of (a hole at) index j, and (the hole at) j is "right" of (the hole at) i, if $i < j$.

First we prove part 1.

Step 1. *A's first hole is filled.*

Let *col* be the hole's color. First, since A's new first hole is to the right of its old first hole (the one being filled), B's excess holes all are still to the left of A's first hole. If *col* was an excess color of A, we are done. Otherwise, B's hole that was matched to A's filled hole becomes an excess hole, and since it occurred no later than the hole it matched, it is to the left of A's new first hole. Notice that $|A| < |B|$ at this point, in addition to the fact that A strongly dominates B.

Step 2. *Eviction.*

If $cur_A > y$, A's new hole does not affect strong domination up to y, and the addition of a new hole to B (whether left of, right of, or at y) cannot affect strong domination. If $cur_A \leq y$, let A's last N_{col} holes of the same color *col* as the block evicted occur at indices $a_1 < a_2 < \cdots < a_{N_{col}}$, and let B's occur at $b_1 < b_2 < \cdots < b_{N_{col}}$. Since A strongly dominates B, we know that $a_i \geq b_i$ for each i. Let B's new hole be its jth nonexcess hole of color *col*, i.e., the new hole occurs between b_{j-1} and b_j, or at an index greater than $b_{N_{col}}$ in which case $j = N_{col} + 1$, or before b_1 in which case $j = 1$. (As a special case, if *col* is an excess color of B, and the new hole is left of B's last excess hole of color *col*, the new hole becomes an excess hole and the last excess hole takes its place in the following argument.) Let A's new hole be its r^{th} hole of color *col*, with a special case similar to that in the definition of j. Let $a'_1 < a'_2 < \cdots < a'_{N_{col}+1}$ be the indices of A's last $N_{col} + 1$ holes of color *col* after the eviction, and let $b'_1 < b'_2 < \cdots < b'_{N_{col}+1}$ be the indices of B's last $N_{col} + 1$ holes of color *col* after the eviction. Then for $i < r$, $a'_i = a_i$ and for $i > r$, $a'_i = a_{i-1}$; for $i < j$, $b'_i = b_i$ and for $i > j$, $b'_i = b_{i-1}$. To show that domination is preserved, we need to show that $a'_i \geq b'_i$ for each i, $1 \leq i \leq N_{col} + 1$. For $i < \min(r, j)$ and $i > \max(r, j)$ it is immediate that $a'_i \geq b'_i$. If $r > j$, then we have

$$\begin{aligned} a'_r &> a_{r-1} \geq b_{r-1} = b'_r, \\ a'_{r-1} &= a_{r-1} \geq b_{r-1} > b'_{r-1}, \\ &\cdots \\ a'_j &= a_j \geq b_j > b'_j \end{aligned}$$

and we are done. If $r \leq j$, then we must show

$$\begin{aligned} a'_j &\geq b'_j, \\ a'_{j-1} &\geq b'_{j-1}, \\ &\cdots \\ a'_{r+1} &\geq b'_{r+1}, \\ a'_r &\geq b'_r. \end{aligned}$$

Suppose that one or more of these inequalities does not hold, and let i be the largest index for which $a'_i < b'_i$. Then either $i = j = N_{col} + 1$ and $a'_i < b'_i$, or

$$a'_i < b'_i < b'_{i+1} \leq a'_{i+1},$$

where A's new hole at a'_r satisfies $a'_r \leq a'_i$. In either case, there is a block that is not requested until index b'_i that is not a hole in A, and the new hole in A is a block requested earlier at index a'_r instead. But the definition of *New* states that the best possible eviction choice is made, i.e., that the block evicted is the block whose next occurrence is at the greatest index among all blocks of color *col* in the cache. Thus we have a contradiction.

Since the holes of color other than *col* are unaffected by this change, and domination of holes of color *col* is preserved, strong domination is preserved. Also, we still have that $|A| < |B|$.

Step 3. *B's first hole is filled.*

Let *col* be the hole's color. If *col* is an excess color of B, then B will have one fewer excess hole of color *col*; the remaining ones are unchanged, and thus are still to the left of A's first hole. Otherwise, the hole was matched to some hole of A, which becomes an excess hole. The newly excess hole's position is relevant in the definition of strong domination only if it is A's first hole; in this case, since neither A's first hole nor B's excess holes are changed, strong domination is preserved. Because $|A| < |B|$ before this step, we have that $|A| \leq |B|$ afterwards, as needed for strong domination.

The proof of part 1 is complete.

For part 2, step 1 is the same as in the proof of part 1. For step 2, first note that A's new hole is to the right of A's (old) first hole (by the *do no harm* rule), so that B's excess holes still precede all of A's holes. Let *col* be the color of A's new hole. If *col* is an excess color of B, an argument similar to the one above for part 1 shows that A's holes of color *col* will dominate B's nonexcess holes of the same color. If *col* is not an excess color of B, the new hole or some previous hole of A will become an excess hole. In the former case, A's last N_{col} holes are unchanged. In the latter case, the index of A's ith nonexcess hole of color *col* is the same as or greater than before, for each $i \leq N_{col}$. No changes are made in step 3.

For part 3 nothing happens in step 1. Let *col* be the color of B's new hole. Again, for step 2, an argument similar to that for part 1 shows that A's nonexcess holes of color *col* dominate B's nonexcess holes of color *col*; if not, A would contain a hole to the left of the next request for some block that is not a hole. If *col* is not an excess color of B, we are done with step 2. Otherwise, we need to show that all of B's excess holes of color *col* precede A's first hole. Suppose that B has $N_{col} + 1$ holes of color *col* at or to the right of A's first hole. A has only N_{col} holes of color *col*, so B has some hole h of color *col* that is not a hole of A and is to the right of A's first hole. Again, A would then contain a hole to the left of the next request for some block that is not a hole. Step 3 is the same as for part 1.

The proof of part 4 is an easy simplification of part 1, since A's new hole is beyond y and need not be considered. (B's new hole may be beyond y as well.) ∎

A particular case in which part 3 of Lemma 9 applies deserves mention. It may be that all blocks of some color *col* are holes in A, i.e., there are no blocks of color *col* in the cache, and that a fetch is not possible since there is no block of color *col* in the cache to evict, but that there are blocks of color *col* that are not holes in B. It may seem that a schedule with B as its set of holes has an advantage since it can make use of its disk *col* while a schedule with A as its set of holes cannot. But there is no advantage, provided that the conditions of strong domination are met. A is a superior state, and the schedule filling one of the holes in B by evicting a block of color *col* is merely "catching up" to the other schedule by eliminating one of its excess holes.

Here is our main result, the upper bound of Theorem 3.

Reverse aggressive *requires less than* $1 + DF/K$ *times the optimal elapsed time to service any request sequence, plus an additive term* DF *independent of the length of the sequence.*

Proof. For $D = 1$, the theorem follows directly from the result of [6]. Thus we may assume $D \geq 2$.

We show that for each $i \geq 0$ (numbering the phases starting with 0), there are times T_i and T'_i, such that

- T'_i is the time *opt*'s cursor reaches the ith phase;
- *reverse aggressive*'s cursor position at time T_i is at least as great as *opt*'s

cursor position at time T'_i;
- for $i > 0$, $T_i - T_{i-1} \leq T'_i - T'_{i-1} + DF - 1$;
- $H^+_{rev}(T_i)$ strongly dominates $H^+_{opt}(T'_i)$;
- if *reverse aggressive*'s busy disk for phase i will become free (i.e., complete any fetch in progress) in $z \leq F - 1$ steps after T_i, then *opt*'s corresponding disk will not become free until $z' \geq z$ steps after T'_i.

If there are p phases, we take T_p (respectively, T'_p) to be the time at which *reverse aggressive* (respectively, *opt*) finishes serving the request sequence.

The theorem will follow from the first three conditions as follows. For each phase i, *reverse aggressive*'s elapsed time $e_{rev}(i) = T_{i+1} - T_i$ and *opt*'s elapsed time $e_{opt}(i) = T'_{i+1} - T'_i$ satisfy

$$e_{rev}(i) \leq e_{opt}(i) + DF - 1$$

so that

$$\frac{e_{rev}(i)}{e_{opt}(i)} \leq 1 + \frac{DF - 1}{e_{opt}(i)}.$$

Each phase, except possibly the last, is of length at least K, so that $e_{opt}(i) \geq K$. Putting these together, we have that for all phases but the last,

$$\frac{e_{rev}(i)}{e_{opt}(i)} \leq 1 + \frac{DF - 1}{K}.$$

The last phase may be incomplete, i.e., may contain requests for fewer than K distinct blocks. *Reverse aggressive* requires at most $DF - 1$ steps more than *opt* to serve the last phase.

We prove the claims about T_i and T'_i by induction. For the base case ($i = 0$), we take $T_0 = T'_0 = 0$. The fact that the claims hold at this time is trivial. For the inductive step, assume the claims hold for the ith phase. We show that they hold for the $(i + 1)$st phase via a two-step process.
- We first show in Lemma 13 that in phase i, *reverse aggressive* (starting at time T_i) loses at most $(D - 1)F$ steps to *opt* (starting at time T'_i).
- We then use this fact to show that at the end of the phase, by giving *reverse aggressive* an extra $DF - 1$ steps relative to *opt* (from the start of the phase), the invariants are restored.

We begin with a formal statement of the first of these steps.

LEMMA 13. *Suppose that at time T_i,* reverse aggressive*'s cursor is at position p_i in the sequence. Let $T'_i + t_O(j)$ (respectively, $T_i + t_R(j)$) denote the time at which* opt *(respectively,* reverse aggressive*) serves the request at cursor position $j \geq p_i$, for any j such that r_j is in phase i. Then for all j in the phase, $t_R(j) \leq t_O(j) + (D - 1)F$.*

Proof. For the sake of contradiction, suppose the contrary, and consider the least index ℓ such that $t_R(\ell) > t_O(\ell) + (D - 1)F$.

First, consider the case in which ℓ precedes the first hole in $H^+_{rev}(T_i)$. Each of *reverse aggressive*'s fetches in progress at time T_i completes by time $T_i + F - 1$, so that *reverse aggressive*'s cursor cannot stall more than $F - 1$ steps before reaching the first hole in $H^+_{rev}(T_i)$. Recall we have assumed $D \geq 2$; thus we have a contradiction.

The remainder of the proof of Lemma 13 (and the bulk of that of the upper bound of Theorem 3) consists of the remaining case, in which ℓ is at or beyond the first hole in $H^+_{rev}(T_i)$. By the minimality of ℓ, $t_R(\ell - 1) \leq t_O(\ell - 1) + (D - 1)F$, and *reverse* *aggressive* stalls at least one step more than *opt* on request r_ℓ. In particular, *reverse aggressive* stalls at time $T_i + t_R(\ell) - 1$, and *opt* does not stall at time $T'_i + t_O(\ell)$. *Reverse aggressive* initiates a prefetch for the block requested at index ℓ at time $T_i + t_R(\ell) - 1 - x$ for some $0 \leq x \leq F - 1$; at this time, r_ℓ is *reverse aggressive*'s first hole. We will show that *opt* must have a hole that it has not yet begun to fill at an index no greater than ℓ at time $T'_i + t_O(\ell) - x$, and thus cannot serve r_ℓ before time $T'_i + t_O(\ell) - x + F > T'_i + t_O(\ell)$. Recall that *reverse aggressive*'s busy disk will be free in $x \leq F - 1$ steps after T_i, and *opt*'s corresponding disk will be free in $z' \geq z$ steps after T'_i.

Reverse aggressive will perform busy-disk fetches continuously, initiating a fetch at time $T_i + z + bF$ for each $b \geq 0$, at least until such a time as there are no holes left in the phase. Once there are no holes left in the phase, *reverse aggressive* will not stall at least until the end of the phase is reached. Let b and δ be such that $t_R(\ell) - 1 - x - z = bF + \delta$ and $0 \leq \delta < F$. Then *reverse aggressive* has filled b holes by busy-disk fetches by time $T_i + t_R(\ell) - 1 - x$, and *opt* has filled at most $b - D + 1$ holes by busy-disk fetches by time $T'_i + t_O(\ell) - x$, since

$$\begin{aligned} t_O(\ell) - x - z' &< t_R(\ell) - x - z - (D-1)F \\ &= bF + \delta + 1 - (D-1)F \\ &\leq (b - D + 2)F. \end{aligned}$$

Let n be the number of non-busy-disk fetches initiated by *opt* by time $T'_i + t_O(\ell) - x$. Consider the sequence $S = ((cur_1, col_1), \ldots, (cur_{n+b-D+1}, col_{n+b-D+1}))$ of fetches *opt* initiates after time T'_i and at or before time $T'_i + t_O(\ell) - x - F$, where the pair (cur, col) denotes that a fetch evicting a block of color col is initiated at cursor position cur. For each fetch (cur', col') of *opt*, we define a *matching fetch opportunity* of *reverse aggressive*. A matching fetch opportunity is a pair (cur, col) such that *reverse aggressive* has the opportunity to initiate a fetch of color col at a cursor position at least as great as cur. Each matching fetch opportunity to a fetch in S allows *reverse aggressive* to initiate a fetch (if allowed by the *do no harm* principle) by time $T_i + t_R(\ell) - 1 - x - F$. They are defined as follows:

- Let *opt*'s jth non-busy-disk fetch be initiated at time $T'_i + t'_j$. This fetch is matched to the fetch on the same disk that *reverse aggressive* initiates (if any) in the time interval

$$[T_i + t'_j + (D-1)F, T_i + t'_j + DF - 1].$$

Note that by the minimality of ℓ, at time $T_i + t'_j + (D-1)F$ *reverse aggressive*'s cursor is already at or beyond the cursor position at which *opt* initiates its jth non-busy-disk fetch, and its disk of the same color becomes free (finishes any fetch already in progress) within another $F - 1$ steps. Therefore, such a fetch opportunity exists. The fact that *reverse aggressive*'s cursor position at the time of this matching fetch opportunity is at least as great as *opt*'s at the time of its fetch will allow us to apply part 1 or part 3 of the strong domination lemma (Lemma 9) to this pair.

If *opt* initiates a total of n non-busy-disk fetches by time $T'_i + t_O(\ell) - x$, then each fetch except (possibly) the last one on each non-busy-disk (i.e., at least $n - (D-1)$ of the n non-busy-disk fetches) is initiated at a time less than or equal to $T'_i + t_O(\ell) - x - F$. Therefore, *reverse aggressive* can initiate a matching fetch if needed at a time strictly less than

$$T_i + t_O(\ell) - x + (D-1)F < T_i + t_R(\ell) - x.$$

- *opt*'s jth busy-disk fetch is matched to the jth busy-disk fetch *reverse aggressive* performs in the phase. Since *reverse aggressive* prefetches continuously using its busy disk, we know that each of these fetch opportunities corresponds to an actual fetch. Part 4 of the strong domination lemma will be applied to this pair of fetches.
- Finally, each non-busy-disk fetch initiated by *opt* between times $T_i' + t_O(\ell) - x - F + 1$ and $T_i' + t_O(\ell) - x$ is matched to one of the last $D - 1$ busy-disk fetches initiated by *reverse aggressive*. Note that there can be only one such fetch of each color. Part 4 of the strong domination lemma will be applied to this pair of fetches.

We claim that *reverse aggressive*'s holes after these $n + b - D + 1$ matching fetch opportunities strongly dominate *opt*'s holes up to the end of the phase after *opt* initiates its sequence S of n non-busy-disk fetches and at most $b - D + 1$ busy-disk fetches. Let R_0 be *reverse aggressive*'s set of holes $H^+_{rev}(T_i)$ at time T_i. Let O_0 be *opt*'s set of holes $H^+_{opt}(T_i')$ at time T_i'. Define O_j, $j \geq 1$, inductively as the set of holes resulting from initiating *opt*'s jth fetch (cur_j, col_j) with the set of holes O_{j-1}; i.e., $O_j = New(O_{j-1}, (cur_j, col_j))$. Similarly, define R_j, $j \geq 1$, inductively by $R_j = New(R_{j-1}, (cur_j, col_j))$. $R_{n+b-D+1}$ is the state that would be reached by starting in *reverse aggressive*'s state R_0, but then initiating fetches (when allowed by *do no harm*) according to *opt*'s prefetching schedule. By a sequence of applications of part 1 and part 3, as appropriate, of the strong domination lemma (Lemma 9), we have that $R_{n+b-D+1}$ strongly dominates $O_{n+b-D+1}$ up to the end of the phase.

We now show that *reverse aggressive*'s holes after its matching fetch opportunities pass strongly dominate $R_{n+b-D+1}$ up to the end of the phase. Because strong domination is transitive (Lemma 12), we will obtain that *reverse aggressive*'s holes strongly dominate *opt*'s up to the end of the phase. Since *opt* and *reverse aggressive* may perform fetches on different disks at different times and in different orders, we need to somehow permute *opt*'s schedule of fetches into *reverse aggressive*'s; then we will be able to make pairwise comparisons between the two sequences of fetches and apply the strong domination lemma. Toward this end, we define the following.

DEFINITION 14. *Consider a fetch sequence, defined by a sequence of triples of the form* (t_j, cur_j, col_j), *where for each* j, $t_j \leq t_{j+1}$ *and* $cur_j \leq cur_{j+1}$. (t_j, cur_j, col_j) *denotes a fetch, or an opportunity to fetch, beginning at time* t_j *with the cursor at position* cur_j, *where the color of the evicted block is* col_j. *A fetch opportunity denotes an opportunity to fetch in the sense that the disk is free, but no fetch may be possible under the optimal prefetching rules.*

DEFINITION 15. *A fetch sequence* S *is obtained from a fetch sequence* S' *by a* busy-early swap *if* S' *and* S *are the same except that a pair* (t_j', cur_j', col_j), $(t_{j+1}', cur_{j+1}', col_{j+1})$ *in* S' *is replaced by* (t_j, cur_j, col_{j+1}), $(t_{j+1}, cur_{j+1}, col_j)$ *in* S, *where* $cur_j \geq p_i$ *(recall that* p_i *is* reverse aggressive*'s cursor position at time* T_i*),* $cur_{j+1} \geq cur_j'$, *and* col_{j+1} *is the color of* reverse aggressive*'s busy disk for the phase.* $cur_j \geq p_i$ *will be enough to ensure that* reverse aggressive *is able to complete a fetch with the busy disk and that the new hole is beyond the end of phase* i, *which is enough to maintain strong domination up to the end of the phase, regardless of the fetch/eviction pair of* opt *to which this fetch of* reverse aggressive *is matched.*

DEFINITION 16. *A fetch sequence* S *is obtained from a fetch sequence* S' *by an* overlapping swap *if* S *and* S' *are the same except that a pair* (t_j', cur_j', col_j), $(t_{j+1}', cur_{j+1}', col_{j+1})$ *in* S' *is replaced by* (t_j, cur_j, col_{j+1}), $(t_{j+1}, cur_{j+1}, col_j)$ *in* S, *where* $t_{j+1}' < t_j' + F$, $t_{j+1} < t_j + F$, $cur_j \geq cur_{j+1}'$, *and* $cur_{j+1} \geq cur_j'$. *(Note* *that $cur_{j+1} \geq cur'_j$ is implied by $cur_j \geq cur'_{j+1}$, since cursor positions increase with time.)*

We extend the notation $New(A, (cur, col))$ to allow a series of fetches or fetch opportunities, with or without the time indices (which have no effect on the resulting set of holes), in the obvious way: $New(A, S) = New(New(A, f_1), f_2, \ldots, f_{|S|})$, where $S = f_1, \ldots, f_{|S|}$ is a sequence of fetches or fetch opportunities.

Before we can complete the proof of Lemma 13, we need the following three lemmas.

LEMMA 17. *Suppose that fetch sequence S within phase i is obtained from fetch sequence S' by a busy-early swap. Then $New(R_0, S)$ strongly dominates $New(R_0, S')$ up to the end of the phase.*

Proof. Let blue denote the color of *reverse aggressive*'s busy disk, and let red denote the color of the first disk to fetch under S' in the swapped pair. We refer to fetches using the blue disk as blue fetches, even though blue is the color of the evicted block; the block fetched may be any color. We refer to fetches using the red disk as red fetches, even though red is the color of the evicted item. The sets of holes of the two sequences immediately before initiating the swapped pair of fetches are the same. In both cases, a blue fetch can be initiated, since by hypothesis there are still holes in the phase. This blue fetch will not require an eviction that creates a new hole within the phase.

Unless the first hole filled is a red block, the set of red blocks in the cache at the time the red fetch is initiated is the same under S' and S. If the first hole is red, then under S', this red block is brought into the cache by the red fetch, and under S, by the blue fetch. Thus, the best eviction opportunity at the time of the red fetch under S is at least as good as that under S', since under S the red fetch occurs at a cursor position cur_{j+1} at least as great as that under S', which is cur_j.

Let the first hole occur at index h_1 and the second at h_2; let the new hole created by the red fetch under S' occur at index h_r. There are two possibilities:

- $h_2 < h_r$. Under S', the red fetch fills h_1 and the blue fetch fills h_2; under S, the blue fetch fills h_1 and the red fetch fills h_2. The red hole created under S is at a position in the request sequence at least as great as h_r, since the cursor position of the red fetch is at least as great as under S'. Under neither sequence does the blue eviction create a new hole in phase i. Thus, the sets of holes remaining in phase i after completing S' and S are the same, or after S one red hole has a greater index than after S'.
- $h_1 < h_r < h_2$. Under S', the red fetch fills h_1 and creates a hole at h_r. This new hole is the first hole at the time of the blue fetch, and thus the blue fetch fills it (leaving h_2 unfilled). Under S, however, the red fetch may be unable to proceed. The blue fetch fills the hole at h_1; after this, the first hole is at h_2. The red eviction of h_r would violate the rule *do no harm*. But the end result is the same as it is under S' (ignoring holes beyond the end of the phase): the next hole is at h_2, and a new blue hole has been created beyond the end of the phase. The red block requested at h_r does not get evicted and then fetched back, as it does under S'. (Again, under S it may be possible to create a red hole with greater index; in this case, h_2 gets filled, and the holes dominate those after S' up to the end of the phase by part 2 of the strong domination lemma.) ∎

LEMMA 18. *Suppose that fetch sequence S is obtained from fetch sequence S' by an overlapping swap. Then for any set A of holes, $New(A, S)$ strongly domi-*

nates $New(A, S')$ up to the end of the entire sequence and thus up to the end of the phase.

Proof. Neither fetch affects the eviction opportunities of the other, since they overlap and evict to different disks. Because they overlap, the first does not bring a block into the cache in time for it to be served before the second fetch starts. An easy consequence of the rules described in section 2.4 is that each block fetched is served at least once before it is subsequently evicted. Because they evict to different disks, the first does not evict a block that could otherwise be evicted by the second.

For each of the two fetches under S', the fetch of the same color under S is initiated at a cursor position at least as great. An argument similar to the proof of Lemma 17 finishes the proof. ∎

LEMMA 19. Reverse aggressive*'s sequence of fetch opportunities can be obtained from the sequence leading to $R_{n+b-D+1}$ (i.e.,* opt*'s sequence of fetches) via a sequence of busy-early swaps, overlapping swaps that do not involve fetches performed by the busy disk, substitutions of busy-disk fetches for non-busy-disk fetches, and insertions of extra fetches not matched to any fetch of* opt.

Proof. The definition of matching fetch opportunities identifies a sufficient set of fetch opportunities. We will show that no operations other than those described are necessary to transform *opt*'s sequence of fetches to *reverse aggressive*'s sequence of matching fetch opportunities.

First we show that for each disk other than the busy disk, any inversion of fetches on that disk and the busy disk is in the "right direction" (i.e., corresponds to a busy-early swap). Let blue denote the color of the busy disk, and let red denote the color of some other disk. For $1 \leq j \leq b$, let $T_i + t_{B_j}$ be the time at which *reverse aggressive*'s jth blue fetch is initiated, and for $1 \leq j \leq b - D + 1$, let $T'_i + t'_{B_j}$ be the time at which *opt*'s jth blue fetch is initiated. For $1 \leq j \leq r$, let t'_{R_j} be the time at which *opt*'s jth red fetch is initiated, and for $1 \leq j \leq r - 1$, let t_{R_j} be the time at which *reverse aggressive*'s matching fetch is initiated, where r is the number of red fetches initiated by *opt* at or before $T'_i + T_O(\ell)$.

First, consider all of *reverse aggressive*'s blue and red fetches except its last $D-1$ blue fetches, and all of *opt*'s blue and red fetches except its last red fetch (which is matched to one of *reverse aggressive*'s last $D - 1$ blue fetches). We have that for all $j \leq b-D+1$, $t_{B_j} \leq t'_{B_j}$ (i.e., *reverse aggressive*'s jth blue fetch is no later than *opt*'s, by the definition of matching fetch opportunities) and for all $j \leq r - 1$, $t_{R_j} \geq t'_{R_j}$ (i.e., *reverse aggressive*'s jth red fetch is no earlier than *opt*'s). Suppose that there is an inversion in the "wrong direction," i.e., that for some j and some k, $t'_{B_j} < t'_{R_k}$ and $t_{R_k} < t_{B_j}$. Then

$$t'_{B_j} < t'_{R_k} \leq t_{R_k} < t_{B_j} \leq t'_{B_j},$$

which contains the contradiction $t'_{B_j} < t'_{B_j}$.

Next, consider *opt*'s last (rth) red fetch. Recall that this fetch is matched to one of *reverse aggressive*'s last $D - 1$ blue fetches. This requires the substitution of a blue fetch for a red fetch, and possibly some number of busy-early swaps to move the blue fetch forward to its place in *reverse aggressive*'s sequence of fetches; no other red fetches in the sequence are affected by this.

For fetches other than blue fetches (i.e., non-busy-disk fetches), let $T'_i + t'_1$ and $T'_i + t'_2$ be the times of two fetches of *opt*, where $t'_1 \leq t'_2$, and let $T_i + t_1$ and $T_i + t_2$ be the times of *reverse aggressive*'s matching fetch opportunities. If *opt*'s fetches do not overlap, then $t'_1 \leq t'_2 - F$. By the definition of matching fetch opportunites, we have

$t_1 \leq t'_1 + DF - 1$ and $t_2 \geq t'_2 + (D-1)F$. Putting these together, we have $t_1 < t_2$, i.e., *reverse aggressive*'s matching fetch opportunities occur in the same order as *opt*'s fetches.

That the cursor positions of the swapped pairs satisfy the inequalities in the definitions of busy-early-swaps and overlapping swaps, respectively, can be seen from the definition of matching fetch opportunities. □

We now complete the proof of Lemma 13 using Lemmas 17, 18, and 19. We show that *reverse aggressive*'s holes at time $T_i + t_R(\ell) - 1 - x$ strongly dominate *opt*'s holes at time $T'_i + t_O(\ell) - x$ up to the end of the phase, as follows. Let $S_{opt} = S_1, S_2, \ldots, S_m = S_{rev}$ be the series of fetch sequences obtained in the transformation of *opt*'s fetch sequence into *reverse aggressive*'s that was shown to exist by Lemma 19. Recall that we have already shown that $New(R_0, S_{opt}) = R_{n+b-D+1}$ strongly dominates *opt*'s set of holes $New(O_0, S') = O_{n+b-D+1}$ up to the end of the phase. For each $1 < i \leq m$, $New(R_0, S_i)$ strongly dominates $New(R_0, S_{i-1})$ by Lemma 17, if S_i is derived from S_{i-1} by a busy-early swap, by Lemma 18, if S_i is derived from S_{i-1} by an overlapping swap, by part 2 of the strong domination lemma (Lemma 9), if S_i is derived from S_{i-1} by an insertion; or by part 4 of the strong domination lemma (Lemma 9), if S_i is derived from S_{i-1} by the substitution of a busy-disk fetch for a non-busy-disk fetch. By transitivity of strong domination (Lemma 12), $New(R_0, S_{rev})$ strongly dominates $New(O_0, S_{opt})$ up to the end of the phase.

Thus we have the following corollary.

COROLLARY 20. Reverse aggressive*'s first hole at time $T_i + t_R(\ell) - 1 - x$ is at a cursor position at least as great as* opt*'s first hole at time $T'_i + t_O(\ell) - x$.*

This contradicts the hypothesis that *reverse aggressive* stalls at time $T_i + t_R(\ell) - 1$ and *opt* does not stall at time $T'_i + t_O(\ell)$, and completes the proof of Lemma 13. □

We now use Lemma 13 to complete the inductive step of the proof of the upper bound of Theorem 3.

Let T'_{i+1} be the time at which *opt*'s cursor first reaches phase $i+1$ (i.e., one greater than the time at which *opt* serves the last request in phase i). Let f'_j be the jth fetch *opt* initiates after time T'_i and at or before time T'_{i+1}, and suppose it begins at time $T'_i + t'_j$. Define the jth *dominating fetch opportunity* to be the fetch opportunity (possibly an actual fetch) that *reverse aggressive* has on the same disk as f'_j in the time interval

$$[T_i + t'_j + (D-1)F, T_i + t'_j + DF - 1],$$

say, at time $T_i + t_j$. (Notice this is a different matching than that used in Lemma 13. In this matching, fetches of all colors are matched in the same way non-busy-disk fetches were matched in Lemma 13.) By Lemma 13, we know that *reverse aggressive*'s cursor position at time $T_i + t_j$ is at least as great as *opt*'s cursor position at time $T'_i + t'_j$.

By the same argument as in the proof of Lemma 19, *reverse aggressive*'s sequence of dominating fetch opportunities can be obtained from *opt*'s sequence of fetches by a series of overlapping swaps and insertions. Applying the strong domination lemma (Lemma 9), Lemma 18, and transitivity of strong domination (Lemma 12) as needed, we obtain that *reverse aggressive*'s holes after its dominating fetch opportunities have passed strongly dominate *opt*'s holes after completing its sequence of fetches. This is the same argument as in the proof of Lemma 13, but without the complication of busy-early swaps.

By Lemma 13, *reverse aggressive*'s cursor reaches phase $i+1$ by time $T_i + (T'_{i+1} - T'_i) + (D-1)F$. Within another $F-1$ steps, *reverse aggressive* initiates its dominating fetches matching the ones *opt* has in progress at time T'_{i+1}. A fetch of *opt* started at time $T'_{i+1} - x$ is matched (if needed) by *reverse aggressive* by time $T_i + (T'_{i+1} - T'_i) + DF - 1 - x$; in particular, if *opt* has a fetch in progress on *reverse aggressive*'s busy disk for phase $i+1$ at time T'_{i+1}, that fetch has at least as many steps remaining at time T'_{i+1} as *reverse aggressive*'s fetch (if any) has remaining at time $T_i + (T'_{i+1} - T'_i) + DF - 1$. Thus if we take time T_{i+1} to be $T_i + (T'_{i+1} - T'_i) + DF - 1$, the invariants are restored. $\square$

4.2. Reverse aggressive: Lower bound. We have been unable to strengthen the lower bound of Cao et al. [6], which showed that *aggressive* can perform $(1 + (F - 1)/K)$ times worse than optimal in the single-disk case. This bound applies directly to *reverse aggressive*, since there is no asymmetry between the reverse and forward problems in the single-disk case. It applies to the multiple-disk case as well, since a request sequence that contains only blocks that reside on a single disk is a special case.

4.3. Conservative: Lower bound. The following example shows that for $D < F \leq K$, there are arbitrarily long strings on which *conservative* requires time $1 + D\frac{K-F}{K}\frac{F}{F+D}$ times the optimal elapsed time.

Suppose that F divides K, and also that D divides K, and consider a repeated cycle on $K + (\frac{K}{F} - 1)D$ blocks. *Conservative* always evicts the page just referenced whenever it fills a hole, since that is the page that will not be needed again for the longest time. Thus *conservative* will never be able to overlap prefetches with each other or with references. Since there are at least $(\frac{K}{F} - 1)D$ holes on each pass through the cycle, *conservative* will spend at least $K + (\frac{K}{F} - 1)D + (\frac{K}{F} - 1)DF$ steps on each pass through the cycle. Suppose that the blocks are colored such that each contiguous sequence of D blocks in the cycle contains one block from each of the D disks. It is not hard to see that *opt* is able to maintain its holes in groups of D, one of each color, spaced F steps apart. Thus *opt* can service the entire sequence without stalling, and requires only $K + (\frac{K}{F} - 1)D$ steps on each pass through the cycle. The ratio of these two expressions (after a little manipulation) turns out to be at least as great as the stated bound.

4.4. Conservative: Upper bound. We now show that on any reference string R, the elapsed time of *conservative* with D disks on R is at most $D + 1$ times the elapsed time of the optimal prefetching strategy on R.

Let m be the minimum number of fetches (which is exactly how many fetches *conservative* performs) on request sequence R. *Conservative*'s elapsed time is at most $|R| + mF$, even if it never overlaps prefetches with each other or with the servicing of requests. Since the optimal algorithm *opt* must perform at least as many fetches as *conservative* and also must service the request sequence R, *opt*'s elapsed time is at least $\max(|R|, mF/D)$. The ratio of these is maximized with $|R| = mF/D$, and has the value $D + 1$.

4.5. Aggressive: Lower bound. The following example shows that for two disks, there are arbitrarily long strings on which *aggressive* requires time $2 - \frac{4}{F+2}$ times the optimal elapsed time (within an additive constant that depends only on F and K). In general, our bound is a little weaker: for D disks, there are arbitrarily long strings on which *aggressive* requires time $D - \frac{3D(D-1)}{F+3(D-1)}$ times the optimal elapsed time (within an additive constant that depends only on F and K). Consider the sequence

$$b_1 b_2 r_1 \cdots r_F b_3 b_4 r_F \cdots r_1 b_2 b_1 r_1 \cdots r_F b_4 b_3 \cdots,$$

where all r_i are red and all b_i are blue. Let $K = F + 2$. The initial cache contents are b_1, b_2, and $r_1 \cdots r_F$; there are holes at the first references to b_3 and b_4. Both algorithms service the initial request of b_1 during the first unit of time. *Aggressive* then evicts the block in its cache not referenced for the longest time, b_1, in order to fetch b_3; the optimal algorithm *opt* does the same. At the completion of this fetch, the next hole for both algorithms is at b_4, and the cursor is at the first request of r_F. *Aggressive* immediately evicts the block among those in the cache not used for the longest time, which is now b_2; *opt* evicts r_1 instead. Both algorithms stall for $F - 2$ steps on the hole at b_4. However, *opt* is able to initiate a fetch of its next hole, r_1, evicting b_3, since the hole is red and the fetch in progress is fetching a blue block; *aggressive* is unable to perform a second fetch in parallel because its next hole (b_2) is also blue. Notice that *aggressive* still has no red holes, and thus can complete only one fetch every F steps. From this point on, *opt* is able to create one red and one blue hole in each subsequence of $F + 2$ requests, and can always fill them without stalling, whereas *aggressive* will always create a pair of blue holes, and will require time $2F$ to serve each subsequence of $F + 2$ requests, since it takes this long to complete two fetches. Thus from this point on, the ratio of *aggressive*'s elapsed time to that of *opt* is $\frac{2F}{F+2} = 2 - \frac{4}{F+2}$.

We have illustrated the case $K = F + 2$, $D = 2$ for simplicity. It is easily generalized to larger values of $\frac{K}{F}$ (which are the cases of interest in practice) as follows: let $K = iF + 2$, and interleave i distinct subsequences of F distinct red blocks each with $i + 1$ distinct pairs of blue blocks in round-robin fashion, reversing each subsequence of red blocks and each pair of blue blocks on alternate occurrences. It is not hard to see that *aggressive* will behave similarly to the illustrated case, and that *opt* is able to service the sequence without stalling (after an initial startup period).

The generalization to $D > 2$ is also straightforward. Consider the sequence

$$b_1 \cdots b_D b_1 \cdots b_{D-2} x_1 \cdots x_{D-1} r_1 \cdots r_{F-D+1} x'_1 \cdots x'_{D-1} \cdots$$

$$\cdots b_{D+1} \cdots b_{2D} b_{D+1} \cdots b_{2D-2} \cdots,$$

where $F > D$ and $K = F + 2D - 1$, the colors of the b_i are all the same, the colors of the x_i are distinct from each other and the color of the b_i, and the color of x'_i is the same as that of x_i. We omit the details of the startup period, and note that if *aggressive* has holes at $b_1 \cdots b_D$, it will fill them by evicting $b_{D+1} \cdots b_{2D}$ and thus requires time at least DF to serve the sequence up to b_{D+1}. Its state is then similar to the state in which it started, and thus the process can repeat indefinitely. *opt*, on the other hand, is able to maintain D holes of D distinct colors, and can serve the sequence without stalling. Each sequence of $3(D - 1) + F$ requests requires time $3(D - 1) + F$ for *opt*, and DF for *aggressive*, for a ratio of

$$\frac{DF}{3(D-1)+F} = D - \frac{3D(D-1)}{F+3(D-1)}.$$

Again, generalizing to arbitrary K/F is easy.

4.6. Aggressive: Upper bound. First we state a very simple lemma, leaving the proof to the reader.

LEMMA 21. *If a set A of holes dominates a set B of holes, and some hole in A is filled and some hole at a larger index added to A, the resulting holes A' dominate B.*

We now show that on any reference string R, the elapsed time of *aggressive* with D disks on R is at most $D(1+\frac{F+1}{K})$ times the elapsed time of the optimal prefetching strategy on R.

Proof. In the analysis of aggressive prefetching with one disk, it was shown that if A's holes dominate B's holes, and A's cursor position is at least as great as B's, and each algorithm initiates a fetch, A's holes will continue to dominate B's when the fetch is completed. This result was referred to as the *domination lemma* [6]. The proof of this is similar to but simpler than that of Lemma 9 for algorithms working with the reverse sequence.

In order to apply this lemma to more than one disk, we must be sure that when we are comparing a fetch A initiates to a fetch B initiates that the hole being filled by A is *the first hole*. If not, the domination lemma does not hold.

In general, we cannot ensure that D parallel prefetches *aggressive* initiates will fill the first D holes, since some of these holes may be of the same color. However, we do know that by the time *aggressive* completes D prefetches on the same disk, the first D holes that were present (and perhaps others) have been filled. Thus our proof strategy is to run *opt* at $1/D$ times the speed of *aggressive*, so that during each subsequence of time in which *aggressive* fills *at least* its first D holes, *opt* can fill *at most* its first D holes. We will show inductively that at the end of each of these subsequences, *aggressive*'s holes dominate *opt*'s holes. This will imply that *aggressive* can take only about D times as long as *opt* to complete a phase.

Notice that as long as there are holes in the phase containing the cursor, there are blocks in the cache which are not requested before the end of the phase (since the cache holds K blocks and there are only K distinct requests in a phase). Since *aggressive* always evicts the block that is not requested for the longest time, once its cursor enters a phase, *aggressive* will not create any new holes within the phase. Also, once *aggressive* enters a phase, each disk will initiate a fetch every F steps as long as there are holes of that disk's color remaining in the phase.

We show that for each i such that $0 \le i < p-1$ where p is the number of phases (numbering the phases starting with 0), there are times T_i and T'_i, such that

- $T_i \le DT'_i + i(D+1)F$;
- *aggressive*'s cursor is in the ith phase of the request sequence at time T_i;
- *opt*'s cursor at time T'_i is not past the first request of phase i;
- $H^-_{agg}(T_i)$ dominates $H^+_{opt}(T'_i)$, so that each of *aggressive*'s disks is either ready to initiate a prefetch or is already filling a hole in phase i, for which *opt* has not yet started filling its matching hole.

The theorem will follow from the first three conditions, as follows. For each phase i, *aggressive*'s elapsed time $e_{agg}(i)$ and *opt*'s elapsed time $e_{opt}(i)$ satisfy

$$e_{agg}(i) \le De_{opt}(i) + (D+1)F$$

so that

$$\frac{e_{agg}(i)}{e_{opt}(i)} \le D + \frac{(D+1)F}{e_{opt}(i)}.$$

Each phase except possibly the last is of length at least K, so that $e_{opt}(i) \ge K$. Putting these together, we have that for all phases but the last,

$$\frac{e_{agg}(i)}{e_{opt}(i)} \le D + \frac{(D+1)F}{K}.$$

The last phase may be incomplete, i.e., may contain requests for fewer than K distinct blocks. *Aggressive* requires at most D times as many steps as *opt* to serve the last phase, as shown below.

This claim is proven by induction on i. The basis ($i = 0$) is trivial, since both algorithms start at the beginning of the first phase in the same state, with all disks idle.

For the induction, assume that the claim is true for i.

We first show that for each index j in phase i, *aggressive*'s cursor passes j after at most D times as many steps as *opt*'s cursor takes to pass j. Let $T_i + t_A(j)$ be the time *aggressive* serves request j, and let $T'_i + t_O(j)$ be the time *opt* serves j. Assume by way of contradiction that *aggressive*'s cursor falls behind *opt*'s (relative to the start of the phase) by more than a factor of D, and let ℓ be the least index for which this happens, i.e., $t_A(\ell) > Dt_O(\ell)$. It must be true that *aggressive* has a hole at ℓ (or equivalently stalls on the ℓth request in the phase) at time $T_i + t_A(\ell) - 1$, and that the ℓth request in the phase is in *opt*'s cache before time $T'_i + t_O(\ell)$, since $T_i + t_A(\ell)$ is the *first* time *aggressive*'s cursor falls behind *opt*'s by more than a factor of D. As noted previously, each disk of *aggressive*'s fills a hole every F steps as long as there are holes of that disk's color in the phase. Let h be the number of holes in $H^-_{agg}(T_i)$ that are the same color as the one at ℓ, up to and including the one at ℓ. Then $t_A(\ell) \leq hF$, since the hole at ℓ is filled at a time no later than $T_i + hF$. $H^+_{opt}(T'_i)$ contains at least h holes at or before ℓ, since $H^-_{agg}(T_i)$ dominates $H^+_{opt}(T'_i)$. Thus the earliest time *opt* could finish filling all its holes up to index ℓ is $T'_i + \lceil h/D \rceil F$, even if it fills a hole every F steps with each disk. Thus we have a contradiction: $hF \geq t_A(\ell) > Dt_O(\ell) \geq D(\lceil h/D \rceil F) \geq hF$.

To show that *aggressive*'s holes after finishing phase i dominate *opt*'s holes, we need another induction. Let I'_j denote the F-step interval $[T'_i + jF, T'_i + (j+1)F)$, $j \geq 0$, and let cur_j be *opt*'s cursor position at time $T'_i + jF$, for each j such that *opt*'s cursor is still in phase i at time $T'_i + jF$. Let $I_j = [T_i + jDF, T_i + (j+1)DF)$. Consider the set of at most D fetches that *opt* initiates during I'_j. We match these to the set of fetches *aggressive* initiates during I_{j+1}. We prove by induction on j that $H^+_{opt}(T'_i + jF)$ is dominated by $H^+_{agg}(T_i + D(j+1)F)$. The base case follows from the hypothesis that $H^-_{agg}(T_i)$ dominates $H^+_{opt}(T'_i)$. Any fetches completed or initiated by *aggressive* during I_0 do not affect this, by Lemma 21. For the inductive step (on j), note that each fetch *opt* initiates during I'_j is initiated at a cursor position at most cur_{j+1}, and that *aggressive*'s cursor position is at least cur_{j+1} during the interval I_{j+1}. Thus *aggressive*'s fetches can be matched to *opt*'s and the domination lemma implies that *aggressive*'s resulting holes $H^+_{agg}(T_i + (j+2)DF)$ dominate *opt*'s resulting holes $H^+_{opt}(T'_i + (j+1)F)$. Any extra fetches of *aggressive* (there may actually be as many as D^2 by *aggressive* and as few as zero by *opt* during their respective time intervals) do not affect this, by Lemma 21. As a special case, if *aggressive* should stop fetching altogether at some time and thus have fewer than D fetches to match to *opt*'s, *aggressive* has reached the optimal cache configuration: its cache contains the next K distinct requests, and its holes are as far from the cursor as possible. These holes certainly dominate *opt*'s holes at any earlier cursor position.

Consider the value j^* such that *opt*'s cursor reaches phase $i+1$ during I'_{j^*}. Then by the preceding arguments, *aggressive*'s cursor reaches phase $i+1$ by time $T_i+(j^*+1)DF$ and *aggressive*'s holes $H^+_{agg}(T_i+(j^*+1)DF) = H^-_{agg}((j^*+1)DF+F)$ after completing all fetches initiated in I_{j^*} dominate *opt*'s holes $H^+_{opt}(T'_i + j^*F)$ after completing all fetches initiated in I'_{j^*-1}. Let $T_{i+1} = T_i + (j^*+1)DF + F$ and let $T'_{i+1} = T'_i + j^*F$, and the conditions for the induction step on the phase index i are met. ∎

5. The algorithms' running times. In this section we consider the time required to determine a prefetching and caching schedule in the uniform-cost RAM model (see, for example, [1]). This is distinct from the time required to serve the sequence in the model described in section 1.2, which is the primary measure we are trying to optimize.

First, consider the single-disk case. We assume that the ith member of the set B of blocks is identified by the integer i. We will need per-block lists of requests (indices in the request sequence R); let $Next(b)$ refer to the head of the list of references to block b. Initially, $Next(b)$ points to the first request of block b; after that request is served, $Next(b)$ will be updated to point to the next occurrence of b in R, and so on. We will also need a vector $InCache$ indexed by the set B indicating for each block whether it is present in the cache, and a pointer $NextHole$ indicating the index of the first hole in the request sequence. Finally, we will need a priority queue $Cache$ containing the identifiers of all blocks present in the cache and keyed on the index in the request sequence of the next request to that block. $Cache$ will need to be augmented by an operation to update the key of an item (which could be implemented as a deletion and a reinsertion), as well as to the usual operations to insert items and delete the item with maximum key. Note that $Cache$ will never contain more than K keys. Each operation on $Cache$ thus requires $O(\log K)$ time (see, for example, [1]). Note that the maximum element in $Cache$, the value of $NextHole$, and the position of the cursor provide the information needed by *aggressive* and *reverse aggressive* to decide when and what to prefetch, and what to evict.

A preprocessing step to initialize these data structures requires time linear in $|B| + |R|$; we assume $K \leq |B|$, since the scheduling problem is trivial otherwise. To maintain these structures when serving a request of block b, we need to update the pointer $Next(b)$ and update b's entry in the priority queue $Cache$. Thus scheduling the servicing of a request requires $O(\log K)$ time. To maintain these structures when evicting a block b_1 and fetching b_2, we delete the maximum element (which is b_1) from $Cache$, insert b_2 in $Cache$, update the vector $InCache$ appropriately, and scan forward in R from $NextHole$ until a request is found that is missing from the cache (by referring to $InCache$); this index becomes the new $NextHole$. These operations require time $O(\log K)$ with the exception of the scan of the request sequence to find the new $NextHole$. The scans require $O(|R|)$ time, amortized over the entire sequence. $|R|$ is an upper bound on the total number of fetches. The reversal of R and of *reverse aggressive*'s reverse schedule can be done in time linear in $|R|$. Thus, each of the algorithms *aggressive* and *reverse aggressive* can be implemented to run in time $O(|B| + |R| \log K)$ in the uniform-cost RAM model.

A simple implementation of *conservative* is to run Belady's paging algorithm, recording each fetch/eviction pair along with a "release index," i.e., the index of the last request of the evicted block (before it is fetched back into the cache later in the schedule, if ever). A similar analysis to that above shows the same bound of $O(|B| + |R| \log K)$ for the construction of this list of fetches and evictions. The list can then be "played back" to construct a schedule for the fetches and the serving of the sequence, issuing each fetch as soon as the cursor has passed the release index and the disk is free. Thus, we have the same bound on *conservative*'s running time as on that of the other algorithms.

In the case of $D > 1$ disks, we assume a constant-time operation yields the disk a block resides on, given the block's identifier. The changes required in the analysis of *conservative* are trivial. For the other algorithms, data structures are maintained on a per-disk basis as needed. $NextHole$ becomes a vector of D entries for *aggressive*. A linear time preprocessing step can be used to produce per-disk request sequences; these are needed to update $NextHole$. In the case of *reverse aggressive*, it is the priority queue $Cache$ that needs to be split into D separate structures, one for each disk; none will ever contain more than K keys. Thus, the running time bound given above applies to the multidisk case as well as the single-disk case.

6. Empirical results.

Reverse aggressive is not a practical algorithm. However, it serves as a benchmark against which to compare practical algorithms such as *aggressive* and the algorithms mentioned below. Perhaps more important, an understanding of the reasons for *reverse aggressive*'s performance guarantee led to the design of the practical algorithm *forestall* with performance matching that of *reverse aggressive* in empirical studies.

As mentioned, in joint work with A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. A. Gibson, and K. Li, we have performed simulation studies of the performance of prefetching and caching algorithms. These results are reported in companion papers [17, 19]. We implemented the *aggressive* and *reverse aggressive* algorithms and tested them on reference streams taken from real file systems. We also implemented two other algorithms, *fixed horizon* and *forestall*.

Fixed horizon. For a fixed value H, whenever there is a missing block at most H references in the future, *fixed horizon* issues a fetch for that block, replacing the cached block whose next reference is furthest in the future, provided that reference is further than H accesses in the future.

Forestall. For each disk, for each i, $i \geq 1$, let d_i denote the distance from the cursor to the ith missing block in the request sequence that resides on the disk. For any $i \geq 1$, if $iF > d_i$, processing will surely stall on the ith missing block or some earlier missing block. It will take iF time units to fetch the first i missing blocks, and at most the next d_i requests can be served concurrently. *Forestall* initiates a prefetch according to the (forward) *optimal fetching* and *optimal replacement* rules whenever $iF \geq d_i$ is true for some i and the *do no harm* rule allows it.

See [19] for intuition and more details on these algorithms. *Forestall* and *fixed horizon* have worst-case performance similar to *aggressive*'s. All three of these algorithms perform much better than the worst case in practice.

Our experimental work models many details of real systems not captured by the theoretical model. These are as follows:

- Disk response times and CPU times between I/O requests are not constant. We use average values for each and expect that variation in event times does not substantially invalidate the algorithm's decisions. In our experimentation, this does not appear to be a major effect. (The systematic effects of disk scheduling on disk response time are considered separately.)
- Disk response time is sensitive to the order in which requests are serviced. In particular, disk scheduling reduces average disk response time as more accesses are presented and allowed to be reordered by the I/O driver. Although *fixed horizon* implicitly allows multiple outstanding requests at each disk, the other algorithms were defined to submit only one request at a time, since in the theoretical model there is no advantage to batching. Because of the significance of the disk scheduling effect, we modify the definitions of the other algorithms to submit disk requests in batches. We have found that the performance of all four algorithms benefits from the CSCAN disk

scheduling algorithm.

- Access patterns exhibit locality of reference, and loads are balanced across the multiple disks when data is laid out well. In practice, this allows the other algorithms to effectively utilize multiple disks, and to achieve elapsed times comparable to the theoretically superior *reverse aggressive*.
- Disk accesses require significant CPU overhead to form the request, communicate with the disk, and service the resulting interrupt(s). Thus, avoidable data fetches may add elapsed time even if they do not cause stalls. Because the theory assumes that fetches entail no CPU overhead, this penalty punishes overly aggressive fetching. In practice, this effect favors the *fixed horizon* algorithm since its late replacement decisions tend to lead to the fewest fetches.

The following is a summary of our empirical findings:

- All four algorithms significantly outperform demand fetching, even when advance knowledge of the access sequence is used to make optimal replacement decisions in conjunction with demand fetching.
- In compute-bound situations, *fixed horizon* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).
- In I/O-bound situations, *aggressive* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).
- In any given situation, one of *fixed horizon* or *aggressive* performs close to the theoretically near-optimal *reverse aggressive*.
- In all situations, *forestall* performs close to *reverse aggressive*.
- When data is well laid out on the disks, disk loads are balanced even without careful replacement choices. For this reason, *reverse aggressive* does not significantly outperform the other algorithms.
- *Fixed horizon* places the least I/O load on the disks, due to its conservative fetching and near-optimal replacement choices; *aggressive* places the greatest load on the disks. *Reverse aggressive* and *forestall* are intermediate between *aggressive* and *fixed horizon*.
- *Forestall* is a promising new approach that combines the best features of the other three algorithms: good performance regardless of I/O- or compute-boundedness, simplicity, and practicality.

For further details, see [19].

7. Summary and open problems.

In this paper, we have considered algorithms for prefetching and caching of data from multiple backing stores. Previous algorithms, designed for the case of a single backing store, were found to be suboptimal by a factor near the number of backing stores. Our main result is a new algorithm, *reverse aggressive*, that is provably near optimal. Although *reverse aggressive* is not a practical algorithm, this paper lays theoretical groundwork for the parallel prefetching and caching problem and exposes its structure. Also, as described in section 6, *reverse aggressive* serves as an important benchmark against which to compare more practical algorithms.

We know of no polynomial-time algorithm for optimal prefetching for multiple disks. It is a difficult problem to find either such an algorithm or a proof of hardness. Another very interesting direction is to extend these results to the case in which only probabilistic information is available about the request sequence.

Acknowledgments. We thank Richard Anderson for helpful discussions of this work and for commenting on a draft of this paper.

Pei Cao, Ed Felten, and Kai Li introduced us to this problem, helped us to formalize the model, and collaborated with us on the empirical evaluation. Their contribution to this work has been enormous.

Last, but not least, Martin Tompa's unfailing patience and encouragement have been indispensable to the first author. Both authors are extremely grateful to Martin for lending his insightful suggestions and constructive criticism to this work at every step of the way.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison–Wesley, Reading, MA, 1983.

[2] S. Albers, N. Garg, and S. Leonardi, *Minimizing stall time in single and parallel disk systems*, in Proceedings of the ACM Symposium on Theory of Computing, Dallas, TX, 1998, pp. 454–462.

[3] L. A. Belady, *A study of replacement algorithms for virtual storage computers*, IBM Systems Journal, 5 (1966), pp. 78–101.

[4] P. Cao, E. Felten, and K. Li, *Application-controlled file caching policies*, in Proceedings of the USENIX Summer 1994 Technical Conference, Boston, MA, 1994, pp. 171–182.

[5] P. Cao, E. Felten, A. Karlin, and K. Li, *Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling*, Technical report TR-CS95-493, Princeton University, Princeton, NJ, 1995.

[6] P. Cao, E. Felten, A. Karlin, and K. Li, *A study of integrated prefetching and caching strategies*, in Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Ottawa, Canada, 1995, pp. 188–197.

[7] P. Cao, E. Felten, and K. Li, *Implementation and performance of application-controlled file caching*, in Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA, 1994, pp. 165–178.

[8] P. M. Chen and D. A. Patterson, *Maximizing performance in a striped disk array*, in Proceedings of the 17th Annual Symposium on Computer Architecture, Seattle, WA, 1990, pp. 322–331.

[9] H. T. Chou and D. J. DeWitt, *An evaluation of buffer management strategies for relational database systems*, in Proceedings of the 19th International Conference on Very Large Data Bases, Dublin, Ireland, 1993, pp. 127–141.

[10] K. M. Curewitz, P. Krishnan, and J. S. Vitter, *Practical prefetching via data compression*, in Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD), Washington, DC, 1993, pp. 257–266.

[11] C. Ellis and D. Kotz, *Prefetching in file system for MIMD multiprocessors*, in Proceedings of the 1989 International Conference on Parallel Processing, St. Charles, IL, 1989, pp. 306–314.

[12] R. J. Feiertag and E. I. Organisk, *The multics input/ouput system*, in Proceedings of the 3rd Symposium on Operating Systems Principles, Palo Alto, CA, 1971, pp. 35–41.

[13] J. Gray, B. Horst, and M. Walker, *Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput*, in Proceedings of the 16th International Conference on VLDB, Brisbane, Australia, 1990, pp. 148–161.

[14] J. Griffioen and R. Appleton, *Reducing file system latency using a predictive approach*, in Proceedings of the USENIX Summer 1994 Technical Conference, Boston, MA, pp. 197–208.

[15] J. H. Howard, M. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, *Scale and performance in a distributed file system*, ACM Trans. Comput. Systems, 6 (1988), pp. 51–81.

[16] M. Kim, *Synchronized disk interleaving*, IEEE Trans. Comput., 35 (1986), pp. 978–988.

[17] T. Kimbrel, P. Cao, E. Felten, A. Karlin, and K. Li, *Integrated parallel prefetching and caching*, in Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA, 1996, pp. 262–263.

[18] T. Kimbrel and A. Karlin, *Near-optimal parallel prefetching and caching*, in Proceedings of the IEEE Symposium on Foundations of Computer Science, Burlington, VA, 1996, pp. 540–549.

[19] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. A. Gibson, A. Karlin, and K. Li, *A trace-driven comparison of algorithms for parallel prefetching and caching*, in Proceedings of the ACM SIGOPS/USENIX Association Symposium on

Operating System Design and Implementation (OSDI), Seattle, WA, 1996, pp. 19–34.

[20] P. Krishnan and J. S. Vitter, *Optimal prediction for prefetching in the worst case*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), Arlington, VA, 1994, pp. 392–401.

[21] D. Kotz and C. Ellis, *Practical prefetching techniques for multiprocessor file systems*, J. Distributed and Parallel Databases, 1 (1993), pp. 33–51.

[22] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, *A fast file system for UNIX*, ACM Trans. Comput. Systems, 2 (1984), pp. 181–197.

[23] L. W. McVoy and S. R. Kleiman, *Extent-like performance from a UNIX file system*, in Proceedings of the 1991 Winter USENIX Conference, Dallas, TX, 1991, pp. 33–43.

[24] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, *Caching in the sprite file system*, ACM Trans. Comput. Systems, 6 (1988), pp. 134–154.

[25] V. S. Pai, A. A. Schäffer, and P. J. Varman, *Markov analysis of multiple-disk prefetching strategies for external merging*, Theoret. Comput. Sci., 128 (1994), pp. 211–239.

[26] M. Palmer and S. B. Zdonik, *Fido: A cache that learns to fetch*, in Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, Spain, 1991, pp. 255–264.

[27] D. A. Patterson, G. Gibson, and R. H. Katz, *A case for redundant arrays for inexpensive disks (RAID)*, in Proceedings of ACM SIGMOD Conference, Chicago, IL, 1988, pp. 109–116.

[28] R. H. Patterson and G. A. Gibson, *Exposing I/O concurrency with informed prefetching*, in Proceedings of the Third International Conference on Parallel and Distributed Information Systems, Austin, TX, 1994, pp. 7–16.

[29] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, *Informed prefetching and caching*, in Proceedings of the 15th Symposium on Operating Systems Principles, Copper Mountain Resort, CO, 1995, pp. 79–95.

[30] K. Salem and H. Garcia-Molina, *Disk striping*, in Proceedings of the 2nd IEEE Conference on Data Engineering, Los Angeles, CA, 1986, pp. 336–342.

[31] A. J. Smith, *Second bibliography on cache memories*, Computer Architecture News, 19 (1991), pp. 154–182.

[32] C. Tait and D. Duchamp, *Service interface and replica management algorithm for mobile file system clients*, in Proceedings of Parallel and Distributed Information Systems, IEEE, Miami Beach, FL, 1991, pp. 190–196.

[33] J. S. Vitter and P. Krishnan, *Optimal prefetching via data compression*, in Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), San Juan, Puerto Rico, 1991, pp. 121–130.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1083–1117

COMPLEXITY RESULTS FOR FIRST-ORDER TWO-VARIABLE LOGIC WITH COUNTING*

LESZEK PACHOLSKI[†], WIESŁAW SZWAST[‡], AND LIDIA TENDERA[‡]

Abstract. Let C_p^2 denote the class of first-order sentences with two variables and with additional quantifiers "there exists exactly (at most, at least) i" for $i \leq p$, and let C^2 be the union of C_p^2 taken over all integers p. We prove that the satisfiability problem for C_1^2 sentences is NEXPTIME-complete. This strengthens the results by [E. Grädel, Ph. Kolaitis, and M. Vardi, Bull. Symbolic Logic, 3 (1997), pp. 53–69], who showed that the satisfiability problem for the first-order two-variable logic L^2 is NEXPTIME-complete and by [E. Grädel, M. Otto, and E. Rosen, *12th Annual IEEE Symposium on Logic in Computer Science*, 1997, pp. 306–317], who proved the decidability of C^2. Our result easily implies that the satisfiability problem for C^2 is in nondeterministic, doubly exponential time. It is interesting that C_1^2 is in NEXPTIME in spite of the fact that there are sentences whose minimal (and only) models are of doubly exponential size.

It is worth noticing that by a recent result of [E. Grädel, M. Otto, and E. Rosen, *Proceedings of 14th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 1200, Springer-Verlag, Berlin, 1997], extensions of two-variable logic L^2 by a weak access to cardinalities through the Härtig (or equicardinality) quantifier is undecidable. The same is true for extensions of L^2 by very weak forms of recursion.

The satisfiability problem for logics with a bounded number of variables has applications in artificial intelligence, notably in modal logics (see, e.g., [W. van der Hoek and M. De Rijke, *J. Logic Comput.*, 5 (1995), pp. 325–345]), where counting comes in the context of graded modalities and in description logics, where counting can be used to express so-called number restrictions (see, e.g., [A. Borgida, *Artificial Intelligence*, 82 (1996), pp. 353–367]).

Key words. decision problem, computational complexity, first-order logic

AMS subject classifications. 03B10, 68Q15, 03D15, 68Q25

PII. S0097539797323005

1. Introduction.

Let L^2 denote the class of first order sentences with two variables over a relational vocabulary, and let C_p^2 denote L^2 extended with additional quantifiers "there exists exactly (at most, at least) i" for $i \leq p$. Finally, let C^2 be the union of C_p^2 taken over all integers p. We prove that the problem of satisfiability of sentences of C_1^2 is NEXPTIME-complete.

Problems concerning decidability of restricted classes of quantificational formulas have been studied since the second decade of this century by many logicians, including Ackermann, Bernays, Gödel, Kalmár, Schönfinkel, Skolem, Wang [1, 2, 5, 11, 12, 24, 33, 34, 35, 37], and many others. In the late twenties and in the thirties (see [7] and [19] for more information) the study of classification of solvable classes of prenex formulas was one of the most active areas of logic. Now, after the works of Gurevich [18], Rabin [30], Shelah [32], and Goldfarb [14], the classification of prenex classes has been completed. Accounts of the classical results in this area can be found in several books [3, 7, 9, 25]. More recent results have been obtained by Lewis and Goldfarb

*Received by the editors June 16, 1997; accepted for publication (in revised form) March 15, 1999; published electronically February 1, 2000. The results included in section 4 have been published as a part of L. Pacholski, W. Szwast, and L. Tendera, *Complexity of two-variable logic with counting*, in 12th Annual IEEE Symposium on Logic in Computer Science, 1997, pp. 318–327.

http://www.siam.org/journals/sicomp/29-4/32300.html

[†]Institute of Computer Science, Wrocław University, Przesmyckiego 20, 51-151 Wrocław, Poland (pacholsk@tcs.uni.wroc.pl). This author was partially supported by KBN grant 8 T11C 029 13.

[‡]Institute of Mathematics, Opole University, Oleska 48, 45-052 Opole, Poland (szwast@math.uni.opole.pl, tendera@math.uni.opole.pl).

[13, 14, 26]. A short survey of the research in this area can be found in [19] (see also the introduction to [15]).

In 1962, in a short note, D. Scott [31] proved that the satisfiability problem for L^2 was decidable. His proof was based on a reduction of this problem to the problem of satisfiability of sentences in the Gödel class with equality. Later, in 1975, Mortimer gave another proof of decidability by proving that L^2 has a finite model property. When in 1984 Goldfarb [14] found a counterexample to the claim, that the Gödel class with equality had a decidable satisfiability problem, the very short and elegant proof by Scott lost its validity. In 1980 Lewis [27] proved that the satisfiability problem for L^2 was NEXPTIME-hard. The complexity of an algorithm which could be extracted from Mortimer's work was doubly exponential. Recently, Grädel, Kolaitis, and Vardi [15] have closed the gap by providing a very elegant proof that the satisfiability problem for L^2 was in NEXPTIME. Later we found another proof [36] of the same result. Our proof was not as nice as the one in [15], but we hoped it could be extended to get the complexity bounds for C^2.

In [16], Grädel, Otto, and Rosen established decidability of the satisfiability problem for C^2. They proved that the set of sentences which have infinite models was recursive, which implied the above mentioned result. No complexity estimates could be obtained from their proof.

In this paper we prove that the satisfiability problem for C^2_1 is in NEXPTIME, so by the result of Lewis [27] it is NEXPTIME-complete. By the reduction of C^2 to C^2_1 given in [16] this implies that C^2 is in 2-NEXPTIME. Although our strict upper bound applies only to C^2_1 we believe that we have developed techniques that can be used to close the gap for the entire class C^2.

Our approach is in a very remote way based on the ideas of Mortimer. A very simple cardinality argument shows that Mortimer's notion of a *star* could not be used to give a NEXPTIME decision procedure. This led to a weakening of this notion to the notion of a *constellation.* As the first application of this notion we gave in [36] another proof of the result of Grädel, Kolaitis, and Vardi [15]. There we have also used a stronger notion of a normal form—going further than Grädel, Otto, and Rosen in [16]—a *constellation form*, in which, additionally, constant symbols do not appear. The proof in [36] is "syntactic"; however, in the case of L^2, the "syntactic" structure coding a model is almost equivalent to a model. This changes dramatically when we move to C^2 and allows for a concise description of models that can be even infinite.

In contrast to [16] our basic notions are almost entirely syntactic. We like the feudal terminology of [16] and we treat *kings* of [16] with proper care and respect. However, *kings* in our sense have other virtues besides belonging to a finite set. Population of our kings is always at most doubly exponential in the size of the language. On the other hand, it is easy to give examples of models whose sets of kings in the sense of [16] have, for a language of bounded size, arbitrary large cardinality. This seems to suggest that the method of [16] could not easily be adapted to give complexity bounds.

To get our result we analyze the structure of the feudal court. We have a few kings and kings are characterized by the fact that they are connected between themselves using only *counting types.* Instead of a more or less uniform court we have a hierarchy V_i for $i < 2^{n^2}$ of vassals, each of which may be a sovereign of perhaps several vassals in V_{i+1}. The union V of all V_i for $i < 2^{n^2}$ is included in the set of *kings* in the sense of [16] and provides information sufficient to reconstruct a model and thus gives rise to a 2-NEXPTIME algorithm for C^2_1 and a 3-NEXPTIME algorithm for C^2. Of course, we cannot easily improve the above bounds, since we can provide a sentence (see Proposition 4.18) of C^2_1 of size n whose unique model coincides with V and has cardinality $O(2^{2^n})$.

To push the lower bound down we had to provide a finer analysis. We have noticed that although the number of vassals in a model can be large (doubly exponential), the number of vassals that are different from the point of view of relations between themselves is smaller (exponential). In more technical terms a potential model is described by a set of indexed constellations and numbers of elements that realize these constellations. Roughly speaking, an indexed constellation—in addition to information on 2-types realized by pairs containing a given element—carries, for certain 2-types, *requests* for partner constellations, that is, constellations that should realize, together with the given constellation, these 2-types. Moreover, we show that the model is composed of some number of parts (only one of them can be infinite), which can be treated separately and independently during construction of the model. To check if the parts can be constructed we use several graph-theoretical results concerning the existence of Hamiltonian cycles, matchings, and bipartition.

It is worth noticing that by a recent result of Grädel, Otto, and Rosen [17], extensions of two-variable logic L^2 by a weak access to cardinalities through the Härtig (or equicardinality) quantifier is undecidable. The same is true for extensions of L^2 by very weak forms of recursion.

The satisfiability problem for logics with a bounded number of variables has applications in artificial intelligence, notably in modal logics (see, e.g., [22]) where counting comes in the context of graded modalities and in description logics, where counting can be used to express so-called number restrictions (see, e.g., [8]). More information on applications and relation of two-variables logics to modal logics is given in [15].

2. Preliminaries.

Throughout the paper we are concerned mainly with signatures that consist of unary and binary predicate letters without Boolean predicates, function symbols, and constants. This restriction allows us to simplify definitions and technical proofs. We would, however, like to emphasize that it is easy to adapt all notions used in this paper and to modify the proofs in order to obtain the same results also for the full first-order two-variable logic with counting, including predicate letters of higher arity and constants (see, e.g., [15] for a proof that predicate letters of higher arity can be eliminated).

We assume that the reader is familiar with standard notions of logic and with basic concepts of computational complexity theory. In this paper, $\mathcal{L}$-structures are denoted by Gothic capital letters and their universes are denoted by corresponding Latin capitals. Furthermore, if a structure $\mathfrak{A}$ is fixed, then its substructure with the universe denoted by a Latin capital is denoted by the corresponding Gothic capital.

By L^2 we denote the class of first-order sentences with two variables over a relational vocabulary, and by C^2_p we denote L^2 extended by additional quantifiers of the form $\exists^{=i}$, $\exists^{\leq i}$, or $\exists^{\geq i}$ (respectively, there exists exactly, at most, at least i) for $i \leq p$. Finally, C^2 is the union of C^2_p taken over all integers p.

Let $\mathcal{L}$ be a relational vocabulary with unary and binary predicate letters only. A 1-*type* $t(x)$ is a maximal consistent set of atomic and negated atomic formulas of the language $\mathcal{L}$ in the variable x. A 2-*type* $t(x, y)$ is a maximal consistent set of atomic and negated atomic formulas of the language $\mathcal{L}$ in the variables x, y, such that $(x \neq y) \in t(x, y)$. A type t is often identified with the conjunction of formulas in t. For a 2-type $t(x, y)$ we denote by $t(x, y)\restriction\{x\}$ the unique 1-type $t(x)$ included in

$t(x, y)$ and we denote by t^* the type dual to t; that is, the type obtained from t by replacing each occurrence of the variable x by y and each occurrence of y by x. If $\mathfrak{A}$ is an $\mathcal{L}$-structure with the universe A, and if $a, b \in A$, then we denote by $tp^{\mathfrak{A}}(a, b)$ the unique type realized by the pair $\langle a, b\rangle$ in $\mathfrak{A}$.

Recall that for any integer function $t(n)$, NTIME$(t(n))$ is the class of all decision problems that can be solved by a nondeterministic Turing machine in time $t(n)$, where n is the length of the input. We put

$$\text{NEXPTIME} = \bigcup\nolimits_p \text{NTIME } (2^{p(n)}),$$

$$\text{2-NEXPTIME} = \bigcup\nolimits_p \text{NTIME } (2^{2^{p(n)}}),$$

where p is a polynomial.

3. On the L^2case. In this section we consider the satisfiability problem for L^2, the first-order logic with two variables and without counting quantifiers. We give an algorithm solving this problem which runs in nondeterministic exponential time. As we have mentioned in the introduction, it follows from the paper of Mortimer [28] that the satisfiability problem for L^2 can be solved by a nondeterministic algorithm in doubly exponential time. An algorithm whose complexity matches the NEXPTIME lower bound given by Lewis [27] was presented in a very nice paper by Grädel, Kolaitis, and Vardi [15]. This algorithm and the bound that follow from Mortimer's work depend on the bounds on the cardinality of a minimal model of an L^2 sentence. Our algorithm, in contrast to the above, does not exploit the bounded model property of L^2.

This section is a modification of [36] and it is included here following a suggestion of one of the referees in order to introduce and explain the techniques used later for logic with counting.

Our approach in a remote way is based on Mortimer's notion of a *star* [28], a *star* being an arbitrary set of 2-types with a consistent *center*. The notion of a star was a very convenient technical tool to describe a finite structure and to check, with the help of Ehrenfeucht games of depth two [10], that this structure is a model of an L^2 sentence. Unfortunately, Mortimer's notion of a star cannot be directly used to give a NEXPTIME decision procedure since the cardinality of a star is exponential and the number of possible stars is doubly exponential in the number of predicate letters in the signature.

We weaken the notion of a star to a notion of a *small constellation* that we introduce after a close analysis of L^2 sentences from the point of view of their satisfiability. As in other related papers [31, 15] we use a variant of a notion of a normal form of first-order sentences. Our notion is called a *constellation form* and it allows to introduce the notion of a small constellation in a very natural way. Unlike a star, a small constellation is of linear size and it contains only these 2-types that describe a relation of a given point to a witness that must exist in a model of an L^2 sentence.

We also introduce a notion of a *small galaxy* as a set of small constellations that can be modeled in a first-order structure and we prove that an L^2-sentence is satisfiable if and only if there exists a small galaxy (Theorem 3.5). A small galaxy has only exponential size.

As the next step we give necessary and sufficient conditions for a set of small constellations to form a small galaxy (Definition 3.12, Theorem 3.13). In the proof of Theorem 3.13 we use notions of special and replicable constellations which are analogous to Mortimer's notions of asymmetric and symmetric stars. These notions are crucial for our analysis of models for L^2-sentences. As a result we get a nondeterministic exponential upper bound for the satisfiability problem for L^2-sentences (Corollary 3.14).

3.1. Small constellations, small galaxies, and satisfiability. Let $\mathcal{R} \subseteq \mathcal{L}$ be a set of binary predicate letters, $\mathcal{R} = \{R_1, \ldots, R_m\}$.

DEFINITION 3.1. *An $\mathcal{L}$-sentence Φ is in constellation form if*

$$\Phi = \forall x \forall y \phi(x,y) \wedge \bigwedge_{1 \le i \le m} \forall x \exists y R_i(x,y),$$

where ϕ is quantifier-free.

This definition may seem too strong. The second part of the formula seems to suggest that all elements are similar from the point of view of $\mathcal{R}$. Note, however, that we do not require that $x \neq y$; therefore, for an element x, by $R_i(x,x)$ we can code those relations R_i, for which the existential quantifier of the second part of Φ does not apply.

Let $\mathcal{A}$ be a set of 2-types closed under operation * and let $\mathcal{A}^+ = \{t \in \mathcal{A} : R_i(x,y) \in t, \text{ for some } i \le m\}$.

DEFINITION 3.2. *Let $S = \{s_0, s_1, \ldots, s_k\}$, where $0 \le k \le m$, s_0 is a* 1*-type and, if $k > 0$, then $s_1, \ldots, s_k \in \mathcal{A}^+$. Define center$(S) = \bigwedge_{0 \le i \le k} s_i \restriction \{x\}$. The set S is a small $\mathcal{A}$-$\mathcal{R}$-constellation if the following conditions hold:*

(1) *center$(S) = s_0$;*

(2) *for every $R_i \in \mathcal{R}$, if $R_i(x,x) \notin$ center(S), then there exists j, $1 \le j \le k$, such that $R_i(x,y) \in s_j$.*

Notice that the notion of a small $\mathcal{A}$-$\mathcal{R}$-constellation depends on a set $\mathcal{A}$ of 2-types and a set $\mathcal{R}$ of binary predicate symbols.

DEFINITION 3.3. *Let $\mathfrak{A}$ be an $\mathcal{L}$-structure. An element $a \in A$ realizes a small $\mathcal{A}$-$\mathcal{R}$-constellation $S = \{s_0, \ldots, s_k\}$ if $tp^{\mathfrak{A}}(a,a) = s_0$, for each $b \in A$, $tp^{\mathfrak{A}}(a,b) \in \mathcal{A}$, and there exists a sequence $b_1, \ldots, b_k$ of elements of A such that $tp^{\mathfrak{A}}(a,b_i) = s_i$, $0 < i \le k$.*

A small $\mathcal{A}$-$\mathcal{R}$-constellation S is realized *in $\mathfrak{A}$ if there exists $a \in A$ which realizes S.*

Note that if an element $a \in \mathfrak{A}$ realizes a small $\mathcal{A}$-$\mathcal{R}$-constellation, then $\mathfrak{A} \models \bigwedge_{1 \le i \le m} \exists y R_i(a,y)$.

DEFINITION 3.4. *Let $\mathcal{S}$ be a set of small $\mathcal{A}$-$\mathcal{R}$-constellations. A structure $\mathfrak{A}$ realizes $\mathcal{S}$ if every element $a \in A$ realizes a small $\mathcal{A}$-$\mathcal{R}$-constellation $S \in \mathcal{S}$, and every small $\mathcal{A}$-$\mathcal{R}$-constellation $S \in \mathcal{S}$ is realized by an element $a \in A$.*

The set $\mathcal{S}$ is a small galaxy if there is a structure $\mathfrak{A}$ such that card$(A) > 1$, and $\mathfrak{A}$ realizes $\mathcal{S}$.

The following theorem gives a necessary and sufficient condition for satisfiability of sentences in constellation form,

THEOREM 3.5. *Let $\mathcal{R} = \{R_1, \ldots, R_m\} \subseteq \mathcal{L}$ and let Φ be an $\mathcal{L}$-sentence in constellation form,*

$$\Phi = \forall x \forall y \phi(x,y) \wedge \bigwedge_{1 \le i \le m} \forall x \exists y R_i(x,y).$$

Put $\mathcal{A} = \{t : t(x,y)$ is a 2*-type over $\mathcal{L}$ and $t(x,y) \to \phi(x,y)\}$.*

Then Φ has a model with at least two-element universe if and only if there exists a set $\mathcal{S}$ of small $\mathcal{A}$-$\mathcal{R}$-constellations which is a small galaxy.

Proof. ($\Rightarrow$) Let $\mathfrak{A} \models \Phi$ and $card(A) > 1$. Since $\mathfrak{A} \models \forall x \forall y \phi(x, y)$, the set $\mathcal{A}$ is closed under the operation *. Moreover, $\mathfrak{A} \models \bigwedge_{1 \le i \le m} \forall x \exists y R_i(x, y)$ implies that every element of A realizes at least one small $\mathcal{A}$-$\mathcal{R}$-constellation. Let $\mathcal{S} = \{S : S$ is a small $\mathcal{A}$-$\mathcal{R}$-constellation and S is realized in $\mathfrak{A}\}$. By Definitions 3.4, $\mathfrak{A}$ realizes $\mathcal{S}$.

($\Leftarrow$) Let $\mathcal{S}$ be a small galaxy and assume $\mathfrak{A} \models \mathcal{S}$. Let $a \in A$. By Definition 3.4, a realizes a small $\mathcal{A}$-$\mathcal{R}$-constellation $S \in \mathcal{S}$, $S = \{s_0, \dots, s_k\}$. This implies that $tp^{\mathfrak{A}}(a, a) = s_0$, and there exists a sequence $b_1, \dots, b_k$ of distinct elements of A such that $b_i \neq a$ for $i = 1, \dots, k$ and

(1) $tp^{\mathfrak{A}}(a, b_i) = s_i$ for $i = 1, \dots, k$,

(2) $tp^{\mathfrak{A}}(a, b) \in \mathcal{A}$ for $b \in \mathcal{A}$.

Therefore, by Definition 3.2, the elements $b_1, \dots, b_k$ are witnesses of a for the part $\bigwedge_{1 \le i \le m} \exists y R_i(x, y)$ of Φ. Moreover, for every $b \in A, b \neq a$, $tp^{\mathfrak{A}}(a, b) \in \mathcal{A}$, and so $\mathfrak{A} \models \phi(a, b)$. Therefore $\mathfrak{A} \models \Phi$. $\Box$

3.2. The reduction. The following reduction theorem is essentially due to Scott [31]. It has been also used in [28] and [15]. We present a slightly modified version of the theorem given in [15].

THEOREM 3.6. *There exists a polynomial time algorithm which, given an L^2 sentence Ψ over an arbitrary relational vocabulary, constructs a sentence Φ in constellation form with the following properties:*

(1) *Ψ is satisfiable if and only if Φ is satisfiable.*

(2) *Every predicate letter occurring in Φ has arity at most 2.*

(3) *If n is the length of Ψ, then Φ contains $O(n)$ different predicate letters and has length $O(n \log n)$.*

3.3. The small galaxy theorem. In this section we fix $\mathcal{R} = \{R_1, \dots, R_m\} \subseteq \mathcal{L}$, and a set $\mathcal{A}$ of 2-types closed under the operation *.

To simplify terminology in this subsection we write "constellation" instead of "small $\mathcal{A}$-$\mathcal{R}$-constellation" and "galaxy" instead of "small galaxy."

By Theorem 3.6 and Theorem 3.5 the satisfiability problem for L^2 sentences can be reduced to the problem of finding an appropriate galaxy. In this subsection we shall give syntactic conditions that are necessary and sufficient for a set of constellations to be a galaxy.

DEFINITION 3.7. *Let S, T be constellations and let $t(x, y) \in \mathcal{A}$. S* is connectable to *T* by *$t(x, y)$ if $center(S) \subseteq t(x, y)$ and $center(T) \subseteq t^*(x, y)$.*

We say that S is connectable to *T if there is a type $t(x, y) \in \mathcal{A}$ such that S is connectable to T by $t(x, y)$.*

PROPOSITION 3.8. *Let $\mathcal{S}$ be a galaxy and $S \in \mathcal{S}$. Then the following conditions are equivalent.*

(1) *There is a structure $\mathfrak{A}$ which realizes $\mathcal{S}$, and S is realized in $\mathfrak{A}$ by at least two elements.*

(2) *S is connectable to S.*

Proof. Let $\mathcal{S}$ be a galaxy, $S \in \mathcal{S}$, and let $\mathfrak{B}$ be a structure such that $\mathfrak{B} \models \mathcal{S}$. To prove the implication (1) $\Rightarrow$ (2), assume that S is realized in $\mathfrak{A}$ by two elements a and b. Let $t(x, y) = tp^{\mathfrak{A}}(a, b)$. Since $\mathfrak{A} \models \mathcal{S}$, we have $t(x, y) \in \mathcal{A}$. Of course, $center(S) \subseteq t(x, y)$ and $center(S) \subseteq t^*(x, y)$.

Now we shall prove that (2) $\Rightarrow$ (1). Let $t \in \mathcal{A}$ be such that $center(S) \subseteq t(x, y)$ and $center(S) \subseteq t^*(x, y)$. Let $b \in B$ realize S in $\mathfrak{B}$. We claim that there exists an extension $\mathfrak{A}$ of $\mathfrak{B}$ such that $A = B \cup \{a\}$, where $a \notin B$ and a realizes S. Indeed, $\mathfrak{A}$ can be obtained from $\mathfrak{B}$ by putting $A = B \cup \{a\}$, $tp^{\mathfrak{A}}(a, b) = t(x, y)$, and $tp^{\mathfrak{A}}(a, c) = tp^{\mathfrak{A}}(b, c)$, for every $c \in B$. $\Box$

The proposition above motivates the following definition.

DEFINITION 3.9. *A constellation S is* replicable *if S is connectable to S. Otherwise, the constellation S is* special.

Let $\mathcal{S}$ be a galaxy, and let $\mathfrak{A}$ satisfy $\mathcal{S}$. In the universe A of $\mathfrak{A}$ we can distinguish the set $K \subseteq A$ consisting of all elements which realize special constellations. Elements of the set K are called *kings*. A *noble* is an element of the set $N = \bigcup_{a\in K}\{b_1, \ldots, b_k \in A : k \leq m \text{ and } \{tp^{\mathfrak{A}}(a), tp^{\mathfrak{A}}(a, b_1), \ldots, tp^{\mathfrak{A}}(a, b_k)\} \in \mathcal{S}\}$. Nobles are those elements of the universe that are necessary for the existence of kings. Define the *court* $C = K \cup N$. Note that $card(C) \leq (m+1)card(K)$. There may also be *plebeians*—elements outside the court; they are not necessary for the kings but perhaps some nobles may need them. Plebeians may also depend on kings to survive.

Remark. The notion of a king in a structure has been used in many places. For example, Gurevich and Shelah have used this notion in [20] to show that their proof of the solvability of the Gödel class without equality could not be generalized to the case with equality. Grädel, Kolaitis, and Vardi have also used this notion in [15]. We would like to point out that although in this paper the kings are defined in terms of constellations, they have the same meaning as in [15].

DEFINITION 3.10. *Denote by $Sp(\mathcal{S})$ the subset of $\mathcal{S}$ consisting of all special constellations and by $Rp(\mathcal{S})$ the set $\mathcal{S} \setminus Sp(\mathcal{S})$.*

The following simple observation establishes relations between the notions defined above.

PROPOSITION 3.11. *Let $\mathcal{S}$ be a galaxy, and let $\mathfrak{A}$ realize $\mathcal{S}$. Then there exist sets K and C such that the following conditions hold.*

(1) *$K \subseteq C \subseteq A$, $card(K) \leq card(Sp(\mathcal{S}))$, and $card(C) \leq (m + 1)card(K)$.*
(2) *Every element $a \in K$ realizes a constellation $S \in Sp(\mathcal{S})$ in $\mathfrak{A}{\restriction}C$.*
(3) *Every constellation $S \in Sp(\mathcal{S})$ is realized by an element $a \in K$ in $\mathfrak{A} \restriction C$.*
(4) *For every S, $T \in Rp(\mathcal{S})$, S is connectable to T.*

Proof. The proof is immediate. □

One can easily check that the converse to the above proposition does not hold. For example, let

$$\begin{aligned}
\mathcal{L} &= \mathcal{R} = \{R_1, R_2\},\\
\mathcal{A} &= \{t_1, t_1^*, t_2, t_2^*, t_3\},\\
s_0(x) &= R_1(x, x) \wedge R_2(x, x),\\
t_0(x) &= \neg R_1(x, x) \wedge \neg R_2(x, x),\\
t_1 &= t_0(x) \wedge R_1(x, y) \wedge \neg R_2(x, y) \wedge R_1(y, x) \wedge R_2(y, x) \wedge s_0(y),\\
t_2 &= t_0(x) \wedge \neg R_1(x, y) \wedge R_2(x, y) \wedge R_1(y, x) \wedge R_2(y, x) \wedge s_0(y),\\
t_3 &= t_0(x) \wedge \neg R_1(x, y) \wedge \neg R_2(x, y) \wedge \neg R_1(y, x) \wedge \neg R_2(y, x) \wedge t_0(y),\\
\mathcal{S} &= \{S, T\}, \text{ where } S = \{s_0\} \text{ and } T = \{t_0, t_1, t_2\}.
\end{aligned}$$

It is easy to see that the constellation S is special and T is replicable. One can check that if we define $K = \{a\}$, $C = K$, $tp^{\mathfrak{A}}(a) = s_0$, then conditions (1)–(3) of Proposition 3.11 hold and, since T is connectable to T by t_3, condition (4) holds too. Unfortunately, the constellation T cannot be realized in any structure, since to realize T we need two elements, b_1 and b_2, such that $tp^{\mathfrak{A}}(b_1) = s_0$, $tp^{\mathfrak{A}}(b_2) = s_0$, and $tp^{\mathfrak{A}}(b_1, b_2) \in \mathcal{A}$, which is not possible. Therefore, $\mathcal{S}$ is not a galaxy.

Now we shall extend the set of conditions given in Proposition 3.11 to a set of conditions that will imply that a set $\mathcal{S}$ of constellations is a galaxy.

DEFINITION 3.12. *Let $\mathcal{S}$ be a set of constellations. A* small representation *of $\mathcal{S}$ is a system*

$$\langle K, C, I, F, G\rangle,$$

where K and C are sets, I, F, G are functions such that $I : C \to \mathcal{S}$, $F : C \times C \to \mathcal{A}$, $G : Rp(\mathcal{S}) \times K \to \mathcal{A}$, and the following conditions hold.

(s1) *$K \subseteq C$, $card(K) \leq card(Sp(\mathcal{S}))$, and $card(C) \leq (m+1)card(K)$.*

(s2) *$I(K) = Sp(\mathcal{S}), I(C \setminus K) \subseteq Rp(\mathcal{S})$, $F(a,a) = F(a,b)\restriction\{x\}$, and for every $a \neq b$ $F(a,b) = F(b,a)^*$.*

(s3) *For every $a \in K$ and every $t \in I(a)$, there is an element $c \in C$ such that $t = F(a,c)$.*

(s4) *For every $b \in C \setminus K$ and every $t \in I(b)$, if there is no $c \in C$ such that $t = F(b,c)$, then there is a constellation $T \in Rp(\mathcal{S})$ such that $I(b)$ is connectable to T by t.*

(s5) *For every S, $T \in Rp(\mathcal{S})$, S is connectable to T.*

(s6) *For every $S \in Rp(\mathcal{S})$ and every $a \in K$, S is connectable to $I(a)$ by $G(S,a)$.*

(s7) *For every $S \in Rp(\mathcal{S})$ and every type $t(x,y) \in S$, if there is no $a \in K$ such that $G(S,a) = t(x,y)$, then there is a constellation $T \in Rp(\mathcal{S})$ such that S is connectable to T by t.*

Conditions (s1), (s2), and (s3) say that the set C is a universe of a structure in which all special constellations are realized. In other words, kings are provided with all they need to survive. Condition (s4) ensures that every noble can find enough plebeians around him. Condition (s5) says that plebeians can live together in one society and, by condition (s6), the society is ruled by kings. Condition (s7) states that plebeians can get what they need—if not from kings, then from somewhere else.

THEOREM 3.13 (small galaxy theorem). *A set of constellations $\mathcal{S}$ is a galaxy if and only if there exists a small representation of $\mathcal{S}$.*

Proof. ($\Rightarrow$) Assume that $\mathcal{S}$ is a galaxy and $\mathfrak{A}$ realizes $\mathcal{S}$. Let K be the set of kings in $\mathfrak{A}$, and let C be the court in $\mathfrak{A}$. For every $a \in C$ choose a constellation $S \in \mathcal{S}$ which is realized by a, and put $I(a) = S$. For every $a, b \in C$, put $F(a,b) = tp^{\mathfrak{A}}(a,b)$. For every constellation $S \in Rp(\mathcal{S})$ find an element $b \in A$ which realizes S, and for every $a \in K$ put $G(S,a) = tp^{\mathfrak{A}}(b,a)$.

It is easy to check that the system $\langle K, C, I, F, G\rangle$ is a small representation of $\mathcal{S}$.

($\Leftarrow$) Let $\mathcal{S}$ be a set of constellations, and let $\langle K, C, I, F, G\rangle$ be a small representation of $\mathcal{S}$.

We shall construct a structure $\mathfrak{A}$ realizing $\mathcal{S}$ such that the universe A of $\mathfrak{A}$ contains C, every $a \in K$ realizes $I(a)$ in $\mathfrak{A}\restriction C$, and $tp^{\mathfrak{A}}(a,b) = F(a,b)$, for each pair $\langle a, b\rangle$ of elements of C.

The construction proceeds in steps. The number of steps can be infinite. In each step new elements are added to the universe. A new element is added when there is a request to satisfy a constellation, say, S. Whenever an element a is added to satisfy S, I is extended by putting $I(a) = S$. An element a such that $I(a) = S$ is *inactivated* after adding enough elements to witness that a realizes S. An unordered pair of elements will be *reserved*, when a type to be realized by this pair has been designated.

In every step of the construction the universe of the part of the structure $\mathfrak{A}$ defined so far is finite. We also assume that there is a fixed linear ordering $<$ of the universe, and each new element added to the universe is greater than all old elements.

Let $\mathcal{S} = \{S_1, \ldots, S_k\}$.

Stage 1.

(1) Let $A = C$.
(2) For every $a, b \in C$, put $tp^{\mathfrak{A}}(a, b) = F(a, b)$ (cf. (s2)).
(3) For every $a \in K$, inactivate a.
(4) For every $a, b \in C$, reserve $\{a, b\}$.
(5) For every $S \in Rp(\mathcal{S})$ such that $I(b) \neq S$, for each $b \in C$, add a new element d to A and put $I(d) = S$.

Stage 2.

(6) Let b be the first active (i.e., yet not inactivated) element of A.
Note that $I(b) \in Rp(\mathcal{S})$. Indeed, $b \notin K$, so either $b \in C \setminus K$, and so $I(b) \in Rp(\mathcal{S})$, by (s2), or b has been added to A in steps 5, 7(a)(ii), or 9(b), and whenever we add a new element b to the universe we always put $I(b) \in Rp(\mathcal{S})$.
(7) If $b \in C \setminus K$, then
  (a) for every $t \in I(b)$ if there is no element $c \in C$ such that $t = F(b, c)$, do
    (i) using (s4) find $T \in Rp(\mathcal{S})$ such that $I(b)$ is connectable to T by t;
    (ii) add a new element d to A;
    (iii) put $I(d) = T$, and $tp^{\mathfrak{A}}(b, d) = t$, reserve $\{b, d\}$;
  (b) inactivate b and go to 6.
(8) Using (s6), for every $a \in K$, put $tp^{\mathfrak{A}}(b, a) = G(I(b), a)$.
Note that, $b \notin C$, so no pair $\{a, b\}$, with $a \in K$, has been reserved earlier.
(9) For every $t \in I(b)$, if there is no $c \in A$ such that $\{c, b\}$ is reserved and $tp^{\mathfrak{A}}(c, b) = t^*$, do
  (a) by (s7) find $T \in Rp(\mathcal{S})$ such that $I(b)$ is connectable to T by t.
  Note that, by step 8, there is no element $a \in K$; such that $t = G(I(b), a)$.
  (b) add a new element d to A;
  (c) put $I(d) = T$, $tp^{\mathfrak{A}}(b, d) = t$, and reserve $\{b, d\}$;
  (d) for every $a < b$, if $\{a, b\}$ is not reserved, then using (s5) find $t \in \mathcal{A}$ such that $I(b)$ is connectable to $I(a)$ by t, put $tp^{\mathfrak{A}}(b, a) = t$ and reserve $\{b, a\}$.
(10) Inactivate b and go to 6.

We shall now show that $\mathfrak{A}$ realizes $\mathcal{S}$. First, let us note that every pair of distinct elements of A realizes in $\mathfrak{A}$ a 2-type of $\mathcal{A}$ (see steps 2, 7(a)(iii), 8, 9(c), 9(d)).

New elements are added at the end of the fixed ordering, and in step 6 we always consider the first active element; therefore every element $a \in A$ will eventually be inactivated. We claim that when an element a is inactivated, then a realizes a constellation of $\mathcal{S}$ in $\mathfrak{A}$. In fact an element inactivated in step 3, by (s2) and (s3) of Definition 3.12, realizes a special constellation of $\mathcal{S}$. Before an element b is inactivated in step 7(b), in step 7(a) every type of $I(b)$ has been realized by a pair $\langle b, a \rangle$ for some $a \in A$. Similarly, step 9 ensures that the element inactivated in step 10 realizes its constellation.

Finally, by step 5 every constellation of $\mathcal{S}$ is realized in $\mathfrak{A}$. $\square$

Now let us consider the cardinality of the structure constructed by the algorithm described above. If $Rp(\mathcal{S}) = \emptyset$, then only the first stage of the algorithm is performed and we get a structure with the universe K. We also get a finite structure if no new elements are added in step 7(a)(i). In this case $I(C) = \mathcal{S}$, and the function F is defined in such a way that no noble element needs a plebeian. In other cases we get an infinite structure. The construction could be modified in such a way that it will stop after a bounded number of steps. However, we omit this modification, since the construction described above is better suited for generalization to logic with counting.

3.4. Complexity.

Corollary 3.14. *There is a nondeterministic algorithm with time complexity $O(2^{cn^2})$, for some constant c, which, given an L^2-sentence Φ, decides if Φ is satisfiable.*

Remark. In [36], using more complicated techniques, we gave a similar algorithm with time complexity $O(2^{cn})$. Here we provide a simplified version only, since it is easier to understand and better explains the methods used in the main part of this paper.

Proof. Let Φ be an L^2 sentence of length n. In the first step we use the polynomial time algorithm of Theorem 3.6 to get a sentence Ψ in constellation form,

$$\Psi = \forall x \forall y \phi(x, y) \wedge \bigwedge_{1 \le i \le m} \forall x \exists y R_i(x, y),$$

which is satisfiable if and only if Φ is satisfiable. Moreover, Ψ has at most $p = O(n)$ predicate letters and has length $O(n \log n)$.

Then we use Theorem 3.5. We build a set $\mathcal{A}$ in time $O(2^{O(p)})$, and, in time $O((2^{4p})^m) = O(2^{O(n^2)})$, we guess a set $\mathcal{S}$ of small $\mathcal{A}$-$\mathcal{R}$-constellations.

Next, we use Theorem 3.12 and we guess sets K and C and functions I, F, and G. Since $card(K) \le 2^{O(n^2)}$ and $card(C) \le (m+1)card(K)$ we can do this in time $O(2^{O(n^2)})$.

Finally, we accept Ψ after checking whether $\langle K, C, I, F, G\rangle$ is a small representation of $\mathcal{S}$. This can also be done in time $O(2^{O(n^2)})$. □

4. Double exponential algorithm.

4.1. Constellations, galaxies, and satisfiability.

Let $\mathcal{R} = \{R_1, \dots, R_m\} \subseteq \mathcal{L}$ be the set of binary predicate letters.

Definition 4.1. *An $\mathcal{L}$-sentence Φ is in* constellation form *if*

$$\Phi = \forall x \forall y \phi(x, y) \wedge \bigwedge_{1 \le i \le m} \forall x \exists^{=m_i} y R_i(x, y),$$

where ϕ is quantifier-free. Φ is in $\exists^{=1}$-constellation form *if $m_i = 1$ for each $i \le m$.*

As Definition 3.1, the definition above may seem too strong, since the second part of the formula seems to suggest that all elements are similar from the point of view of $\mathcal{R}$. However, as before, the fact whether $R_i(x, x)$ holds is used to code those relations R_i for which the counting quantifier does not apply.

Let $\mathcal{A}$ be a set of 2-types closed under operation *.

Definition 4.2.

$\mathcal{A} = \mathcal{A}^{\leftrightarrow} \dot\cup \mathcal{A}^{\leftarrow} \dot\cup \mathcal{A}^{\rightarrow} \dot\cup \mathcal{A}^{-}$, *where*

$\mathcal{A}^{\leftrightarrow} = \{t \in \mathcal{A} :$ *there are* $i, j \le m$ *such that* $R_i(x, y) \in t$ *and* $R_j(y, x) \in t\}$,

$\mathcal{A}^{\leftarrow} = \{t \in \mathcal{A} : t \notin \mathcal{A}^{\leftrightarrow}$ *and there exists* $i \le m$ *such that* $R_i(y, x) \in t\}$,

$\mathcal{A}^{\rightarrow} = \{t \in \mathcal{A} : t \notin \mathcal{A}^{\leftrightarrow}$ *and there exists* $i \le m$ *such that* $R_i(x, y) \in t\}$,

$\mathcal{A}^{-} = \{t \in \mathcal{A} :$ *for every* $i \le m, \neg R_i(x, y) \in t$ *and* $\neg R_i(y, x) \in t\}$.

In the definition above $\mathcal{A}^{\leftrightarrow}$, $\mathcal{A}^{\leftarrow}$, and $\mathcal{A}^{\rightarrow}$ represent *counting types.* Since R_i appears in the second part of the formula Φ in constellation form, it follows that $R_i(x, y) \in t$ implies that for (a, b) and (a, b') realizing t we always have $b = b'$.

Definition 4.3. *Let $S = \{s_0, s_1, \dots, s_k\}$, where $k \ge 0$, s_0 is a 1-type and, if $k > 0$, then $s_1, \dots, s_k \in \mathcal{A}^{\leftrightarrow} \cup \mathcal{A}^{\rightarrow}$. Define $center(S) = \bigwedge_{0 \le i \le k} s_i \restriction \{x\}$. The set S is an* $\mathcal{A}$-$\mathcal{R}$-constellation *if the following conditions hold:*

(1) *$center(S) = s_0$;*
(2) *for every $R_i \in \mathcal{R}$, if $R_i(x,x) \notin center(S)$, then there is exactly one j, $1 \leq j \leq k$, such that $R_i(x,y) \in s_j$;*
(3) *for every $R_i \in \mathcal{R}$, if $R_i(x,x) \in center(S)$, then for every j, $1 \leq j \leq k$, $R_i(x,y) \notin s_j$.*

Notice that the notion of an $\mathcal{A}$-$\mathcal{R}$-constellation depends on fixed sets $\mathcal{A}$ of 2-types and $\mathcal{R}$ of binary predicate symbols. Moreover, the number of 2-types in a constellation does not exceed $card(\mathcal{R})$. It does not follow from Definition 4.3 that each constellation contains a counting type. There may be constellations S such that $center(S) = \{R_i(x,x) : R_i \in \mathcal{R}\}$. In fact $center(S)$ codes the relations in $\mathcal{R}$ which are not used in S in the context of counting.

DEFINITION 4.4. *Let $\mathfrak{A}$ be an $\mathcal{L}$-structure. An element $a \in A$* realizes *an $\mathcal{A}$-$\mathcal{R}$-constellation $S = \{s_0, \ldots, s_k\}$ if $tp^{\mathfrak{A}}(a,a) = s_0$, and there exists a unique sequence $b_1, \ldots, b_k \in A$ that $tp^{\mathfrak{A}}(a,b_i) = s_i$, $1 \leq i \leq k$, and for every $b \in A$, $b \neq a$, $b \neq b_i$, $1 \leq i \leq k$, we have $tp^{\mathfrak{A}}(a,b) \in \mathcal{A}^{\leftarrow} \cup \mathcal{A}^{-}$. An $\mathcal{A}$-$\mathcal{R}$-constellation S is* realized *in $\mathfrak{A}$ if there exists an element $a \in A$ which realizes S.*

For $a \in A$, we write $C_a^{\mathfrak{A}}$ to denote the unique $\mathcal{A}$-$\mathcal{R}$-constellation realized by a.

DEFINITION 4.5. *Let $\mathcal{S}$ be a set of $\mathcal{A}$-$\mathcal{R}$-constellations. A structure $\mathfrak{A}$* realizes *$\mathcal{S}$ if every element in A realizes an $\mathcal{A}$-$\mathcal{R}$-constellation and every constellation in $\mathcal{S}$ is realized in $\mathfrak{A}$.*

A set $\mathcal{S}$ of $\mathcal{A}$-$\mathcal{R}$-constellations is a galaxy *if there is a structure $\mathfrak{A}$ such that $card(A) > 1$, and $\mathfrak{A}$ realizes $\mathcal{S}$.*

The following theorem gives a necessary and sufficient condition for satisfiability of sentences in $\exists^{=1}$-constellation form.

THEOREM 4.6. *Let $\mathcal{R} = \{R_1, \ldots, R_m\} \subseteq \mathcal{L}$, and let Φ be an $\mathcal{L}$-sentence in $\exists^{=1}$-constellation form,*

$$\Phi = \forall x \forall y \phi(x,y) \wedge \bigwedge_{1 \leq i \leq m} \forall x \exists^{=1} y R_i(x,y).$$

Put $\mathcal{A} = \{t : t(x,y)$ is a 2-type over $\mathcal{L}$ and $t(x,y) \rightarrow \phi(x,y)\}$.

Then Φ has a model with at least two-elements if and only if there exists a set of $\mathcal{A}$-$\mathcal{R}$-constellations which is a galaxy.

Proof. ($\Rightarrow$) Assume that $\mathfrak{A} \models \Phi$ and $card(A) > 1$. Since $\mathfrak{A} \models \forall x \forall y \phi(x,y)$, the set $\mathcal{A}$ is closed under *. Since $\mathfrak{A} \models \bigwedge_{1 \leq i \leq m} \forall x \exists^{=1} y R_i(x,y)$, every element of A realizes some $\mathcal{A}$-$\mathcal{R}$-constellation. Therefore $\mathcal{S} = \{C_a^{\mathfrak{A}} : a \in A\}$ is a galaxy.

($\Leftarrow$) Let $\mathcal{S}$ be a galaxy and assume that $\mathfrak{A}$ realizes $\mathcal{S}$. Let $a \in A$. By Definition 4.5, a realizes an $\mathcal{A}$-$\mathcal{R}$-constellation $S \in \mathcal{S}$, $S = \{s_0, \ldots, s_k\}$. This implies that $tp^{\mathfrak{A}}(a,a) = s_0$ and that there exists a sequence $b_1, \ldots, b_k$ of distinct elements of A such that $b_i \neq a$ for $i = 1, \ldots, k$, and

(1) $tp^{\mathfrak{A}}(a,b_i) = s_i$, for $i = 1, \ldots, k$,
(2) $tp^{\mathfrak{A}}(a,b) \in \mathcal{A}^{\leftarrow} \cup \mathcal{A}^{-} \subset \mathcal{A}$, for each $b \in \mathcal{A}$, such that $b \neq a$ and $b \neq b_i$, $i = 1, \ldots, k$.

Therefore, by Definition 4.3, the elements $b_1, \ldots, b_k$ are witnesses of a for the part

$$\bigwedge_{1 \leq i \leq m} \exists^{=1} y R_i(x,y).$$

Moreover, for every $b \in A, b \neq a$, $tp^{\mathfrak{A}}(a,b) \in \mathcal{A}$, and so $\mathfrak{A} \models \phi(a,b)$. Hence, $\mathfrak{A} \models \Phi$. ▯

4.2. The reduction. The following theorem has been shown in [16].

THEOREM 4.7. *There is a recursive reduction* NF *from C^2-sentences to C^2-sentences in normal form over an extended vocabulary, which is sound for satisfiability: Φ is satisfiable if and only if* $\mathrm{NF}(\Phi)$ *is satisfiable.*

In the above theorem the normal form is slightly weaker than our $\exists^{=1}$-constellation form. The difference is that in the $\exists^{=1}$-constellation form the quantifier free part of the sentences with prefix $\forall\exists^{=1}$ is atomic, whereas in the normal form in the sense of [16] it could be any quantifier free two-variable formula. This additional condition can be easily met by introducing new relation symbols for quantifier free formulas, and adding $\forall\forall$-sentences defining the newly introduced symbols.

From the proof of Theorem 4.7 given in [16] the following corollaries can be derived.

COROLLARY 4.8. *There exists a polynomial time algorithm which, given a C^2_1 sentence Ψ over a relational vocabulary, constructs a sentence Φ in $\exists^{=1}$-constellation form with the following properties:*

(1) *Ψ is satisfiable if and only if Φ is satisfiable.*
(2) *Every predicate letter occurring in Φ has arity at most* 2.
(3) *If n is the length of Ψ, then Φ contains $O(n)$ different predicate letters and has length $O(n \log n)$.*

The reduction for the full logic C^2 is more expensive.

COROLLARY 4.9. *There exists an exponential time algorithm which, given a C^2 sentence Ψ over a relational vocabulary, constructs a sentence Φ in $\exists^{=1}$-constellation form with the following properties:*

(1) *Ψ is satisfiable if and only if Φ is satisfiable.*
(2) *Every predicate letter occurring in Φ has arity at most* 2.
(3) *If n is the length of Ψ, then Φ contains $O(2^n)$ different predicate letters and has length $O(2^{O(n)})$.*

The exponential increase of the length of the sentence Φ given by the algorithm in Corollary 4.9 is caused by the necessity to introduce as many new predicate letters as the maximal integer which appear as an index of a counting quantifier. If integers are represented in binary we have to introduce $O(2^n)$ new predicate letters for a sentence of length n. We do not know any better reduction and this is the main reason why we cannot improve the upper complexity bound for the satisfiability problem for the full C^2 from double to single exponential.

4.3. The galaxy theorem. In this subsection we fix $\mathcal{R} = \{R_1, \ldots, R_m\} \subseteq \mathcal{L}$ and a set $\mathcal{A}$ of 2-types closed under *. Henceforth, whenever the sets $\mathcal{A}$ and $\mathcal{R}$ are fixed, we write "a constellation" instead of "an $\mathcal{A}$-$\mathcal{R}$-constellation."

So far we have shown that the satisfiability problem for C^2_1 sentences can be transformed to the problem of finding an appropriate galaxy. In this section we shall formulate syntactic conditions which are necessary and sufficient for a set of constellations to form a galaxy, but before doing that, in order to acquaint the reader with our basic technique and to provide a better background for the proof of the main result of this section, we shall state and prove some basic properties of constellations.

At the beginning we introduce a syntactic notion of *connectability* of two constellations. This definition says that two constellations are connectable by a 2-type t if t is a connective type for them; that is, t contains the centers of both constellations and either t is noncounting, or t and t^* are distributed between these two constellations. In other words, this notion provides a necessary condition for two constellations to be realizable in the same structure.

The notion of connectability, together with the notions of constellation and galaxy, plays a crucial role in this section and is basic in the whole paper. An easy observation (Proposition 4.13) shows that in a specific situation this notion suffices to formulate very simple conditions that allows to solve the satisfiability problem. This "specific" situation can be described in both semantic and syntactic terms. There is a structure in which every constellation is realized infinitely many times, or every two constellations are connectable by a noncounting type. Intuitively it means that there are no privileged elements in the structure.

Next, we consider the case when there are some privileged elements in a model. We prove that if a constellation S is realized in a structure sufficiently often, then we can build a structure in which S appears infinitely many times (Lemma 4.2). Constellations that can be realized infinitely often are easy to deal with, in contrast with those that always appear only finitely many times.

Lemma 4.3 plays a crucial role in the proof of the main result of this section—the galaxy theorem. It says that every galaxy can be partitioned into two sets: constellations which can be realized by at most r elements and constellations which can be realized by infinitely many elements. The integer r is bounded by an exponential function of the number of constellations in a given galaxy. To prove this lemma, for a structure $\mathfrak{A}$ realizing the given galaxy, we define a sequence of sets $V_1 \subset V_2 \subset \cdots \subset V_{p-1}$ of subsets of A. The set V_1 consists of lords, that is, of elements of A which realize constellations appearing in $\mathfrak{A}$ very rarely—less than $2m + 1$ times. Every set V_{i+1}, for $i > 1$, besides members of V_i, contains elements that are vassals of elements of V_i. They realize constellations appearing in $\mathfrak{A}$ not very often with respect to the cardinality of the set V_i of sovereigns of the elements of V_{i+1}. In this way we obtain a finite hierarchy of elements of A and therefore a hierarchy of elements of the galaxy—constellations realized by elements of appropriate V_i. This hierarchy does not necessarily include all constellations.

All the results mentioned above give several necessary conditions for a set of constellations to be a galaxy. As the next step we introduce the notion of a *finite representation of* a set of constellations (Definition 4.14), and we prove the galaxy theorem (Theorem 4.15) which says that the problem whether a set of constellations $\mathcal{S}$ is a galaxy can be reduced to the problem whether there exists a finite representation of $\mathcal{S}$.

Since the components of a finite representation are either finite sets of bounded cardinality or functions from such sets into some fixed finite sets, and since the conditions on the components are easily[1] computable, the galaxy theorem forms a basis for a decision procedure for the satisfiability problem for C_1^2 (Corollary 4.17 in 4.4).

We begin the technical part of this section with some additional definitions.

DEFINITION 4.10. *Let S, T be constellations, and let $t(x, y) \in \mathcal{A}$. S* is connectable *to T* by *$t(x, y)$ if $center(S) \subseteq t(x, y)$, $center(T) \subseteq t^*(x, y)$, and*

(1) *$t \in S$ and $t^* \in T$ if $t \in \mathcal{A}^{\leftrightarrow}$,*

(2) *$t \in S$ if $t \in \mathcal{A}^{\rightarrow}$,*

(3) *$t^* \in T$ if $t \in \mathcal{A}^{\leftarrow}$.*

DEFINITION 4.11. *Let $\mathcal{S}$ be a galaxy, and assume that $\mathfrak{A}$ realizes $\mathcal{S}$. Define a function* $\mathrm{rank}_{\mathfrak{A}} : \mathcal{S} \to \mathbb{N} \cup \{\infty\}$ *putting* $\mathrm{rank}_{\mathfrak{A}}(S) = card(\{a \in A : C_a^{\mathfrak{A}} = S\})$*.*

We write $\mathrm{rank}(\mathcal{S}) = \infty$ *if there is a structure $\mathfrak{B}$ realizing $\mathcal{S}$ such that*

$$\min_{S \in \mathcal{S}}(\mathrm{rank}_{\mathfrak{B}}(S)) > 2m + 1.$$

[1]In this section "easily" means in double exponential time.

LEMMA 4.1. *Assume that $\mathfrak{A}$ is a structure which realizes $\mathcal{S}$. Let $S, T \in \mathcal{S}$. If* $\mathrm{rank}_{\mathfrak{A}}(S) > 2m+1$, *and* $\mathrm{rank}_{\mathfrak{A}}(T) > 2m+1$, *then there exist $a, b \in A$ such that $C_a^{\mathfrak{A}} = S$, $C_b^{\mathfrak{A}} = T$, and $tp^{\mathfrak{A}}(a,b) \in \mathcal{A}^-$.*

Proof. Let $X = \{a \in A : C_a^{\mathfrak{A}} = S\}$, $Y = \{a \in A : C_a^{\mathfrak{A}} = T\}$, and assume that $card(X) \geq card(Y) > 2m+1$.

By Definition 4.4, for every $a \in X$ ($a \in Y$) there is at most m distinct elements b such that $tp^{\mathfrak{A}}(a,b) \in S$ ($tp^{\mathfrak{A}}(a,b) \in T$, respectively). Therefore, the number of pairs $\langle a, b\rangle$ such that $a \in X$, $b \in Y$, and $tp^{\mathfrak{A}}(a,b) \in S$ or $tp^{\mathfrak{A}}(b,a) \in T$ does not exceed $m \cdot card(X) + m \cdot card(Y) \leq 2m \cdot card(X)$. On the other hand, the number of pairs $\langle a, b\rangle$ such that $a \in X$ and $b \in Y$ is $card(X) \cdot card(Y) > (2m+1)card(X)$. ☐

COROLLARY 4.12. *Let $\mathcal{S}$ be a galaxy with* $\mathrm{rank}(\mathcal{S}) = \infty$. *Then there exists a structure $\mathfrak{B}$ realizing $\mathcal{S}$ such that* $\mathrm{rank}_{\mathfrak{B}}(S) = \infty$, *for each $S \in \mathcal{S}$.*

Proof. Assume that $\mathfrak{A}$ realizes $\mathcal{S}$, $S \in \mathcal{S}$, and $\mathrm{rank}_{\mathfrak{A}}(S) = n$. We claim that there is an extension $\mathfrak{B}$ of $\mathfrak{A}$, such that $\mathfrak{B}$ realizes $\mathcal{S}$, and $\mathrm{rank}_{\mathfrak{B}}(S) > n$. By Lemma 4.1, for all constellations $S, T \in \mathcal{S}$, there exists a type $t(x,y) \in \mathcal{A}^-$ such that S is connectable to T by $t(x,y)$. Let $\mathfrak{A}'$ be a structure isomorphic to $\mathfrak{A}$ such that $A \cap A' = \emptyset$. Define $B = A \cup A'$, and let $\mathfrak{B} \restriction A = \mathfrak{A}$, $\mathfrak{B} \restriction A' = \mathfrak{A}'$. Now, for every $a \in A$, and every $a' \in A'$, find $t(x,y) \in \mathcal{A}^-$ such that $C_a^{\mathfrak{A}}$ is connectable to $C_{a'}^{\mathfrak{A}'}$ by $t(x,y)$, and put $tp^{\mathfrak{B}}(a,a') = t(x,y)$. ☐

PROPOSITION 4.13. *The following conditions are equivalent:*

(1) $\mathrm{rank}(\mathcal{S}) = \infty$;

(2) (a) *for every $S \in \mathcal{S}$, and every $s(x,y) \in S$, there exists $T \in \mathcal{S}$ such that S is connectable to T by $s(x,y)$;*

(b) *for every $S, T \in \mathcal{S}$, S is connectable to T by some $t(x,y) \in \mathcal{A}^-$.*

Proof. (1) $\Rightarrow$ (2). Condition (a) follows from Definition 4.5 and condition (b) from Lemma 4.1.

(2) $\Rightarrow$ (1). We shall give an algorithm which constructs a structure $\mathfrak{A}$ realizing $\mathcal{S}$. In the process of construction new elements will be added to the universe; some elements of the universe will be inactivated and some unordered pairs of elements will be reserved. An element a will be inactivated when the constellation S that had been earlier assigned to a has been built, i.e., when elements which witness that a realizes S have been added. An unordered pair $\{a,b\}$ will be reserved, when the type realized by $\{a,b\}$ has been defined. Moreover, a function $I : A \mapsto \mathcal{S}$ will be defined in such a way that $I(a) = C_a^{\mathfrak{A}}$ for every $a \in A$.

In every step of the construction the part of the model defined so far $\mathfrak{A}$ will be finite. We assume that a linear ordering $<$ of the universe is given such that a new element added to the universe is always greater then the old elements.

Let $\mathcal{S} = \{S_1, \ldots, S_k\}$.

(1) Let $A = \{a_1, \ldots, a_k\}$. Put $I(a_i) = S_i$, $i = 1, \ldots, k$.
(2) Let $a \in A$ be the first active (not yet inactivated) element.
(3) For every $t_i \in I(a)$, if there is no element $b \in A$ such that $\{a,b\}$ is reserved, and $tp^{\mathfrak{A}}(a,b) = t_i$, then
    (a) add a new element b_i to A,
    (b) put $tp^{\mathfrak{A}}(a,b_i) = t_i$,
    (c) find a constellation $T \in \mathcal{S}$ such that S is connectable to T by t_i,
    (d) put $I(b_i) = T$, and reserve $\{a, b_i\}$.
(4) For every $c < a$ put $tp^{\mathfrak{A}}(a,c) = t \in \mathcal{A}^-$ such that $I(a)$ is connectable to $I(c)$ by t.
(5) Inactivate a.

(6) Go to 2.

If $\min_{S\in\mathcal{S}}(\mathrm{rank}_{\mathfrak{A}}(S)) \leq 2m+1$, perform the operations from the proof of Corollary 4.12. ◻

LEMMA 4.2. *Assume that $\mathfrak{A}$ realizes $\mathcal{S}$. Let V be a finite subset of A and let $\mathcal{S}' = \{C_a^{\mathfrak{A}} : a \in A\setminus V\}$. If $\mathcal{S}'\cap\{C_a^{\mathfrak{A}} : a\in V\} = \emptyset$, and for every $a \in A\setminus V$, $\mathrm{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) > \max(card(V)\cdot m, 2m+1)$, then there is a structure $\mathfrak{B}$ realizing $\mathcal{S}$, such that for every $S\in\mathcal{S}'$ $\mathrm{rank}_{\mathfrak{B}}(S)=\infty$, and $\mathrm{rank}_{\mathfrak{B}}(S) = \mathrm{rank}_{\mathfrak{A}}(S)$, for every $S\in\mathcal{S}\setminus\mathcal{S}'$.*

Proof. Let $\mathfrak{A}$ realize $\mathcal{S}$.

An iterative application of the following algorithm applied to every $S\in\mathcal{S}'$ yields a structure $\mathfrak{B}$ such that $\mathrm{rank}_{\mathfrak{B}}(S)=\infty$ for every $S\in\mathcal{S}'$ and $\mathrm{rank}_{\mathfrak{B}}(S)=\mathrm{rank}_{\mathfrak{A}}(S)$ for every $S\in\mathcal{S}\setminus\mathcal{S}'$ (inactivation, reserving elements, and the function I play the same role as in the proof of Proposition 4.13). At the beginning, put $\mathfrak{A}'=\mathfrak{A}$.

```
(1) Let $A'' = A'\cup\{x\}$.
(2) For every $a\in A'$ put $I(a) = C_a^{\mathfrak{A}'}$ and inactivate $a$.
(3) Put $I(x) = S$.
(4) Let $x$ be the first active element of $A''$.
(5) Find $a\in A\setminus V$ such that $C_a^{\mathfrak{A}} = I(x)$, and $tp^{\mathfrak{A}}(a,b)\in\mathcal{A}^{-}\cup\mathcal{A}^{\rightarrow}$, for every $b\in V$.
    Such an element $a$ exists, since there are at most $m\cdot card(V)$ elements $c\in A\setminus V$ such that $tp^{\mathfrak{A}}(c,b)\in\mathcal{A}^{\leftrightarrow}\cup\mathcal{A}^{\leftarrow}$.
(6) For every $t_i\in S$,
    (a) if there is $b\in V$ such that $tp^{\mathfrak{A}}(a,b)=t_i$, then put $tp^{\mathfrak{A}}(x,b)=t_i$, and reserve $\{x,b\}$
        else
    (b) if there is no element $d\in A''$ such that $\{x,d\}$ is reserved and $tp^{\mathfrak{A}''}(x,d)=t_i$, then
        add a new element $b_i$ to $A''$,
        put $tp^{\mathfrak{A}}(x,b_i)=t_i$, reserve $\{x,b_i\}$,
        find $a_i\in A\setminus V$ such that $tp^{\mathfrak{A}}(a,a_i)=t_i$, and put $I(b_i)=I(a_i)$.
    An element $a_i$ can be found since $C_a^{\mathfrak{A}}=S$ and therefore there is an element $c_i\in A$ such that $tp^{\mathfrak{A}}(a,c_i)=t_i$. Since $c_i\in A\setminus V$ and $\mathfrak{A}$ realizes $\mathcal{S}$ we have $I(c_i)=C_{c_i}^{\mathfrak{A}}\in\mathcal{S}'$.)
(7) For every $c<x$, if $\{c,x\}$ is not reserved, put $tp^{\mathfrak{A}''}(x,c)=t\in\mathcal{A}^{-}$ such that $I(x)$ is connectable to $I(c)$ in $\mathfrak{A}$ by $t$, and reserve $\{x,c\}$.
    (By Lemma 4.1, for every $S,T\in\mathcal{S}'$, $S$ is connectable to $T$ in $\mathfrak{A}$ by some $t\in\mathcal{A}^{-}$.)
(8) Inactivate $x$.
(9) Go to 4.
```

One application of the above algorithm to the constellation $S\in\mathcal{S}'$ and the structure $\mathfrak{A}'$ expands the structure $\mathfrak{A}'$ to a structure $\mathfrak{A}''$ such that $\mathrm{rank}_{\mathfrak{A}''}(S) > \mathrm{rank}_{\mathfrak{A}'}(S)$.

In step 6, when the types $tp^{\mathfrak{A}''}(x,b)$ are defined, where $b\in V$, the constellation realized by b does not change, since $tp^{\mathfrak{A}''}(x,b)\in\mathcal{A}^{\rightarrow}$. Also in step 7, the constellations realized by the elements $c<x$ are not changed since only types in $\mathcal{A}^{-}$ are used.

Every $x\in A''$ is eventually inactivated since new elements b_i are added at the end of the ordering. When an element x is inactivated it is ensured that for every $c<x$, $tp^{\mathfrak{A}''}(c,x)$ is defined, and $C_x^{\mathfrak{A}''}\in\mathcal{S}$. ◻

LEMMA 4.3. *Let $\mathcal{S}$ be a galaxy. There is a constant r, $r = O(m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$, and there exists a structure $\mathfrak{B}$ realizing $\mathcal{S}$ such that* $\operatorname{rank}_{\mathfrak{B}}(S) < r$ *or* $\operatorname{rank}_{\mathfrak{B}}(S) = \infty$ *for every $S \in \mathcal{S}$.*

Proof. Let $\mathfrak{A}$ realize $\mathcal{S}$. It suffices to define $V \subseteq A$ of appropriate cardinality which satisfies the conditions of Lemma 4.2.

The set V will be constructed in stages.

STAGE 1. Let $V_1 = \{a \in A : \operatorname{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) \le 2m + 1\}$.

Note that if $V_1 = \emptyset$, then by Corollary 4.12, there is a structure $\mathfrak{B}$ such that for every $S \in \mathcal{S}$, $\operatorname{rank}_{\mathfrak{B}}(S) = \infty$. In this case only Stages 1 and 2 are performed.

STAGE i $(i > 1)$.

(1) If $\operatorname{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) > card(V_{i-1}) \cdot m$ for every $a \in A \setminus V_{i-1}$,
then put $V_i = V_{i-1}$ and `stop`.

(2) Put $V_i = V_{i-1} \cup \{a \in A \setminus V_{i-1} : \operatorname{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) \le card(V_{i-1}) \cdot m\}$.

(3) Go to Stage $i + 1$.

Note that there is a stage i such that $V_i = V_{i-1}$. Indeed, for every stage i, let $C(V_i) = \{S \in \mathcal{S} :$ there is $a \in V_i$ such that $C_a^{\mathfrak{A}} = S\}$. Hence, for every $i > 1$, if $V_i \ne V_{i-1}$, then $C(V_i) \supset C(V_{i-1})$. Therefore, since $\mathcal{S}$ is finite, the number p of stages performed is less than or equal to $card(\mathcal{S})$. Put $V = V_p$.

Now, we estimate $card(V)$. We have $card(V_1) \le (2m+1)\cdot card(\mathcal{S})$, and, for $i > 1$,

$$card(V_i) \le card(V_{i-1}) + m \cdot card(V_{i-1}) \cdot (1 + m \cdot card(\mathcal{S})).$$

If we put $q = 1 + m \cdot card(\mathcal{S})$, then we have $card(V_1) \le 3q^p$. Moreover, for every $a \in V$, $\operatorname{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) \le card(V)$, and for every $a \in A \setminus V, \operatorname{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) > card(V) \cdot m$. Put $r = 3 \cdot (1 + m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$. This by Lemma 4.2 finishes the proof. □

Now we are ready to introduce the main definition of this section.

DEFINITION 4.14. *Let $\mathcal{S}$ be a set of constellations. A* finite representation *of $\mathcal{S}$ is a system*

$$\langle \mathcal{S}_1, V, C, I, F, G\rangle,$$

where $\mathcal{S}_1$ is a set of constellations , V and C are sets, I, F, G are functions such that

$$I : C \mapsto \mathcal{S}, \quad F : C \times C \to \mathcal{A}, \quad G : (\mathcal{S} \setminus \mathcal{S}_1) \times V \to \mathcal{A},$$

and the following conditions hold:

(f1) *$\mathcal{S}_1 \subseteq \mathcal{S}$, $V \subseteq C$, and $card(C) \le m \cdot (2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$;*

(f2) *$I(V) = \mathcal{S}_1$, $I(C \setminus V) \subseteq \mathcal{S} \setminus \mathcal{S}_1$,*
$F(a, a) = F(a, b)\lceil\{x\}$, and for every $a \ne b$, $F(a, b) = F(b, a)^*$,
$G : (\mathcal{S} \setminus \mathcal{S}_1) \times V \to \mathcal{A}^{\rightarrow} \cup \mathcal{A}^{-}$;

(f3) *For every $b \in C$ define $D(b) = \{F(b, c) : c \in C, c \ne b, F(b, c) \in \mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow}\}$. Then*

  (a) *for every $b \in C$ and every $t \in D(b)$, there is exactly one $c \in C, c \ne b$ such that $F(b, c) = t$,*

  (b) *$D(a) = I(a)$ for every $a \in V$,*

  (c) *for every $b \in C \setminus V$, we have $D(b) \subseteq I(b)$, and for every $t \in I(b) \setminus D(b)$ there is $T \in \mathcal{S} \setminus \mathcal{S}_1$ such that $I(b)$ is connectable to T by t;*

(f4) *for every $S, T \in \mathcal{S} \setminus \mathcal{S}_1$, S is connectable to T by some $t \in \mathcal{A}^{-}$;*

(f5) *for every $S \in \mathcal{S} \setminus \mathcal{S}_1$ we have $\{G(S, a) \in \mathcal{A}^{\rightarrow} : a \in V\} \subseteq S$, and for every $a \in V$, S is connectable to $I(a)$ by $G(S, a)$;*

(f6) *for every $S \in \mathcal{S} \setminus \mathcal{S}_1$ and every $t \in S$, if $t \in \mathcal{A}^{\rightarrow}$ and there is no $T \in \mathcal{S} \setminus \mathcal{S}_1$ such that S is connectable to T by t, then there is a unique $a \in V$ such that $G(S, a) = t$;*

(f7) *for every $S \in \mathcal{S} \setminus \mathcal{S}_1$ and every $t \in S$, if there is no $a \in V$ such that $G(S,a) = t$, then there is $T \in \mathcal{S} \setminus \mathcal{S}_1$ such that S is connectable to T by t.*

We say that $\mathcal{S}$ is finitely representable*, if there is a finite representation of $\mathcal{S}$.*

Let us note that the notion of a finite representation is almost identical with the notion of small representation (Definition 3.12). The role of replicable small constellations is taken here by constellations in $\mathcal{S} \setminus \mathcal{S}_1$.

THEOREM 4.15 (galaxy theorem). *A set of constellations $\mathcal{S}$ is a galaxy if and only if $\mathcal{S}$ is finitely representable.*

Proof. ($\Rightarrow$) Let $\mathcal{S}$ be a galaxy. Use Lemma 4.3 to get a structure $\mathfrak{A}$ and a constant r such that $\mathfrak{A}$ realizes $\mathcal{S}$, and for every $S \in \mathcal{S}$, $\mathrm{rank}_{\mathfrak{A}}(S) < r$ or $\mathrm{rank}_{\mathfrak{A}}(S) = \infty$. Put

$\mathcal{S}_1 = \{S \in \mathcal{S} : \mathrm{rank}_{\mathfrak{A}}(S) < r\}$,

$V = \{a \in A : \mathrm{rank}_{\mathfrak{A}}(C_a^{\mathfrak{A}}) < r\}$,

$C = V \cup \{a \in A : \text{there is } b \in V \text{ such that } tp^{\mathfrak{A}}(b,a) \in \mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow}\}$.

Note that if $S \in \mathcal{S} \setminus \mathcal{S}_1$, then $\mathrm{rank}_{\mathfrak{A}}(S) = \infty$. For every $a \in V$, put $I(a) = C_a^{\mathfrak{A}}$. For every $a, b \in C, a \neq b$, put $F(a,b) = tp^{\mathfrak{A}}(a,b)$. For every $S \in \mathcal{S} \setminus \mathcal{S}_1$ find $a \in A \setminus C$ such that $C_a^{\mathfrak{A}} = S$, and for every $c \in V$ put $G(S,c) = tp^{\mathfrak{A}}(a,c)$.

It is easy to check that conditions (f1)–(f7) of Definition 4.14 hold.

($\Leftarrow$) Let $\langle \mathcal{S}_1, V, C, I, F, G \rangle$ be a finite representation of $\mathcal{S}$.

Case 1. $\mathcal{S}_1 = \emptyset$. By condition (f2), $V = \emptyset$. Consequently conditions (f7) and (f4) are equivalent to conditions (a) and (b) of Proposition 4.13.

Case 2. $\mathcal{S}_1 = \mathcal{S}$. By condition (f2), $V = C$. Therefore, by conditions (f1)–(f3), we can define $\mathfrak{V}$ realizing $\mathcal{S}$ with universe V in which $tp^{\mathfrak{V}}(a,b) = F(a,b)$ for every $a, b \in V$.

Case 3. $\mathcal{S}_1 \neq \emptyset$ and $\mathcal{S}_1 \neq \mathcal{S}$. We shall construct a structure $\mathfrak{A}$ realizing $\mathcal{S}$ with the universe A ($A \supseteq C$). In our infinite construction, inactivation and reservation have the same role as in the previous algorithms. Additionally, function I will be extended to all elements of A in such a way that for every $b \in A$, $I(b) = C_b^{\mathfrak{A}}$. In each step of construction the universe of partially defined model $\mathfrak{A}$ will be finite, and we assume that there is a fixed linear ordering on the universe such that any new element added to the universe is greater than the old ones.

```
STAGE 1.
  (1) Let $A = C$.
  (2) For every $a, b \in C$, put $tp^{\mathfrak{A}}(a,b) = F(a,b)$ (cf. (f2)).
  (3) For every $a \in V$, inactivate $a$.
  (4) For every $a, b \in C$, reserve $\{a, b\}$.
  (5) For every $S \in \mathcal{S} \setminus \mathcal{S}_1$, if there is no $b \in C$ such that $I(b) = S$, then add
      a new element $d$ to $A$ and put $I(d) = S$.
STAGE 2.
  (6) Let $b$ be the first active element of $A$ (note that $I(b) \in \mathcal{S} \setminus \mathcal{S}_1$).
  (7) If $b \in C$, then
      (a) for every $t \in I(b) \setminus D(b)$ do
           (i) find $T \in \mathcal{S} \setminus \mathcal{S}_1$ such that $I(b)$ is connectable to $T$ by $t$ (use
               (f3-c)),
          (ii) add a new element $d$ to $A$ and put $I(d) = T$,
          (ii) put $tp^{\mathfrak{A}}(b,d) = t$ and reserve $\{b, d\}$,
      (b) inactivate $b$ and go to 6.
  (8) For every $a \in V$, put $tp^{\mathfrak{A}}(b,a) = G(I(b),a)$ (use (f5) and (f6)).
  (9) For every $t \in I(b)$, if there is no $c \in A$ such that $\{c, b\}$ is reserved, and
```

$tp^{\mathfrak{A}}(c,b) = t^*$, then
(a) find $T \in \mathcal{S} \setminus \mathcal{S}_1$ which is connectable to $I(b)$ by t^* (use (f7)),
(b) add a new element d to A, and put $I(d) = T$,
(c) put $tp^{\mathfrak{A}}(b,d) = t$, and reserve $\{b,d\}$.

(10) For every $a < b$, if $\{a,b\}$ is not reserved, then using (f4) find $t \in \mathcal{A}^-$ such that $I(b)$ is connectable to $I(a)$ by t, put $tp^{\mathfrak{A}}(b,a) = t$ and reserve $\{b,a\}$.

(11) Inactivate b and go to 6.

After performing Stage 1, every $a \in V$ realizes the constellation $I(a) \in \mathcal{S}$ in the partially defined structure $\mathfrak{A}$, by f(2) every constellation $S \in \mathcal{S}_1$ is realized by some $a \in V$, and only elements of V have been inactivated. Moreover, for every $S \in \mathcal{S}$, there is $b \in A$ such that $I(b) = S$.

The role of Stage 2 is to realize all constellations of $\mathcal{S} \setminus \mathcal{S}_1$ by elements of A. Step 7 is executed if, at Stage 1, some types between the chosen b and the other elements of A were defined using function F. For every type of $I(b)$ that has not been defined, a new element d is added to A, and some $T \in \mathcal{S} \setminus \mathcal{S}_1$ is put as $I(d)$. Therefore, only constellations of $\mathcal{S} \setminus \mathcal{S}_1$ have to be realized in the next steps. In step 10, when the types between an element b and smaller, already inactivated elements are defined, the constellation realized by the earlier elements are not changed because only types in $\mathcal{A}^-$ are used.

Every $b \in A$ is eventually inactivated since new elements are added at the end of the ordering. When an element b is inactivated it is ensured that $C_b^{\mathfrak{A}} \in \mathcal{S}$. ∎

COROLLARY 4.16. *If $\langle \mathcal{S}_1, V, C, I, F, G \rangle$ is a finite representation of $\mathcal{S}$, then there exists a structure $\mathfrak{A}$ realizing $\mathcal{S}$, such that $A = C \cup B$ for every $a \in B$, $\mathrm{rank}_{\mathfrak{A}}(C_b^{\mathfrak{A}}) = \infty$, and conditions* (f1)–(f7) *hold.*

Proof (sketch). Take $\mathfrak{A}$ given by part ($\Leftarrow$) of the proof of Theorem 4.15. ∎

4.4. Complexity. In this subsection, using the galaxy theorem proved in the previous subsection, we provide an algorithm solving the satisfiability problem for C_1^2. This algorithm works in nondeterministic, double exponential time. In the next section, using more sophisticated techniques, we will show that this bound can be improved.

COROLLARY 4.17. $\mathrm{SAT}(\mathcal{C}_1^2) \in$ 2-NEXPTIME.

Proof. We describe a nondeterministic algorithm which for every sentence $\Phi \in \mathcal{C}_1^2$ decides if Φ is satisfiable and works in time doubly exponential with respect to the length of Φ.

Let Φ be a $\mathcal{C}_1^2$-sentence of length n. In the first step we use the polynomial time algorithm from Corollary 4.8 to obtain a sentence Ψ in $\exists^{=1}$-constellation form

$$\Psi = \forall x \forall y \phi(x,y) \wedge \bigwedge_{1 \le i \le m} \forall x \exists^{=1} y R_i(x,y),$$

which is satisfiable if and only if Φ is satisfiable. Moreover Ψ has at most $p = O(n)$ predicate letters and has length $O(n \log n)$.

Then we use Theorem 4.6. We build the set $\mathcal{A}$ in time $O(2^{4p})$, and we guess a set $\mathcal{S}$ of $\mathcal{A}$-$\mathcal{R}$-constellations. Note that $card(\mathcal{A}) \le 2^{4p}$, and $card(\mathcal{S}) \le (2^{4p})^m$, where $m = card(\mathcal{R})$ is the number of existential quantifiers in Ψ. Therefore $\mathcal{S}$ can be guessed in time $(2^{4p})^m \cdot m \cdot 4p = O(2^{O(n^2)})$.

Next, we apply Theorem 4.15 and guess sets $\mathcal{S}_1$, V, and C, as well as functions I, F, and G as in Definition 4.14. Since $\mathcal{S}_1 \subseteq \mathcal{S}$, and $card(V) \le card(C) \le m \cdot (2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$ we can guess the components in time:

$$\begin{aligned}
\mathcal{S}_1 &\quad — \quad O(card(\mathcal{S})) = O\big(2^{O(n^2)}\big),\\
V \text{ and } C &\quad — \quad O(m(2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}) = O\big(2^{2^{O(n^2)}}\big),\\
I, F &\quad — \quad O((card(C)^2) = O\big(2^{2^{O(n^2)}}\big),\\
G &\quad — \quad O(card(\mathcal{S} \setminus \mathcal{S}_1) \cdot card(V)) = O\big(2^{2^{O(n^2)}}\big).
\end{aligned}$$

Therefore, the time required for this step is bounded by $O(2^{2^{cn^2}})$ for some constant c.

Finally, we check whether the system $\langle \mathcal{S}, \mathcal{S}_1, V, C, I, F, G\rangle$ is a finite representation of $\mathcal{S}$. It is easy to see that this can also be done in time $O(2^{2^{cn^2}})$. □

4.5. An example. It is well known that the class $\mathcal{C}_1^2$ admits axioms of infinity, i.e., there are satisfiable sentences of $\mathcal{C}_1^2$ that have only infinite models. Using the notion of a finite representation, in the proof of Corollary 4.17 we have described an algorithm solving the satisfiability problem for sentences in constellation form, which did not depend on constructing a complete model.

The size of the finite representation $\langle \mathcal{S}_1, V, C, I, F, G\rangle$ of a set of constellations $\mathcal{S}$ depends mainly on the cardinality of the set C. It is bounded in Definition 4.14 by a number that is exponential with respect to the number of constellations in $\mathcal{S}$ and double exponential with respect to the number of predicate letters in the signature.

Below we give an example Φ of a sentence in $\mathcal{C}_1^2$ which has finite models, and is such that for every finite representation $\langle \mathcal{S}_1, V, C, I, F, G\rangle$ of the set of constellations realized in a model of Φ, the cardinality of C is doubly exponential. In this example, following the idea of Lewis's proof [27] that NEXPTIME is reducible to the monadic Gödel class, we use a concise representation of the successor relation between encodings of natural numbers that is reminiscent to that used by Jones and Selman in [23].

Let n be a positive integer.

Let $\mathcal{L} = \{B_1, \ldots, B_n, C_0, C_1, \ldots, C_n,$ *Root*, *Leaf*, *Left*, *Right*, *In*, $R\}$, where B_i, C_i, *Root*, *Leaf* are monadic predicate letters and *Left*, *Right*, *In*, R are binary predicate letters. The sentence Φ will describe the unique model (up to isomorphism) that is a full binary tree of height $2^n - 1$.

The sentence Φ is a conjunction of the following sentences.

$\forall x \exists^{=1} y\ R(x, y)$,
$\forall x \exists^{=1} y\ \mathit{Left}(x, y)$,
$\forall x \exists^{=1} y\ \mathit{Right}(x, y)$,
$\forall x \exists^{=1} y\ \mathit{In}(x, y)$,
$\forall x \forall y\ \mathit{Root}(x) \wedge \mathit{Root}(y) \to x = y$,
$\forall x \forall y\ R(x, y) \to \mathit{Root}(y)$,
$\forall x \forall y\ \neg[\mathit{Left}(x, y) \wedge \mathit{Right}(x, y)] \vee \mathit{Leaf}(x)$,
$\forall x\ \mathit{Leaf}(x) \to [\mathit{Left}(x, x) \wedge \mathit{Right}(x, x)]$,
$\forall x \forall y\ \mathit{In}(x, y) \to [\neg \mathit{Leaf}(y) \wedge (\mathit{Left}(y, x) \vee \mathit{Right}(y, x))$
$\qquad \vee \mathit{Root}(x) \wedge \mathit{Root}(y)]$,
$\forall x\ \mathit{Root}(x) \leftrightarrow \bigwedge_{0 \le i < n} \neg B_i(x)$,
$\forall x\ \mathit{Leaf}(x) \leftrightarrow \bigwedge_{0 \le i < n} B_i(x)$,
$\forall x\ C_0(x) \wedge [\bigwedge_{0 \le i < n} (C_i(x) \leftrightarrow (C_{i-1}(x) \wedge B_i(x)))]$,
$\forall x \forall y\ [\neg \mathit{Leaf}(x) \wedge (\mathit{Left}(x, y) \vee \mathit{Right}(x, y))] \to$
$\qquad \bigwedge_{0 \le i < n} [B_i(y) \leftrightarrow \neg(B_i(x) \leftrightarrow C_{i-1}(x))]$.

The sentence Φ is a slight modification of the example given in [29]. A similar example can also be found in [16]. It is worth noticing that Φ is in $\exists^{=1}$-constellation form. We have here $\mathcal{R} = \{\mathit{In}, \mathit{Left}, \mathit{Right}, R\}$.

PROPOSITION 4.18. *The sentence Φ is satisfiable, and if $\mathfrak{A}$ realizes Φ, then $card(\mathfrak{A}) = 2^{2^n} - 1$.*

Proof. Let, for every d, $0 \leq d \leq 2^n - 1$, $Level_d$ denote the unique 1-type over the set $\{B_1, \ldots, B_n\}$ such that $B_i(x) \in Level_d$ if and only if the ith bit of d in the binary notation is 1.

It is easy to see that a full binary tree $\mathfrak{T}$ (see Figure 1) is the unique model of Φ with the interpretations for the predicate letters such that for every $a, b \in \mathfrak{T}$

$Root(a)$	iff	a is the root of $\mathfrak{T}$,
$Leaf(a)$	iff	a is a leaf of $\mathfrak{T}$,
$Left(a, b)$	iff	b is the immediate left successor of a or $a = b$ is a leaf of $\mathfrak{T}$,
$Right(a, b)$	iff	b is the immediate right successor of a or $a = b$ is a leaf of $\mathfrak{T}$,
$In(a, b)$	iff	b is the immediate predecessor of a or $a = b$ is the root of $\mathfrak{T}$,
$Level_d(a)$	iff	the distance from the root to a is equal to d.

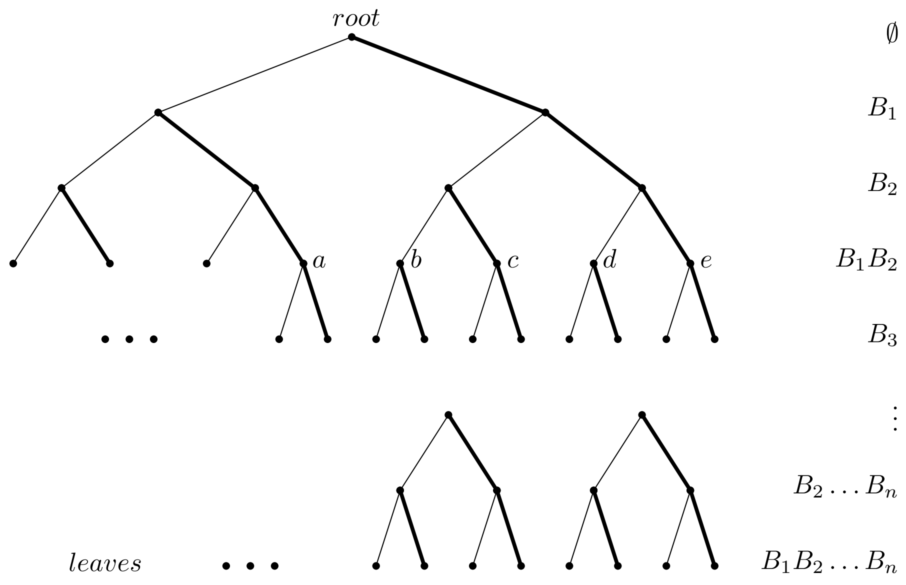


FIG. 1. *The model for Φ.*

Since the remaining predicates are explicitly defined by Φ, their interpretations can be derived from the interpretations above.

Note that for every d, $0 \leq d \leq 2^n - 1$, there is $a \in T$ such that $Level_d(a)$, and for every $a \in T$, $Leaf(a)$ if and only if $Level_{2^n-1}(a)$. $\square$

5. The main result.

5. The main result. The main result of this paper is the following theorem.

THEOREM 5.1. SAT($\mathcal{C}_1^2$) $\in$ NEXPTIME.

We begin this section by providing some intuition arising from a close analysis of the example given in section 4.5. Then we define a notion of a *concise representation* of a set of constellations which will play a similar role to the notion of finite representation but will require less space. Finally, we show how to use this notion to get a nondeterministic decision procedure working in exponential time, and solving the satisfiability problem for C_1^2. In the last step we use graph-theoretical notions and results given in the appendix.

5.1. Example continued. In this subsection we want to provide some intuition on how to improve the double exponential upper complexity bound for C^2_1. This will be done by discussing in greater detail the example from section 4.5 of the sentence Φ that describes a binary tree of exponential height.

Let us first examine the types and constellations realized in the model of Φ (see Figure 1). For every pair of elements x and y such that y is an immediate successor of x, x and y are joined with a thin or a thick line depending on whether $\mathit{Left}(x,y)$ or $\mathit{Right}(x,y)$ holds and then also, $\mathit{In}(y,x)$ holds. Moreover, every element x of the tree should in the picture be joined to the root by a line representing $R(x,\text{root})$.

For every element x, the unique 1-type realized by x contains positive formulas of the form $B_i(x)$ for those and only those B_i-s which are listed on the right margin. The values of C_i-s on x are determined by the values of B_i-s. If an element x is neither the root nor a leaf then the 1-type realized by x does not contain any positive appearance of other predicate letters.

Below, when describing types, we list only atomic formulas with both x and y, omitting the remaining two-variable conjuncts which are negations of atomic formulas that are not listed. The elements on the same level $d > 0$ of the tree realize constellations of two kinds with the same center. For $d = 2, 3, \dots, 2^n - 3$ we have the following constellations, each containing exactly four 2-types:

$S = \{\mathit{Left}(y,x) \wedge \mathit{In}(x,y), \mathit{Left}(x,y) \wedge \mathit{In}(y,x), \mathit{Right}(x,y) \wedge \mathit{In}(y,x), R(x,y)\}$,

$T = \{\mathit{Right}(y,x) \wedge \mathit{In}(x,y), \mathit{Left}(x,y) \wedge \mathit{In}(y,x), \mathit{Right}(x,y) \wedge \mathit{In}(y,x), R(x,y)\}$.

Therefore, elements denoted by a, c, and e realize the constellation T, and the constellation S is realized by b and d.

The constellations realized on the first level include exactly three 2-types:

$S = \{\mathit{Left}(y,x) \wedge \mathit{In}(x,y) \wedge R(x,y), \mathit{Left}(x,y) \wedge \mathit{In}(y,x), \mathit{Right}(x,y) \wedge \mathit{In}(y,x)\}$,

$T = \{\mathit{Right}(y,x) \wedge \mathit{In}(x,y) \wedge R(x,y), \mathit{Left}(x,y) \wedge \mathit{In}(y,x), \mathit{Right}(x,y) \wedge \mathit{In}(y,x)\}$.

The root and the leaves realize constellations containing exactly two 2-types. The root realizes the constellation

$S = \{\mathit{Left}(x,y) \wedge \mathit{In}(y,x) \wedge R(y,x), \mathit{Right}(x,y) \wedge \mathit{In}(y,x) \wedge R(y,x)\}$

and $\{\mathit{In}(x,x), R(x,x)\} \subset \mathit{center}(S)$.

If x is a leaf, then

$S = \{\mathit{Left}(y,x) \wedge \mathit{In}(x,y), R(x,y)\}$,

$T = \{\mathit{Right}(y,x) \wedge \mathit{In}(x,y), R(x,y)\}$,

and $\{\mathit{Left}(x,x), \mathit{Right}(x,x)\} \subset \mathit{center}(S) = \mathit{center}(T)$.

Note that since elements on different levels of the tree realize distinct constellations, the number of constellations realized in $\mathfrak{T}$ is exponential with respect to the number of predicate letters in $\mathcal{L}$. Moreover, the number of vassals, $\mathit{card}(V)$, is exponential with respect to the number of constellations realized in $\mathfrak{T}$. Therefore, the number of elements that are indistinguishable from the point of view of constellation they realize can be double exponential.

One could imagine that in order to check whether a sentence in $\exists^{=1}$-constellation form is satisfiable it is not necessary to have the complete submodel with the universe C defined by the finite representation, but it should be sufficient to know which constellations are realized in the submodel and in which number. It is, however, hard to adapt this idea directly.

Note that some elements that realize the same constellation in a given model can be distinguished by taking into account constellations that are realized by partners of the constellations—elements connected to them with a counting type. For example, elements a and c of the tree shown in Figure 1 realize the same constellation but they can be distinguished, since the predecessor of a realizes a constellation of kind T, whereas the predecessor of c realizes a constellation of kind S. Elements a and e, however, remain indistinguishable even if we take into account the additional information.

The above remarks suggest that in order to push the complexity down, a potential model can be described by a set of *indexed* constellations and numbers of elements that realize these constellations. Roughly speaking, an indexed constellation in addition to the information on 2-types realized by an element, carries requests for partner constellations that should realize, together with the host, the 2-types of the host constellation.

5.2. Concise representation. At the end of section 4.3, using the galaxy theorem, which allows us to transform the problem whether a set of constellations is a galaxy into the problem whether the same set is finitely representable, we gave an algorithm solving the satisfiability problem for C^2_1. As it was shown in the previous subsection, the size of a finite representation can be exponential with respect to the number of constellations.

In this section we shall define the notion of a concise representation of a set of constellations which will play a similar role to finite representation but will use only polynomial space with respect to the number of constellations.

We need several additional notions, the most important of which are the notion of an *indexed constellation* (Definition 5.5) and of a *$\mathcal{X}$-rnk-model* (Definition 5.9). An indexed constellation is a pair $\langle S, f\rangle$, where S is a constellation and f is a function that associates a constellation T to each 2-type of S. This definition allows to control not only which constellation S is realized by an element a but also which constellations are realized in the neighborhood of a, that is, by elements that together with a realize 2-types of S.

Definition 5.9 describes a model very precisely. It says which indexed constellations are realized and in which amount. Additionally, it allows to partition a model into parts, each part containing elements realizing the same indexed constellation, and it specifies 2-types that can be realized by elements from these parts.

The first easy fact proved here (Proposition 5.10) gives several necessary conditions for a set of constellations to be a galaxy. These conditions are described in terms of new notions introduced below. We hope that through studying this easy proposition the reader will get familiar with the complex terminology and notation use here. It should also provide a good background for the most important notion of concise representation.

Lemma 5.1 is an analogue of the galaxy theorem, and one could think that it could form a basis to formulate another algorithm for the satisfiability problem for C^2_1, as in section 4.3. However, although the space needed to write a concise representation is small, and most of the conditions of the definition of concise representation are easily[2] computable, condition (c5), however, seems to require still double exponential time since it requires checking whether there exists a model of double exponential cardinality, which in addition satisfies some conditions. As a first step towards removing this difficulty we prove the decomposition theorem (Lemma 5.2) that shows that such a model is composed of a certain number of parts which can be treated separately and independently. Unfortunately, to check if the parts can be constructed we need several technical lemmas.

[2]Here, "easily" means in exponential time.

Let $\mathcal{R} \subseteq \mathcal{L}$ be a fixed set of predicate letters with $card(\mathcal{R}) = m$. Let $\mathcal{A}$ be a fixed set of 2-types closed under the operation *, and let $\mathcal{S}$ be a set of constellations $\mathcal{S} = \{S_1, \ldots, S_w\}$.

Definition 5.2. *Let $S \in \mathcal{S}$. A set S' of types is a* subconstellation of S *if $S' \subseteq S$ and $center(S) \in S'$.*

Note that a subconstellation S' of S is a constellation (cf. Definition 4.3) only if $S' = S$.

Definition 5.3. *Let $\mathfrak{A}$ be an $\mathcal{L}$-structure, and let $S' = \{s_0, s_1, \ldots, s_l\}$ be a subconstellation of S for some $S \in \mathcal{S}$. An element $a \in A$* realizes S' *if $tp^{\mathfrak{A}}(a, a) = s_0$, and there exists a unique sequence $b_1, \ldots, b_l \in A$ such that $tp^{\mathfrak{A}}(a, b_i) = s_i$, $0 < i \leq l$, and $tp^{\mathfrak{A}}(a, b) \in \mathcal{A}^{\leftarrow} \cup \mathcal{A}^{-}$, for each $b \in A$, $b \neq a$, $b \neq b_i$, $0 < i \leq l$. A subconstellation S' is* realized *in $\mathfrak{A}$ if there exists $a \in A$ which realizes S'.*

Definition 5.4. *Let $\mathcal{S}'$ be a set of subconstellations. A structure $\mathfrak{A}$ realizes $\mathcal{S}'$, if every subconstellation $S' \in \mathcal{S}'$ is realized in $\mathfrak{A}$, and every element $a \in A$ realizes a subconstellation of $\mathcal{S}'$.*

Definition 5.5. *Let $S \in \mathcal{S}$. An* indexed constellation *S^f is a pair $\langle S, f \rangle$, where $S \in \mathcal{S}$, and $f : S \setminus \{center(S)\} \to \mathcal{S}$ is a function such that, for every $s \in S$, S is connectable to $f(s)$ by s.*

We denote by $\mathcal{S}_{ind}$ the set of all indexed constellations of $\mathcal{S}$.

Definition 5.6. *Let $\mathfrak{A}$ be an $\mathcal{L}$-structure, and let $a \in A$ realize a constellation of $\mathcal{S}$. Assume that for every $b \in A$, $b \neq a$, if $tp^{\mathfrak{A}}(a, b) \in C_a^{\mathfrak{A}}$, then b realizes a constellation of $\mathcal{S}$.*

We denote by $ind_a^{\mathfrak{A}}$ the function $ind_a^{\mathfrak{A}} : C_a^{\mathfrak{A}} \setminus center(C_a^{\mathfrak{A}}) \to \mathcal{S}$ such that for every $b \in A$, $b \neq a$, if $tp^{\mathfrak{A}}(a, b) \in C_a^{\mathfrak{A}}$, then $ind(tp^{\mathfrak{A}}(a, b)) = C_b^{\mathfrak{A}}$.

An element a of $\mathfrak{A}$ realizes an indexed constellation *S^f if $C_a^{\mathfrak{A}} = S$, and $ind_a^{\mathfrak{A}} = f$.*

Some explanation of Definitions 5.5 and 5.6 was already given in the previous section. Consider again the example of section 4.5. The elements a, c, and e (see Figure 1) realize the same constellation; however, a and c realize different indexed constellations, whereas a and e realize the same indexed constellation.

Definition 5.7. *Let $\mathcal{T}, \mathcal{U} \subseteq \mathcal{S}$. A set $\mathcal{X}$, $\mathcal{X} \subseteq \mathcal{S}_{ind}$, is an* indexing of $\mathcal{T}$ restricted to $\mathcal{U}$ *if for every $S \in \mathcal{T}$ there is a function f such that $S^f \in \mathcal{X}$, and for every $S^f \in \mathcal{X}$ we have $S \in \mathcal{T}$, and $f : S \setminus \{center(S)\} \to \mathcal{U}$.*

We say that a set $\mathcal{X}$, $\mathcal{X} \subseteq \mathcal{S}_{ind}$ is an indexing of $\mathcal{T}$ *if $\mathcal{X}$ is an indexing of $\mathcal{T}$ restricted to $\mathcal{S}$.*

A pair $\langle \mathcal{X}, rnk \rangle$ is a rnk-indexing *of $\mathcal{T}$ if $\mathcal{X}$ is an indexing of $\mathcal{T}$ and rnk is a function such that $rnk : \mathcal{X} \to \mathbb{N}^+$.*

Note that if $\langle \mathcal{X}, rnk \rangle$ is a rnk-indexing of $\mathcal{T}$, $\mathcal{U} \subseteq \mathcal{T}$, $\mathcal{X}' = \{U^f : U^f \in \mathcal{X}$ and $U \in \mathcal{U}\}$ and $rnk' = rnk \restriction \mathcal{X}'$, then $\langle \mathcal{X}', rnk' \rangle$ is a rnk'-indexing of $\mathcal{U}$.

Let $\mathcal{X} \subseteq \mathcal{S}_{ind}$, $\mathcal{U} \subseteq \mathcal{S}$, and $T^f \in \mathcal{X}$. Denote by $T^f \restriction \mathcal{U}$ the subconstellation $= \{s \in T : f(s) \in \mathcal{U}\} \cup \{center(T)\}$ of T.

Definition 5.8. *Let $\langle \mathcal{X}, rnk \rangle$ be a rnk-indexing of $\mathcal{T}$, and let A be a finite set. A function $lab : A \overset{onto}{\longrightarrow} \mathcal{X}$ is called a* rnk-labeling of A *if for every $T^f \in \mathcal{X}$, $card(\{a \in A : lab(a) = T^f\}) = rnk(T^f)$.*

Let $\langle \mathcal{X}, rnk \rangle$ be a rnk-indexing of $\mathcal{T}$, and let A be a finite set. If lab is a rnk-labeling of A, then, for every $T_i \in \mathcal{T}$, we put $A_i^{lab} = \{a \in A : lab(a) = T_i^f$ for some f such that $T_i^f \in \mathcal{X}\}$.

Definition 5.9. *Let $\mathcal{X} \subseteq \mathcal{S}_{ind}$ and let $\langle \mathcal{X}, rnk \rangle$ be a rnk-indexing of $\mathcal{T}$. An $\mathcal{L}$-structure $\mathfrak{A}$ is a* $\mathcal{X}$-rnk-model for $\mathcal{T}$ *($\mathfrak{A} \models_{\mathcal{X}}^{rnk} \mathcal{T}$) if and only if there exists a rnk-labeling lab of A such that for every A_i^{lab}, A_j^{lab}, and every $a \in A_i^{lab}$, a realizes the* *subconstellation* $lab(a)\restriction\{T_j\}$ *in the substructure of* $\mathfrak{A}$ *restricted to* $\{a\} \cup A_j^{lab}$.

The notions of an indexed constellation and of a $\mathcal{X}$-rnk-model are fundamental in our proof of the single exponential upper bound on the complexity of SAT($\mathcal{C}_1^2$). The intuitions behind the above definitions are explained by the next proposition.

Proposition 5.10. *If* $\mathcal{S}$ *is a galaxy then there exist a structure* $\mathfrak{A}$*, a set* $C \subseteq A$*, subsets* $\mathcal{S}_1, \mathcal{S}_2$ *of* $\mathcal{S}$*,* $\mathcal{X} \subseteq \mathcal{S}_{ind}$*, and a function* $rnk : \mathcal{X} \to \{1, \ldots, card(C)\}$*, such that* $\mathfrak{A}$ *realizes* $\mathcal{S}$*,* $card(C) \leq m(2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$*, and the following conditions hold:*

(1) $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$*,* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$*, where* $\mathcal{X}_1$ *is an indexing of* $\mathcal{S}_1$ *restricted to* $\mathcal{S}_1 \cup \mathcal{S}_2$ *and* $\mathcal{X}_2$ *is an indexing of* $\mathcal{S}_2$ *restricted to* $\mathcal{S}$*,* $\langle \mathcal{X}, rnk\rangle$ *is an rnk-indexing of* $\mathcal{S}_1 \cup \mathcal{S}_2$*,*

(2) *for every* $a \in C$*,* $C_a^{\mathfrak{A}} \in \mathcal{S}_1 \cup \mathcal{S}_2$*,*

(3) *for every* $a \in A$*, if* $C_a^{\mathfrak{A}} \in \mathcal{S}_1$*, then* $a \in C$ *and, for every* $a \in A$ *and every* $S \in \mathcal{S}_1$*, if* $C_a^{\mathfrak{A}} = S$*, then* $C_a^{\mathfrak{C}} = S$*,*

(4) $\mathfrak{C} \models_{\mathcal{X}}^{rnk} \mathcal{S}_1 \cup \mathcal{S}_2$*,*

(5) *for every* $S \in \mathcal{S} \setminus \mathcal{S}_1$*,* $\mathrm{rank}_{\mathfrak{A}}(S) = \infty$.

Proof. Let $\mathcal{S}$ be a galaxy, and let $\langle \mathcal{S}_1, V, C, I, F, G\rangle$ be a finite representation of $\mathcal{S}$ which exists by Theorem 4.15. Let $\mathfrak{A}$ be a structure realizing $\mathcal{S}$ whose existence follows from Corollary 4.16. The domain of the structure $\mathfrak{A}$ is divided into two parts: B and C with $B = A \setminus C$, and $V \subset C$.

Let $\mathcal{S}_1 = \{S \in \mathcal{S} : S = C_a^{\mathfrak{A}}$, for some $a \in V\}$,

$\mathcal{S}_2 = \{S \in \mathcal{S} : S = C_a^{\mathfrak{A}}$, for some $a \in C \setminus V\}$,

$\mathcal{X}_1 = \{C_a^{\mathfrak{A},f} : a \in V, f = ind_a^{\mathfrak{A}}\}$,

$\mathcal{X}_2 = \{C_a^{\mathfrak{A},f} : a \in C \setminus V, f = ind_a^{\mathfrak{A}}\}$, $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$,

and for every $S^f \in \mathcal{X}$, let $rnk(S^f) = card(\{a \in C : S = C_a^{\mathfrak{A}}$ and $f = ind_a^{\mathfrak{A}}\}$.

Assume that $\mathcal{S}_1 = \{S_1, \ldots, S_x\}$, $\mathcal{S}_2 = \{S_{x+1}, \ldots, S_y\}$, and $\mathcal{X} = \{\langle S_1, f_{11}\rangle, \ldots, \langle S_1, f_{1,v_1}\rangle, \langle S_2, f_{21}\rangle, \ldots, \langle S_2, f_{2,v_2}\rangle, \ldots, \langle S_y, f_{y1}\rangle, \ldots, \langle S_y, f_{y,v_y}\rangle\}$, where $v_i = card(\{f_{ij} : S_i^{f_{ij}} \in \mathcal{X}\})$, $1 \leq i \leq y$.

We have partitioned the set C into sets $C_1, \ldots, C_y$ in such a way that for every $a \in C$, if $a \in C_i$, then $C_a^{\mathfrak{A}} = S_i$ (see Figure 2). Furthermore, every set C_i is partitioned into classes of elements realizing the same indexed constellations $\langle S_i, f_{ij}\rangle$. Moreover, for every $a \in C_i$, if $C_i \subseteq V$, then $C_a^{\mathfrak{C}} = S_i$. This means that for every $a \in V$, for every $b \in B$, $tp^{\mathfrak{A}}(a, b) \in \mathcal{A}^{\leftarrow} \cup \mathcal{A}^{-}$ ($\notin \mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow}$) which is denoted in Figure 2 by the slashed arrows.

Define a rnk-labeling lab of C letting $lab(c) = S^f$, where $S = C_c^{\mathfrak{A}}$ and $f = ind_c^{\mathfrak{A}}$.

It is easy to check that conditions (1)–(5) hold. ◻

The main idea of the proof of Theorem 5.1 is to replace in Proposition 5.10 the condition *there exists a structure* $\mathfrak{A}$ with something easier to verify and to substitute the implication by an equivalence. To do this we add several additional conditions, which are easily computable.

Definition 5.11. *A* concise representation of $\mathcal{S}$ *is a system*

$$\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{X}, rnk, \mathcal{Y}\rangle,$$

where $\mathcal{S}_1, \mathcal{S}_2$ *are sets of constellations,* $\langle \mathcal{X}, rnk\rangle$ *is a rnk-indexing of* $\mathcal{S}_1 \cup \mathcal{S}_2$*,* $\mathcal{Y}$ *is an indexing of* $\mathcal{S} \setminus \mathcal{S}_1$*, and the following conditions hold:*

(c1) $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{S}$*,* $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$*,*

(c2) $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$*, where* $\mathcal{X}_1$ *is an indexing of* $\mathcal{S}_1$ *restricted to* $\mathcal{S}_1 \cup \mathcal{S}_2$ *and* $\mathcal{X}_2$ *is an indexing of* $\mathcal{S}_2$ *restricted to* $\mathcal{S}$*,*

(c3) *for every* $\langle S, f_1\rangle, \langle S, f_2\rangle \in \mathcal{Y}$*,* $f_1 = f_2$*,*

(c4) $\sum_{S^f \in \mathcal{X}} rnk(S^f) \leq m(2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}$*,*

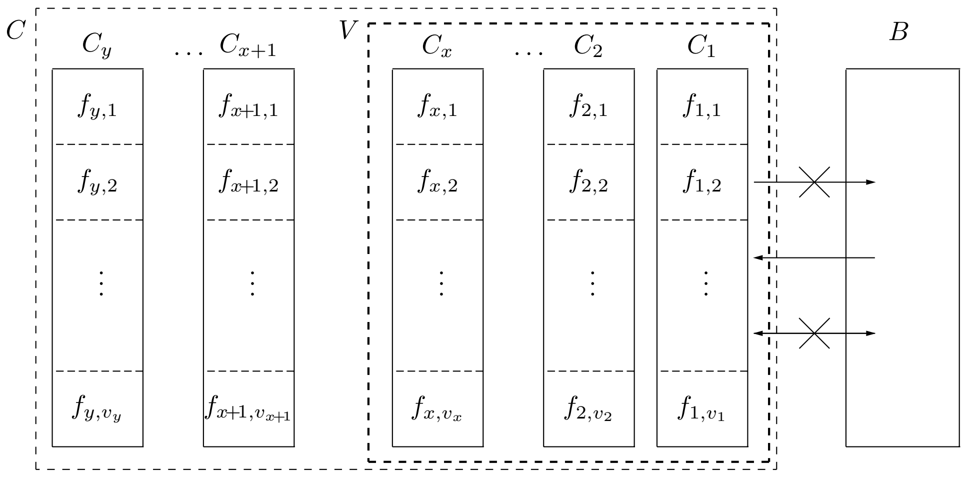


FIG. 2. *Partition of the universe according to the rnk-labeling.*

(c5) *there is a structure $\mathfrak{C}$ such that $\mathfrak{C} \models^{rnk}_{\mathcal{X}} \mathcal{S}_1 \cup \mathcal{S}_2$,*

(c6) *for every $S^f \in \mathcal{Y}$ and every $s \in S$, if $f(s) \in \mathcal{S}_1$, then $s \in \mathcal{A}^{\rightarrow}$,*

(c7) *for every $S^f \in \mathcal{Y}$ and every $T \in \mathcal{S}_1$, $card(\{s \in S\colon f(s) = T\}) \leq \sum_{T^f \in \mathcal{X}_1} rnk(T^f)$ and if $card(\{s \in S\colon f(s) = T\}) < \sum_{T^f \in \mathcal{X}_1} rnk(T^f)$, then there exist $t \in \mathcal{A}^-$ such that S is connectable to T by t,*

(c8) *for every $S, T \in \mathcal{S} \setminus \mathcal{S}_1$, S is connectable to T by some $t \in \mathcal{A}^-$.*

Some comments are in order here. Conditions (c1)–(c8) of the above definition precisely describe the situation illustrated in Figure 2.

(c1) Three subsets of $\mathcal{S}$ are distinguished: $\mathcal{S}_1$, constellations realized by elements of V; $\mathcal{S}_2$, constellations realized by elements of $C \setminus V$; and $\mathcal{S} \setminus \mathcal{S}_1$, constellations realized by elements of B.

(c2) $\mathcal{X}_1$ is an indexing of $\mathcal{S}_1$ and it defines partitions of set $C_i \subseteq V$ into appropriate classes. The indexing guarantee that the elements of V realize constellations of $\mathcal{S}_1$ in the substructure restricted to C. Similarly, $\mathcal{X}_2$ defines partitions of $C_i \subseteq C \setminus V$.

(c3) The indexing $\mathcal{Y}$ defines 2-types which are realized by pairs of elements $\langle a, b \rangle$, where $a \in B$, and $b \in V$. Note that if B is nonempty, then the constellations realized in B or in $C \setminus V$ are expected to appear infinitely many times.

(c4) The function *rnk* defines the cardinality of every class of C_i and thus the cardinality of V and C.

(c5) This condition takes care of definability of $\mathfrak{C}$. In particular, it specifies which elements in C_i are connected with elements in C_j by counting types and which counting types are used to realize the connections.

(c6) An element $a \in B$ can be connected to an element $b \in V$ only by a type from $\mathcal{A}^{\rightarrow} \cup \mathcal{A}^-$. These types do not change the constellation realized by elements in V.

(c7) If an element $a \in B$ realizes S such that $S^f \in \mathcal{Y}$, then the number of elements in V is sufficient to realize types of S. Moreover, it is possible to define noncounting types between a and elements of V, if necessary.

(c8) Every two constellations which are realized infinitely many times are connectable by a noncounting type in $\mathcal{A}^-$.

Lemma 5.1. *A set of constellations $\mathcal{S}$ is a galaxy if and only if there exists a concise representation of $\mathcal{S}$.*

Proof. ($\Rightarrow$) Let $\mathcal{S}$ be a galaxy. Take a structure $\mathfrak{A}$, a set $C \subseteq A$, $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{S}$, $\mathcal{X}$, and a function *rnk* given by Proposition 5.10. For every constellation $S \in \mathcal{S} \setminus \mathcal{S}_1$ find an element $b \in A \setminus C$ such that b realizes S (if it exists) and add the indexed constellation realized by b to $\mathcal{Y}$. Note that if $\mathcal{S} = \mathcal{S}_1$, then $\mathcal{Y} = \emptyset$.

It is obvious that $\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{X}, rnk, \mathcal{Y}\rangle$ satisfies conditions (c1)–(c8).

($\Leftarrow$) Let $\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{X}, rnk, \mathcal{Y}\rangle$ be a concise representation of $\mathcal{S}$. By Theorem 4.15 it suffices to show that there exists a finite representation of $\mathcal{S}$. Let $\mathfrak{C}$ be the structure given by (c5), and let *lab* be the *rnk*-labeling of C. Let $V = \{a \in C : lab(a) \in \mathcal{S}_1\}$. We will define functions I, F, G so that the system $\langle \mathcal{S}_1, V, C, I, F, G\rangle$ is a finite representation of $\mathcal{S}$. To define I, for every $a \in C$, put $I(a) = S$, where $lab(a) = \langle S, f\rangle$. In order to define F, for every $a, b \in C$, put $F(a, b) = tp^{\mathfrak{C}}(a, b)$. Now, let $S \in (\mathcal{S} \setminus \mathcal{S}_1)$, $S = \{s_0, s_1, \ldots, s_k\}$. By (c3), there exists $\langle S, f\rangle \in \mathcal{Y}$. By condition (c8), find k distinct elements $a_1, \ldots, a_k \in V$ such that for every i, $1 \leq i \leq k$, $f(s_i) = lab(a_i)$. For every a_i, $1 \leq i \leq k$, define $G(S, a_i) = s_i$. If there is an element $b \in V$ such that $G(S, b)$ has not been defined yet, then, by (c8), find a type $s \in \mathcal{A}^-$ such that S is connectable to $lab(b)$ by s, and put $G(S, b) = s$. It is easy to check that conditions (f1)–(f7) of Definition 4.14 hold. $\square$

5.3. Complexity.

Proof of Theorem 5.1. The proof proceeds in the same way as the proof of Corollary 4.17.

Let Φ be a $\mathcal{C}^2_1$-sentence of length n. By Corollary 4.8 we obtain a sentence Ψ in $\exists^{=1}$-constellation form which is satisfiable if and only if Φ is satisfiable. After defining the set $\mathcal{A}$, we guess a set $\mathcal{S}$ of $\mathcal{A}$-$\mathcal{R}$-constellations, as in Theorem 4.6.

Then, we guess sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of constellations, an *rnk*-indexing $\langle \mathcal{X}, rnk\rangle$ of $\mathcal{S}_1 \cup \mathcal{S}_2$, and an indexing $\mathcal{Y}$ of $\mathcal{S} \setminus \mathcal{S}_1$, as in Definition 5.11.

In contrast to the proof of Corollary 4.17, this step can be performed in time $O(2^{n^3})$ since $card(\mathcal{S}_{ind}) \leq 2^{n^3}$, and the length of a maximal value of the function *rnk* is bounded by $\log(m(2m \cdot card(\mathcal{S}))^{card(\mathcal{S})}) = O(2^{dn^2})$ for some constant d.

Finally, we check in time $O(2^{dn^3})$ whether $\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{X}, rnk, \mathcal{Y}\rangle$ is a concise representation of $\mathcal{S}$. $\square$

Remark. It can be easily seen that all the conditions of Definition 5.11 except (c5) can be verified in time $O(2^{dn^3})$ for some constant d. It is less obvious that the same holds for (c5). We devote the next subsection to the problem how to verify (c5) in exponential time.

As a consequence of Theorem 5.1, by Corollary 4.9, we get the following corollary.

Corollary 5.12. SAT($\mathcal{C}^2$) $\in$ 2-NEXPTIME.

There are at least two reasons why it is difficult to improve the above result. One has been already discussed at the end of section 4.2. Another one is that it is difficult to generalize the notion of a constellation to count an arbitrary number of witnesses without increasing the number of possible constellations to double exponential. In spite of this we conjecture that the satisfiability problem for the full C^2 has only exponential complexity.

5.4. Verification of (c5). The following definition will be used in the decomposition theorem (Lemma 5.2).

DEFINITION 5.13. *Let $\mathcal{T} \subseteq \mathcal{S}$, and let $\langle \mathcal{X}, rnk\rangle$ be an rnk-indexing of $\mathcal{T}$. For $S_i, S_j \in T$ we define $\mathcal{X}_{ij} = \{S^f \in \mathcal{X} : S = S_i \text{ or } S = S_j\}$, and $rnk_{ij} = rnk{\restriction}\mathcal{X}_{ij}$.*

The following lemma gives a condition which is equivalent to condition (c5) of Definition 5.11 but is more tractable.

LEMMA 5.2 (decomposition theorem). *Let $\mathcal{T} \subseteq \mathcal{S}$, and let $\langle \mathcal{X}, rnk\rangle$ be an rnk-indexing of $\mathcal{T}$. Then the following conditions are equivalent:*

(1) *There is a structure $\mathfrak{C}$ such that $\mathfrak{C} \models^{rnk}_{\mathcal{X}} \mathcal{T}$.*

(2) *For every $S_i, S_j \in \mathcal{T}$ there exists a structure $\mathfrak{C}_{ij}$ such that $\mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}$.*

Remark. Condition 1 coincides with condition (c5).

Proof. (1) $\Rightarrow$ (2) is obvious.

(2) $\Rightarrow$ (1). Assume that Condition (2) holds. Let $\mathcal{X} = \{\langle S_1, f_{1,1}\rangle, \dots, \langle S_1, f_{1,v_1}\rangle, \langle S_2, f_{2,1}\rangle, \dots, \langle S_2, f_{2,v_2}\rangle, \dots, \langle S_y, f_{y,1}\rangle, \dots, \langle S_y, f_{y,v_y}\rangle\}$. For every i, j, $1 \le i \le j \le y$, let $\mathfrak{C}_{ij}$ be a structure such that $\mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}$, and let lab_{ij} be the rnk_{ij}-labeling of (the universe of) $\mathfrak{C}_{ij}$.[3] Note that

$$card\Big(\Big\{a \in C_{ij} : lab_{ij}(a) = S_i^f \text{ and } S_i^f \in \mathcal{X}_{ij}\Big\}\Big) = \sum_{S_i^f \in \mathcal{X}_{ij}} rnk_{ij}(S_i^f) = card(C_{ii}).$$

We are going to define a structure $\mathfrak{C}$. Let C, the universe of $\mathfrak{C}$, be a set such that there exists a rnk-labeling of C, and let lab be such a labeling. It is easy to notice that C could also be defined as a union of disjoint copies of C_{ii}. In fact, define $C_i = \{a \in C : lab(c) = S_i^f$, for some function $f\}$ for every i, $1 \le i \le y$. Then $C = \bigcup_{i \le y} C_i$, and $card(C_i) = card(C_{ii})$.

Now we shall define types realized by pairs of elements in $\mathfrak{C}$. For every i, j such that $1 \le i \le j \le y$, choose a function g_{ij} such that $g_{ij} : C_{ij} \xrightarrow[onto]{1-1} C_i \cup C_j$, and for every $a \in C_{ij}$, $lab_{ij}(a) = lab(g_{ij}(a))$. For every a, b such that $a \in C_i$, $b \in C_j$, and $i \le j$, put $tp^{\mathfrak{C}}(a, b) = tp^{\mathfrak{C}_{ij}}(g_{ij}^{-1}(a), g_{ij}^{-1}(b))$.

To show that the structure $\mathfrak{C}$ satisfies condition 1, let $S_i, S_j \in \mathcal{T}$, and $a \in C_i$ (cf. Definition 5.9). Set $B = \{tp^{\mathfrak{C}}(a, b) : b \in C_j\} \cap (\mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow})$. By construction, $B = \{tp^{\mathfrak{C}_{ij}}(g_{ij}^{-1}(a), g_{ij}^{-1}(b)) : b \in C_j\} \cap (\mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow})$ and $B \cup \{tp^{\mathfrak{C}_{ij}}(a, a)\} = lab_{ij}(g_{ij}^{-1}(a)){\restriction}\{S_j\} = lab(a){\restriction}\{S_j\}$. $\square$

Now we present three technical lemmas, each of them showing how to check condition 2 of Lemma 5.2. Lemma 5.3 deals with the case $i = j$, Lemma 5.4 with the case when $i \ne j$ and both S_i and S_j are realized by many elements, and Lemma 5.5 with the case when $i \ne j$, but only one of S_i and S_j is realized by many elements.

We are very sorry that in spite of the suggestions of the referees, many requests of our friends, and our best intentions we have not been able to make the proofs more readable.

We have tried to write a reader-friendly paper, but we have been only partially successful in this attempt. We did not find a way to avoid yet another technical definition below.

We will use the following notation.

[3]The notation $\mathcal{X}_{ii}, C_{ii}$, and $\mathfrak{C}_{ii}$ can be misleading—the notation $\mathcal{X}_i, C_i$, and $\mathfrak{C}_i$ would be more intuitive, but we keep the less intuitive notation since it is more uniform.

Definition 5.14. *Assume that $\mathcal{T} \subseteq \mathcal{S}$, and $\langle \mathcal{X}, rnk\rangle$ is a rnk-indexing of $\mathcal{T}$. Let q be a positive integer. For $S_i, S_j \in \mathcal{T}$ and every $s \in S_i$, define*

$$\begin{aligned}
c_i &= \sum_{S_i^f \in \mathcal{X}} rnk\big(S_i^f\big),\\
u_{ij}(s) &= \sum_{S_i^f \in \mathcal{X}, f(s)=S_j} rnk\big(S_i^f\big) \qquad \text{for } s \in S_i,\\
S_{ij}^q &= \{s \in S_i \cap \mathcal{A}^{\leftrightarrow} : u_{ij}(s) \le q\},\\
\mathcal{X}_{ij}^q &= \big\{S_i^f \in \mathcal{X}_{ij} : \text{ for some } s \in S_{ij}^q, f(s) = S_j\big\}\\
&\quad \cup \big\{S_j^f \in \mathcal{X}_{ij} : \text{ for some } s \in S_{ij}^q, f(s^*) = S_i\big\},\\
rnk_{ij}^q &= rnk_{ij}{\restriction}\mathcal{X}_{ij}^q.
\end{aligned}$$

Observe that the notions defined by Definition 5.14 have the following meaning in the context of $\mathfrak{C}_{ij}$.

c_i	is the cardinality of $\mathfrak{C}_{ii}$;
$u_{ij}(s)$	is the number of all elements a which realize $S_i^f{\restriction}\{S_j\}$, and for which there is a b, $b \neq a$, such that $\langle a, b\rangle$ realizes s, and b realizes $S_j^f{\restriction}\{S_i\}$;
S_{ij}^q	is the set of 2-types of S_i which are realized in $\mathfrak{C}_{ij}$ by at most q pairs $\langle a, b\rangle$ such that a realizes $S_i^f{\restriction}\{S_j\}$, and b realizes $S_j^f{\restriction}\{S_i\}$;
$\mathcal{X}_{ij}^q$	is the restriction of $\mathcal{X}_{ij}$ to the constellations including types of S_{ij}^q;
rnk_{ij}^q	is the restriction of rnk_{ij} to $\mathcal{X}_{ij}^q$.

Note that $u_{ij}(s)$, c_i, S_{ij}^q, $\mathcal{X}_{ij}^q$, and rnk_{ij}^q are easily computable from $\mathcal{X}$ and rnk in time $O(2^{n^3} \cdot 2^{dn^2}) = O(2^{en^3})$ for some constant e.

Definition 5.15. *Let $\mathcal{X} \subseteq \mathcal{S}_{ind}$, let $\langle \mathcal{X}, rnk\rangle$ be an rnk-indexing of $\mathcal{T}$, and let $\mathcal{B} \subseteq \mathcal{A}$. Given a $\mathcal{L}$-structure $\mathfrak{A}$ we write $\mathfrak{A} \models_{\mathcal{B},\mathcal{X}}^{rnk} \mathcal{T}$ if and only if there exists an rnk-labeling lab of the universe A of $\mathfrak{A}$ such that for every $S_i, S_j \in \mathcal{T}$ and every $a \in A_i^{lab}$,*

$$tp^{\mathfrak{A}}(a, a) = center(lab(a){\restriction}\{S_i\}),$$

$$\{tp^{\mathfrak{A}}(a, b) : b \in A_j^{lab}\} \cap (\mathcal{A}^{\rightarrow} \cup \mathcal{A}^{\leftrightarrow}) = lab(a){\restriction}\{S_j\} \cap \mathcal{B},$$

and for every 2-type $t \in lab(a){\restriction}\{S_j\} \cap \mathcal{B}$ there exists a unique $b \in A_j^{lab}$ such that $tp^{\mathfrak{A}}(a, b) = t(x, y)$.

The above definition says that $\mathfrak{A} \models_{\mathcal{B},\mathcal{X}}^{rnk} \mathcal{T}$ if and only if there exists a rnk-labeling lab of A such that for every A_i^{lab}, A_j^{lab}, and every $a \in A_i^{lab}$, a realizes the subconstellation $lab(a){\restriction}\{S_j\} \cap \mathcal{B}$ in the substructure of $\mathfrak{A}$ with the universe $\{a\} \cup A_j^{lab}$. Note that in case $\mathcal{B} = \mathcal{A}$ the above definition is equivalent to Definition 5.9.

Lemma 5.3. *Let $S_i \in \mathcal{T}$, and assume that $c_i > 2^m$. There is a structure $\mathfrak{C}_{ii}$ such that $\mathfrak{C}_{ii} \models_{\mathcal{X}_{ii}}^{rnk_{ii}} \{S_i, S_i\}$ if and only if the following conditions hold:*

(i) *for every $s \in S_i \cap \mathcal{A}^{\leftrightarrow}$, $u_{ii}(s) = u_{ii}(s^*)$ and $u_{ii}(s)$ is even if $s = s^*$;*

(ii) *S_i is connectable to S_i by some $t \in \mathcal{A}^{-}$;*

(iii) *there exists a structure $\mathfrak{C}'$ such that $\mathfrak{C}' \models_{S_{ii}^q, \mathcal{X}_{ii}'}^{rnk_{ii}'} \{S_i\}$, where $q = 14m$.*

Proof. ($\Rightarrow$) This follows directly from the definition of $u_{ii}(s)$, Lemma 4.1, and Definitions 5.9, 5.15, and 5.14.

($\Leftarrow$) Assume that (i)–(iii) hold. Assume that $\{s_1,\ldots,s_v\} = \{s \in S_i \cap \mathcal{A}^{\leftrightarrow} : u_{ii}(s) > 0\} \cup \{s \in S_i \cap \mathcal{A}^{\rightarrow} : u_{ii}(s) > 0\}$, where $v < card(S_i)$ and $\{s_1,\ldots,s_p\} = S_{ii}^{14m}$. Let lab' be the rnk'_{ii}-labeling of C' given by (iii) and Definition 5.15.

Let C_{ii} be a set of cardinality $\sum_{S_i^f \in \mathcal{X}_{ii}} rnk(S_i^f)$, and let lab be a rnk_{ii}-labeling of C_{ii}. We shall build a structure $\mathfrak{C}_{ii}$ with the universe C_{ii} such that, for every $a \in C_{ii}$, a realizes $lab(a)\restriction\{S_i\}$ in C_{ii}. The structure $\mathfrak{C}_{ii}$ will be built in several steps.

First we define an embedding h of $\mathfrak{C}'$ into $\mathfrak{C}_{ii}$ as follows.

Let $h : C' \xrightarrow[into]{1-1} C_{ii}$, be such that for every $a \in C'$, $lab'(a) = lab(h(a))$. For every $a, b \in h(C')$, such that $tp^{\mathfrak{C}'}(h^{-1}(a), h^{-1}(b)) \in \mathcal{A}^{\leftrightarrow} \cup \mathcal{A}^{\rightarrow}$, define $tp^{\mathfrak{C}_{ii}}(a,b) = tp^{\mathfrak{C}'}(h^{-1}(a), h^{-1}(b))$.

Now it remains to define, for each type $s_l \in \{s_{p+1},\ldots,s_v\}$, the set of pairs of elements of C_{ii} which realize s_l. To do this we shall use the graph-theoretical Lemmas A.2 and A.3 given in the appendix. We proceed by induction. Assume that, for some $l \geq p$, the sets of pairs satisfying types $s_1,\ldots,s_{l-1}$ have been defined. We shall now define set of pairs satisfying s_l.

Case 1. $s_l \in \mathcal{A}^{\leftrightarrow}$. Let $X, Y \subseteq C_{ii}$ be defined as follows:

$X = \{a \in C_{ii} : s_l \in lab(a)\restriction\{S_i\}\}$,

$Y = \{a \in C_{ii} : s_l^* \in lab(a)\restriction\{S_i\}\}$.

Observe that s_l can be realized only by pairs $\langle a, b\rangle$ such that $a \in X$, and $b \in Y$. Also, if $s_l = s_j^*$, for some $j < l$, then the type s_l has been considered. By condition (i), $card(X) = card(Y)$, and if $s_l = s_l^*$, then $card(X)$ is even. Moreover, since $card(X) = u_{ii}(s_l)$, we have $card(X) > 14m$.

Let $G^* = (X \cup Y, E^*)$ be the graph such that $E^* = \{\{a,b\} :$ either $\langle a,b\rangle$ or $\langle b,a\rangle$ realizes a type s_j, $j < l\}$. Then, $d(G^*) < l \leq m$. Let $G = (X \cup Y, E)$ be the graph complement of G^*, and let $n = card(X \cup Y)$. We have $d(G) > n - l + 1 \geq n - m$. Note that if $E(a,b)$, then the type of $\langle a,b\rangle$ has not been specified so far.

Case 1a. $s_l = s_l^*$. By the definition of X and Y, we have $X = Y$. By Lemma A.2, there exists a Hamiltonian cycle $\mu = [a_1,\ldots,a_n]$ in G, and by (i) n is even. For each odd j such that $1 \leq j \leq n$, put $tp^{\mathfrak{C}_{ii}}(a_j, a_{j+1}) = s_l$.

Case 1b. $s_l \in \mathcal{A}^{\leftrightarrow}$, and $s_l \neq s_l^*$. Let $X' = X \setminus Y$, $Y' = Y \setminus X$, $Z = X \cap Y$, $n' = card(X') = card(Y')$, and $n_Z = card(Z)$. We consider two subcases.

Subcase 1ba. $card(Z) \leq 2m$. (See Figure 3.)

Let $G_X = (Z, X', E_X)$, where $E_X = \{\{a,b\} \in E : a \in Z, b \in X'\}$. Then G_X is a bipartite graph such that for every $A \subseteq Z$, $card(\Gamma_{G_X}(A)) > n' - l + 1 > 3m > card(A)$. Therefore, by Lemma A.1, there is a matching E'_X of Z onto $X_Z \subset X'$. For every $a \in X_Z$, $b \in Z$ such that $\{a,b\} \in E'_X$, put $tp^{\mathfrak{C}_{ii}}(a,b) = s_l$.

Similarly, there exists a matching E'_Y from Z onto $Y_Z \subset Y'$ in the graph G_Y defined in the same way as G_X. For every $a \in Z$, $b \in Y_Z$ such that $\{a,b\} \in E'_Y$, put $tp^{\mathfrak{C}_{ii}}(a,b) = s_l$.

Finally, let $G' = (X' \setminus X_Z, Y' \setminus Y_Z, E')$ be the bipartite graph such that $E' = \{\{a,b\} \in E : a \in X' \setminus X_Z, b \in Y' \setminus Y_Z\}$. Put $n'' = card(X' \setminus X_Z) = card(Y' \setminus Y_Z)$. Then, $n'' > 2m$ and $d(G') > n'' - l + 1 > n'' - m$. By Lemma A.3, there exists a matching E_1 from $X' \setminus X_Z$ onto $Y' \setminus Y_Z$. For every $a \in X' \setminus X_Z$, $b \in Y' \setminus Y_Z$ such that $\{a,b\} \in E_1$, put $tp^{\mathfrak{C}_{ii}}(a,b) = s_l$.

Subcase 1bb. $card(X') \leq 2m$.

We have $X \cup Y = X' \dot\cup Z \dot\cup Y'$ and $d(G) \geq n - l + 1 > n - m$. By Lemma A.4, there exists a set $Z' \subset Z$ such that X' and Y' can be matched onto Z'. Let $E_{X'}$ be the matching of X' onto Z', and let $E_{Y'}$ be the matching of Y'

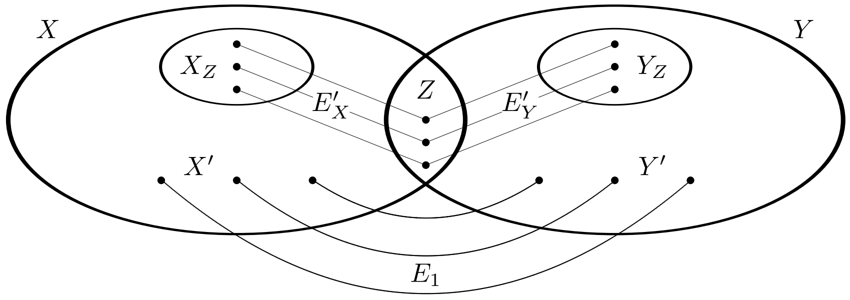


FIG. 3. *Subcase* 1BA.

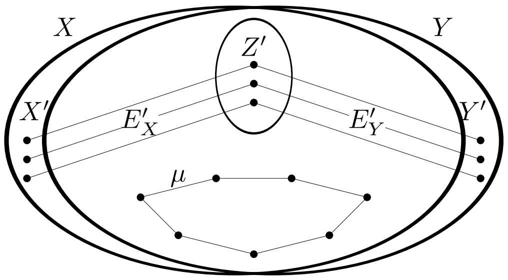


FIG. 4. *Subcase* 1BB.

onto Z'.

For every $a \in X'$, $b \in Z'$ such that $\{a, b\} \in E_{X'}$, put $tp^{\mathfrak{C}_{ii}}(a, b) = s_l$. For every $b \in Z'$, $c \in Y'$ such that $\{b, c\} \in E_{Y'}$, put $tp^{\mathfrak{C}_{ii}}(b, c) = s_l$.

Finally, let $G' = (Z \setminus Z', E')$, where $E' = \{\{a, b\} \subset Z \setminus Z' : \{a, b\} \in E\}$. Put $n'' = card(Z \setminus Z')$. Then, $n'' > 2m$, and $d(G') \geq n'' - l + 1 > n - m$. By Lemma A.2, there exists a Hamiltonian cycle $\mu = [a_1, \ldots, a_{n''}]$ in G'. For every j such that $1 \leq j < n''$, put $tp^{\mathfrak{C}_{ii}}(a_j, a_{j+1}) = s_l$, and put $tp^{\mathfrak{C}_{ii}}(a_{n''}, a_1) = s_l$.

CASE 2. $s_l \in \mathcal{A}^{\rightarrow}$. Let $X = \{a \in C_{ii} : s_l \in S_i^f \restriction \{S_i\}$, where f is a function such that $lab(a) = S_i^f\}$.

Let $G^* = (C_{ii}, E^*)$, where $E^* = \{\{a, b\} :$ either $\langle a, b\rangle$ or $\langle b, a\rangle$ realizes s_j, for some $j < l\}$. Then, $d(G^*) < l \leq m$. Let $G = (C_{ii}, E)$ be the graph complement of G^*, and let $n = card(C_{ii})$. We have $d(G) > n - l + 1 > n - m$, so by Lemma A.2, there exists a Hamiltonian cycle $\mu = [a_1, \ldots, a_n]$ in G. For every j such that $1 \leq j < n$ and $a_j \in X$, put $tp^{\mathfrak{C}_{ii}}(a_j, a_{j+1}) = s_l$, and put $tp^{\mathfrak{C}_{ii}}(a_n, a_1) = s_l$.

Notice that by condition (ii), there exists a type $t \in \mathcal{A}^-$ such that S_i is connectable to S_i by t. Therefore to finish the proof, it suffices to define $tp^{\mathfrak{C}_{ii}}(a, b) = t$, for any $a, b \in C_{ii}$, such that $tp^{\mathfrak{C}_{ii}}(a, b)$ has not been defined yet. □

LEMMA 5.4. *Let $S_i, S_j \in \mathcal{T}$, $i \neq j$, and assume that $c_i, c_j > 3m$. There is a structure $\mathfrak{C}_{ij}$ such that $\mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}$ if and only if the following conditions hold:*

(i) *there are structures $\mathfrak{C}_{ii}$, $\mathfrak{C}_{jj}$ such that $\mathfrak{C}_{ii} \models^{rnk_{ii}}_{\mathcal{X}_{ii}} \{S_i\}$, and $\mathfrak{C}_{jj} \models^{rnk_{jj}}_{\mathcal{X}_{jj}} \{S_j\}$;*

(ii) *$u_{ij}(s) = u_{ji}(s^*)$, for each $s \in S_i \cap \mathcal{A}^{\leftrightarrow}$, and $u_{ji}(s) = u_{ij}(s^*)$, for each $s \in S_j \cap \mathcal{A}^{\leftrightarrow}$;*

(iii) *there exists a structure $\mathfrak{C}'$ such that $\mathfrak{C}' \models^{rnk^q_{ij}}_{S^q_{ij}, \mathcal{X}^q_{ij}} \{S_i, S_j\}$, where $q = 2m$;*

(iv) *S_i is connectable to S_j by some $t \in \mathcal{A}^-$.*

Proof. ($\Rightarrow$) This part of the proof is obvious.

($\Leftarrow$) Assume that (i)–(iv) hold. Let $\{s_1, \ldots, s_v\} = \{s \in S_i \cap \mathcal{A}^{\leftrightarrow} : u_{ij} > 0\} \cup \{s \in (S_i \cup S_j) \cap \mathcal{A}^{\rightarrow} : u_{ij}(s) > 0\}$, where $v < card(S_i \cup S_j)$. Assume that $S^{2m}_{ij} = \{s_1, \ldots, s_p\}$, and $\{s_{r+1}, \ldots, s_v\} = \{s \in (S_i \cup S_j) \cap \mathcal{A}^{\rightarrow} : u_{ij}(s) > 2m\}$.

Let $\mathfrak{C}_{ii}$ and $\mathfrak{C}_{jj}$ be such that (i) holds, and assume that $\mathfrak{C}_{ii}$ and $\mathfrak{C}_{jj}$ are disjoint. Let $C_{ij} = C_{ii} \cup C_{jj}$. For every $a, b \in C_{ii}$, put $tp^{\mathfrak{C}_{ij}}(a, b) = tp^{\mathfrak{C}_{ii}}(a, b)$, and for every $a, b \in C_{jj}$, put $tp^{\mathfrak{C}_{ij}}(a, b) = tp^{\mathfrak{C}_{jj}}(a, b)$. Let lab_{ii} be the rnk_{ii}-labeling of C_{ii}, and lab_{jj} be the rnk_{jj}-labeling of C_{jj} as in Definition 5.9. To define the rnk_{ij}-labeling lab_{ij} of C_{ij}, we put $lab_{ij} = lab_{ii} \cup lab_{jj}$.

Now, for $a \in C_{ii}$ and $b \in C_{jj}$, we shall assign a type to $\langle a, b \rangle$, in such a way that for every $a \in C_{ii}$, a realizes in $\mathfrak{C}_{ij} \restriction C_{jj} \cup \{a\}$ the subconstellation $lab_{ij}(a) \restriction \{S_j\}$, and for every $b \in C_{jj}$, b realizes in $\mathfrak{C}_{ij} \restriction C_{ii} \cup \{b\}$ the subconstellation $lab(b)_{ij} \restriction \{S_i\}$.

As in the proof of Lemma 5.3, the construction proceeds in steps. First, we embed the structure $\mathfrak{C}'$ given by (iii) into $\mathfrak{C}_{ij}$, and then we consider types $s_l \in \{s_{p+1}, \ldots, s_v\}$.

Define

$X = \{a \in C_{ii} : s_l \in lab(a) \restriction \{S_j\}\}$,

$Y = \{a \in C_{jj} : s^*_l \in lab(a) \restriction \{S_i\}\}$.

Now, we deal with types in $\mathcal{A}^{\leftrightarrow}$. Let $s_l \in \mathcal{A}^{\leftrightarrow}$. By (ii) $card(X) = card(Y)$. Moreover, by the definition of u_{ij}, for every $s \in \mathcal{A}^{\leftrightarrow}$, $s \in S_i$ if and only if $s^* \in S_j$. Since $card(X) = u_{ij}(s_l) > 2m$, we can proceed as in Case 1b of the proof of Lemma 5.3 with $Z = \emptyset$.

Finally assume that we have already dealt with the types $s_1, \ldots, s_r$, and we want to realize types in $s_{r+1}, \ldots, s_v$.

Let $E = \{\{a, b\} : a \in C_{ii}, b \in C_{jj}$ and $\langle a, b \rangle$ realizes s_j, $j \leq r\}$, and $G = (C_{ii}, C_{jj}, E)$ be a bipartite graph. For every $a \in C_{ii}$, let $d'(a) = card(lab(a) \restriction \{S_j\})$.

Now, by Lemma A.5, G can be expanded to a bipartite graph $G' = (C_{ii}, C_{jj}, E')$ such that $E \subseteq E'$,

(a) for every $a \in C_{ii}$, $d_{G'}(a) = d'(a)$ and

(b) for every $b \in C_{jj}$, $d_{G'}(b) \leq c_i - m$.

By (a), for every $a \in C_{ii}$, we can find $card(lab(a) \restriction \{S_j\} \cap \mathcal{A}^{\rightarrow})$ elements $b \in C_{jj}$ such that $\langle a, b \rangle \in E' \setminus E$ and assign types in $lab(a) \restriction \{S_j\} \cap \mathcal{A}^{\rightarrow}$ to pairs $\langle a, b \rangle$. On the other hand, by (b), for every $b \in C_{jj}$, we can find $card(lab(b) \restriction \{S_i\} \cap \mathcal{A}^{\rightarrow})$ elements $a \in C_{ii}$ such that $\langle a, b \rangle \notin E'$ to realize the types of $lab(b) \restriction \{S_i\} \cap \mathcal{A}^{\rightarrow}$. ☐

Let

$$u = \sum_{s \in \mathcal{A}^{\leftrightarrow} \cup \mathcal{A}^{\leftarrow}} u_{ij}(s) + \sum_{s \in \mathcal{A}^{\rightarrow}} u_{ji}(s).$$

The integer u is the number of pairs of elements of $\mathfrak{C}_{ij}$ which realizes counting types.

LEMMA 5.5. *Let $S_i, S_j \in \mathcal{T}$, $i \neq j$, and assume that $c_i > 3m$ and $c_j \leq 3m$. There is a structure $\mathfrak{C}_{ij}$ such that $\mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}$ if and only if the following conditions hold:*

(i) *there are structures* $\mathfrak{C}_{ii}$, $\mathfrak{C}_{jj}$ *such that* $\mathfrak{C}_{ii} \models^{rnk_{ii}}_{\mathcal{X}_{ii}} \{S_i\}$, *and* $\mathfrak{C}_{jj} \models^{rnk_{jj}}_{\mathcal{X}_{jj}} \{S_j\}$;

(ii) *for every* $s \in S_i \cap \mathcal{A}^{\leftrightarrow}$, $u_{ij}(s) = u_{ji}(s^*)$, *and for every* $s \in S_j \cap \mathcal{A}^{\leftrightarrow}$, $u_{ji}(s) = u_{ij}(s^*)$;

(iii) *there exists a structure* $\mathfrak{D}$ *with the domain* $D = D_1 \dot{\cup} D_2$, *and there exists a* rnk'*-indexing* $\langle \mathcal{X}', rnk' \rangle$ *of* $\{S_i, S_j\}$ *such that* $card(D_1) \leq 3m^2$, $card(D_1) \leq c_i$, $card(D_2) = c_j$, $\mathcal{X}' \subseteq \mathcal{X}_{ij}$, *for every* $S_j^f \in \mathcal{X}_{ij}$, $S_j^f \in \mathcal{X}'$, *for every* $S_j^f \in \mathcal{X}'$, $rnk'(S_j^f) = rnk_{ij}(S_j^f)$, *for every* $S_i^f \in \mathcal{X}'$, $rnk'(S_i^f) \leq rnk_{ij}(S_i^f)$ *and*

$$\mathfrak{D} \models^{rnk'}_{\mathcal{X}'} \{S_i, S_j\};$$

(iv) *for every* $S_i^f \in \mathcal{X}_{ij}$, $card((S_i^f \restriction \{S_j\}) \setminus \{center(S_i)\}) \leq c_j$;

(v) *if* $u < c_i \cdot c_j$, *then* S_i *is connectable to* S_j *by some* $t \in \mathcal{A}^-$.

Proof. ($\Rightarrow$) This direction is obvious.

($\Leftarrow$) Assume that (i)–(v) hold. Assume the structures $\mathfrak{C}_{ii}$ and $\mathfrak{C}_{jj}$ given by (i) have disjoint universes. Put $C_{ij} = C_{ii} \dot{\cup} C_{jj}$, for every $a, b \in C_{ii}$, put $tp^{\mathfrak{C}_{ij}}(a,b) = tp^{\mathfrak{C}_{ii}}(a,b)$, and for every $a, b \in C_{jj}$, put $tp^{\mathfrak{C}_{ij}}(a,b) = tp^{\mathfrak{C}_{jj}}(a,b)$. Let lab_{ii} be the rnk_{ii}-labeling of C_{ii}, and lab_{jj} be the rnk_{jj}-labeling of C_{jj} as in the definition 5.9. Define the rnk_{ij}-labeling lab_{ij} of the set C_{ij} by putting $lab_{ij} = lab_{ii} \cup lab_{jj}$.

Now, by (iii), define an embedding h of $\mathfrak{D}$ into $\mathfrak{C}_{ij}$ as follows.

Let $h : D \xrightarrow[into]{1-1} C_{ij}$ be a mapping such that for every $a \in D$, $lab'(a) = lab(h(a))$. For every $a, b \in h(D)$ such that $a \in C_{ii}$, $b \in C_{jj}$ put $tp^{\mathfrak{C}_{ij}}(a,b) = tp^{\mathfrak{D}}(h^{-1}(a), h^{-1}(b))$.

Note that by Definition 5.15, for every $a \in C_{jj} \dot{\cup} h(D_1)$, $C_a^{\mathfrak{C}_{ij} \restriction h(D)} = lab(a) \restriction \{S_i, S_j\}$. Moreover, by (ii) and (iii), for every indexed constellation $S_i^f \in \mathcal{X}_{ij}$, if there is a type $s \in S_i \cap \mathcal{A}^{\leftrightarrow}$ such that $f(s) = S_j$, then $rnk'(S_i^f) = rnk(S_i^f)$. It follows that if $a \in C_{ii} \setminus h(D)$, then $s \notin \mathcal{A}^{\leftrightarrow}$ for each $s \in lab(a) \restriction \{S_j\}$. By (iv), for every $a \in C_{ii} \setminus h(D)$, for every $s \in lab(a) \restriction \{S_j\})$, find $b \in C_{jj}$ such that $tp^{\mathfrak{C}_{ij}}(a,b)$ has not been defined, and put $tp^{\mathfrak{C}_{ij}}(a,b) = s$. To finish the proof of Lemma 5.5, for every $a \in C_{ii}$, for every $b \in C_{jj}$, if $tp^{\mathfrak{C}_{ij}}(a,b)$ has not been defined, then put $tp^{\mathfrak{C}_{ij}}(a,b) = t$, where $t \in \mathcal{A}^-$ and S_i is connectable to S_j by t (cf. (v)). □

COROLLARY 5.16. (c5) *can be checked in exponential time.*

Proof. By decomposition theorem (Lemma 5.2), it suffices to check whether, for every $S_i, S_j \in \mathcal{S}_1 \cup \mathcal{S}_2$,

$$(*) \text{ there exists } \mathfrak{C}_{ij} \text{ such that } \mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}.$$

In the case $\boldsymbol{i = j}$, if $c_i \leq 2^m$, (*) can be checked by guessing the structure $\mathfrak{C}_{ii}$ of cardinality c_i and then verifying if $\mathfrak{C}_{ii} \models^{rnk_{ii}}_{\mathcal{X}_{ii}} \{S_i, S_i\}$. This will take no more than $O(2^{m^2}) = O(2^{n^2})$ steps. If $card(C_{ii}) > 2^m$, then by Lemma 5.3, it suffices to verify conditions (i)–(iii). This can be done in time $O(2^{dn^3})$ for some constant d.

In the case $\boldsymbol{i \neq j}$, if $c_i, c_j \leq 3m$, it suffices to guess a structure $\mathfrak{C}_{ij}$ of cardinality $c_i + c_j$ and verify if $\mathfrak{C}_{ij} \models^{rnk_{ij}}_{\mathcal{X}_{ij}} \{S_i, S_j\}$. This can be done in time polynomial with respect to m. To check whether (*) holds for $c_i > 3m$, it suffices to verify conditions (i)–(iv) of Lemma 5.4 or conditions (i)-(v) of Lemma 5.5. This also can be done in time $O(2^{dn^3})$ for some constant d.

In each of the cases above, checking whether (*) holds can be done in time $O(2^{dn^3})$, so it takes at most $O((2^{dn^3})^2) = O(2^{cn^3})$ steps to verify that (*) holds for every i, j. This finished the proof and completes the proof of Theorem 5.1. □

Appendix. We assume that the reader is familiar with the basic notions of graph theory. We use standard notation of graph theory (see, e.g., [4]).

In this paper a *graph* $G = (X, E)$ is a *finite* set X of nodes and a set E of edges, which are unordered pairs of nodes. For $x \in X$ we denote by $\Gamma_G(x)$ the set of neighbors of x, i.e., the set $\{y : \{x, y\} \in E\}$, and, for $A \subseteq X$, we put $\Gamma_G(A) = \bigcup_{a \in A} \Gamma_G(a)$. The degree of a node x, denoted by $d_G(x)$, is the number of neighbors of x. By $d(G)$ we denote the minimal value of $d_G(x)$. Given a graph $G = (X, E)$, a matching is defined as a set $E_0 \subseteq E$ such that, for each pair $\{u, v\}, \{u', v'\} \in E_0$ of edges, we have $\{u, v\} \cap \{u', v'\} = \emptyset$. A graph is *bipartite* if its nodes can be partitioned into two sets X_1, X_2 such that no two nodes in the same set are adjacent; such a bipartite graph is often denoted as $G = (X_1, X_2, E)$. Given a bipartite graph $G = (X, Y, E)$ and we say that X is matched into Y if there is a matching $E_0 \subseteq E$ such that for every $x \in X$ there exists $y \in Y$ such that $\{x, y\} \in E_0$.

Let m be a fixed nonnegative integer. The proof of the main result of this paper (Theorem 5.1) heavily depends on the following lemmas.

Lemma A.1 (König–Hall theorem (see [21]); cf. [4, p. 134]). *In a bipartite graph $G = (X, Y, E)$, X can be matched into Y if and only if $card(\Gamma_G(A)) \geq card(A)$ for every $A \subseteq X$.*

Lemma A.2. *Let G be a graph with n nodes and with $d(G) \geq n - m$. If $n > 2m$, then G has a Hamiltonian cycle.*

Proof. This is an easy consequence of the theorem by Bondy [6] (cf. [4, p. 212]) which says that *a graph G with $n \geq 3$ nodes, and with degrees $d_1 \leq \cdots \leq d_n$ has a Hamiltonian cycle if for every i, j such that $i \neq j$, $d_i \leq i$ and $d_j \leq j$, we have $d_i + d_j \geq n$.*

In fact, we have $d(G) \geq n - m$, so $d_i + d_j \geq n$ always holds for $n > 2m$. $\square$

Lemma A.3. *If $G = (X, Y, E)$ is a bipartite graph such that $card(X) = card(Y) = n$, $d(G) \geq n - m$ and $n > 2m$, then X can be matched into Y.*

Proof. We use Lemma A.1. Towards a contradiction, assume that there exists $A \subseteq X$ such that $card(\Gamma_G(A)) < card(A)$. Then $card(\Gamma_G(A)) < n$ and, since $d(G) \geq n - m$, $card(\Gamma_G(A)) \geq n - m$ and $card(A) > n - m$. Let $y \in Y \setminus \Gamma_G(A)$. For every $x \in X$, if $x \in \Gamma_G(y)$, then $x \notin A$. Moreover, $card(\Gamma_G(y)) \geq n - m$ and so, $card(A) < m$. This gives a contradiction if $n > 2m$. $\square$

Lemma A.4. *Let $G = (V, E)$ be a graph with n nodes such that $d(G) \geq n - m$, and assume that $V = X' \dot{\cup} Z \dot{\cup} Y'$, where $card(X') = card(Y') \leq 2m$. If $n > 14m$, then there exists $Z' \subseteq Z$ such that both X' and Y' can be matched onto Z'.*

Proof. Let $X' = \{a_1, \dots, a_k\}$, $Y' = \{b_1, \dots, b_k\}$, where $k \leq 2m$. Since $d(G) \geq n - m$, we have $card(\Gamma_G(a_1)) \geq n - m$. We claim that there is an element $c_1 \in \Gamma_G(a_1) \cap Z$ such that $c_1 \in \Gamma_G(b_1) \cap Z$. Indeed, $card(\Gamma_G(a_1) \cap Y) \geq n - m - 4m = n - 5m$, and $card(\Gamma_G(b_1) \cap Z) \geq n - 5m$. Therefore, $\Gamma_G(a_1) \cap \Gamma_G(b_1) \cap Z \neq \emptyset$, provided $n > 10m$. Similarly, $\Gamma_G(a_i) \cap \Gamma_G(b_i) \cap (Z \setminus \{c_1, \dots, c_{i-1}\}) \neq \emptyset$, provided $n > 10m + 2(i - 1)$. $\square$

Lemma A.5. *Let $G = (X, Y, E)$ be a bipartite graph such that for every $b \in Y$, $d_G(b) \leq card(X) - m$. Let $d' : X \mapsto \{0, \dots, m\}$ and for every $a \in X$, $d_G(a) \leq d'(a)$. If $card(X), card(Y) > 3m$, then there exists a bipartite graph $G' = (X, Y, E')$ such that $E \subseteq E'$, for every $a \in X$, $d_{G'}(a) = d'(a)$, and for every $b \in Y$, $d_{G'}(b) \leq card(X) - m$.*

Proof. Let $G = (X, Y, E)$ be a bipartite graph such that for every $b \in Y$, $d_G(b) \leq card(X) - m$, and let $d' : X \mapsto \{0, \dots, m\}$ be a function such that for every $a \in X$, $d_G(a) \leq d'(a)$.

To build the graph G' we will add new edges to the graph G repeating the following operation until we get a graph as needed.

(*) Let a be an element of X such that $d_G(a) < d'(a)$. Find an element $b \in Y$ such that $\{a,b\} \notin E$ and for every $c \in Y$ with $\{a,c\} \notin E$, $d_G(b) \leq d_G(c)$. Put $E = E \cup \{a,b\}$.

We will now show that an element b as above exists, and that the operation (*) preserves assumptions of the lemma.

Since $d_G(a) < m < card(Y)$, there exist at least $card(Y) - m$ elements c in Y such that $\{a,c\} \notin E$. We claim that among these elements there is an element b such that $d_G(b) < card(X) - m$. In fact, towards a contradiction, assume that $d_G(b) = card(X) - m$, for each $c \in Y$ such that $\{a,c\} \notin E$. Then

$$\sum_{c\in Y} d_G(c) = \sum_{c\in Y\setminus\Gamma_G(a)} d_G(c) + \sum_{c\in\Gamma_G(a)} d_G(c) \geq (card(Y) - m)(card(X) - m).$$

On the other hand, $d_G \leq m$, for every $e \in X$, which gives

$$\sum_{c\in Y} d_G(c) = \sum_{e\in X} d_G(e) < m \cdot card(X).$$

By the above inequalities,

$$m \cdot card(X) > (card(Y) - m)(card(X) - m)$$

and hence,

$$card(X)card(Y) - m(card(X) + card(Y)) - m(card(X) - m) < 0. \tag{A.1}$$

Assume $card(X) \geq card(Y)$. Then,

$$\begin{aligned} card(X)card(Y) - m(card(X) + card(Y)) - m(card(X) - m) \\ \geq card(X)card(Y) - 3m \cdot card(X) + m^2 \\ > card(X)(card(Y) - 3m) > 0, \text{provided } card(Y) > 3m. \end{aligned}$$

The last inequality contradicts (A.1). $\square$

Acknowledgments. We would like to thank the referees for their valuable comments that helped to improve the presentation.

REFERENCES

[1] W. Ackermann, *Über die Erfüllbarkeit gewisser Zählausdrücke*, Math. Ann., 100 (1928), pp. 638–649.
[2] W. Ackermann, *Beiträge zum Entscheidungsproblem der mathematischen Logik*, Math. Ann., 112 (1936), pp. 419–432.
[3] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.
[4] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1971.
[5] P. Bernays and M. Schönfinkel, *Zum Entscheidungsproblem der mathematischen Logik*, Math. Ann., 99 (1928), pp. 342–372.
[6] J. A. Bondy, *Properties of graphs with constraints and on degrees*, Studia Sci. Math. Hungar., 4 (1969), pp. 473–475.
[7] E. Börger, E. Grädel, and Y. Gurevich, *The Classical Decision Problem*, Perspect. Math. Logic, Springer-Verlag, Berlin, 1997.
[8] A. Borgida, *On the relative expressive power of description logics and predicate calculus*, Artificial Intelligence, 82 (1996), pp. 353–367.
[9] B. Dreben and W. Goldfarb, *The Decision Problem: Solvable Classes of Quantificational Formulas,*, Addison-Wesley, Reading, MA, 1979.

[10] A. Ehrenfeucht, *An application of games to the completeness problem for formalised theories*, Fund. Math., 49 (1961), pp. 129–141.
[11] K. Gödel, *Ein Spezialfall des Entscheidungsproblem des theoretischen Logik*, Ergebnisse eines mathematischen Kolloquiums, 2 (1932), pp. 27–28.
[12] K. Gödel, *Zum Entscheidungsproblem des logischen Funktionenkalküls*, Monatshefte für Mathematik und Physik, 40 (1933), pp. 433–443.
[13] W. Goldfarb, *On Decision Problems for Quantification Theory*, Ph.D. thesis, Harvard University, Cambridge, MA, 1974. The abstract appears in Notices Amer. Math. Soc., 21 (1974), p. 280.
[14] W. Goldfarb, *The unsolvability of the Gödel class with identity*, J. Symbolic Logic, 49 (1984), pp. 1237–1252.
[15] E. Grädel, P. Kolaitis, and M. Vardi, *On the decision problem for two-variable first-order logic*, Bull. Symbolic Logic, 3 (1997), pp. 53–69.
[16] E. Grädel, M. Otto, and E. Rosen, *Two-variable logic with counting is decidable*, in 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, 1997, pp. 306–317.
[17] E. Grädel, M. Otto, and E. Rosen, *Undecidability Results on Two-Variables Logics*, in Proceedings of 14th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 1200, Springer–Verlag, Berlin, 1997.
[18] Y. Gurevich, *The decision problem for standard classes*, J. Symbolic Logic, 41 (1976), pp. 460–464.
[19] Y. Gurevich, *On the classical decision problem*, in Current Trends in Theoretical Computer Science, G. Rosenberd, and A. Salomaa, eds., World Scientific, River Edge, NJ, 1993, pp. 254–265.
[20] Y. Gurevich and S. Shelah, *Random models and the Gödel case of the decision problem*, J. Symbolic Logic, 48 (1983), pp. 1120–1124.
[21] P. Hall, *On representations of subsets*, J. London Math. Soc., 10 (1934), pp. 26–30.
[22] W. van der Hoek and M. De Rijke, *Counting objects*, J. Logic Comput., 5 (1995), pp. 325–345.
[23] N. D. Jones and A. L. Selman, *Turing machines and the spectra of first-order formulas*, J. Symbolic Logic, 39 (1974), pp. 139–150.
[24] L. Kalmár, *Über die Erfüllbarkeit derjenigen Zählausdrücke, welche in der Normalform zwei benachbarte Allzeichen enthalten*, Math. Ann., 108 (1933), pp. 466–484.
[25] H. Lewis, *Unsolvable Classes of Quantificational Formulas*, Addison-Wesley, Reading, MA, 1979.
[26] H. Lewis and W. Goldfarb, *The decision problem for formulas with a small number of atomic subformulas*, J. Symbolic Logic, 38 (1973), pp. 471–480.
[27] H. R. Lewis, *Complexity results for classes of quantificational formulas*, J. Comput. System Sci., 21 (1980), pp. 317–353.
[28] M. Mortimer, *On languages with two variables*, Z. f. Logik Grundlagen Math., 21 (1975), pp. 135–140.
[29] L. Pacholski, W. Szwast, and L. Tendera, *Complexity of two-variable logic with counting*, in 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, 1997, pp. 318–327.
[30] M. Rabin, *Decidability of second-order theories and automata on infinite trees*, Trans. Amer. Math. Soc., 141 (1969), pp. 1–35.
[31] D. Scott, *A decision method for validity of sentences in two variables*, J. Symbolic Logic, 27 (1962), p. 477.
[32] S. Shelah, *Decidability of a portion of the predicate calculus*, Israel J. Math., 28 (1977), pp. 32–44.
[33] T. Skolem, *Untersuchungen über die Axiome des Klassenkalküls und über Produktations- und Summationsprobleme, welche gewisse Klassen von Aussagen betreffen*, Skrifter utgit au Vidensk. i Kristiania I, Math.-nat. Klasse, 3 (1919), p. 37.
[34] T. Skolem, *Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen*, Norsk Vid-Akad. Oslo Mat.-Natur Kl. Skr., 4 (1920).
[35] T. Skolem, *Über die mathematische Logik*, Norsk Mat. Tidsskrift, 106 (1928), pp. 125–142.
[36] W. Szwast and L. Tendera, *First-Order Two-Variable Logic Is in NEXPTIME*, manuscript.
[37] H. Wang, *Dominoes and the $\forall\exists\forall$-case of the decision problem*, in Proceedings Symposium on Mathematical Theory of Automata, New York, Brooklyn Polytechnic Institute, 1962, pp. 23–55.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1118–1131

MAKING NONDETERMINISM UNAMBIGUOUS[*]

KLAUS REINHARDT[†] AND ERIC ALLENDER[‡]

Abstract. We show that in the context of nonuniform complexity, nondeterministic logarithmic space bounded computation can be made unambiguous. An analogous result holds for the class of problems reducible to context-free languages. In terms of complexity classes, this can be stated as

$$\mathrm{NL/poly} = \mathrm{UL/poly},$$
$$\mathrm{LogCFL/poly} = \mathrm{UAuxPDA}(\log n, n^{O(1)})/\mathrm{poly}.$$

Key words. nondeterministic space, unambiguous computation, NLOG, ULOG, LogCFL

AMS subject classifications. 68Q15, 68Q10

PII. S0097539798339041

1. Introduction.

In this paper, we combine two very useful algorithmic techniques (the inductive counting technique of [Imm88, Sze88] and the isolation lemma of [MVV87]) to give a simple proof that two fundamental concepts in complexity theory coincide in the context of nonuniform computation: nondeterminism and unambiguity.

Unambiguous computation has been the focus of much attention over the past three decades. The notion of nondeterminism is a fundamental notion in many areas of computer science, and the version of nondeterminism where *at most one* nondeterministic path is accepting has proved to be one of the most meaningful restrictions of nondeterminism to study. For example,

- unambiguous context-free languages form one of the most important subclasses of the class of context-free languages;
- the complexity class UP (unambiguous polynomial time) was first defined and studied by Valiant [Val76], and a necessary precondition for the existence of one-way functions is for P to be properly contained in UP [GS88].

Although UP is one of the most intensely studied subclasses of nondeterministic polynomial time (NP), it is neither known nor widely believed that UP contains any sets that are hard for NP under any interesting notion of reducibility. (Recall that Valiant and Vazirani showed that "*Unique.Satisfiability*"—the set of all Boolean formulae with *exactly one* satisfying assignment—is hard for NP under probabilistic reductions [VV86]. However, the language *Unique.Satisfiability* is hard for coNP under $\leq_m^p$ reductions, and thus is not in UP unless NP = coNP.)

Nondeterministic and unambiguous space-bounded computation have also been the focus of much work in computer science. For instance, the question of whether every context-sensitive language has an unambiguous context-sensitive grammar is

[*]Received by the editors May 15, 1998; accepted for publication (in revised form) April 13, 1999; published electronically February 1, 2000. A preliminary version of this paper appeared in the Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), 1997.

http://www.siam.org/journals/sicomp/29-4/33904.html

[†]Wilhelm-Schickard Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany (reinhard@informatik.uni-tuebingen.de). This author was supported in part by the DFG project La 618/3-1 KOMET.

[‡]Department of Computer Science, Rutgers University, Piscataway, NJ 08855-1179 (allender@cs.rutgers.edu). This author was supported in part by NSF grants CCR-9509603 and CCR-9734918. This work was performed while the author was a visiting scholar at the Wilhelm-Schickard Institut für Informatik, Universität Tübingen, supported by DFG grant TU 7/117-1.

really a question about whether nondeterministic and unambiguous linear space coincide. This question remains open. In recent years, nondeterministic logspace (NL) has been the focus of much attention, in part because NL captures the complexity of many natural computational problems [Jon75]. The unambiguous version of NL, denoted UL, was first explicitly defined and studied in [BJLR91, AJ93]. A language A is in UL if and only if there is a nondeterministic logspace machine M accepting A such that, for every x, M has at most one accepting computation on input x.

Our results indicate that NL and UL are probably equal, but we cannot prove this equality. In order to state our theorem, we first need to discuss the issue of *uniformity.*

Complexity classes such as P, NP, and NL that are defined in terms of machines are known as "uniform" complexity classes, in contrast to "nonuniform" complexity classes, which are defined most naturally in terms of families of circuits $\{C_n\}$, with a circuit for each input length. In order to make a circuit complexity class "uniform," it is necessary to require that the function $n \mapsto C_n$ be "easy" to compute in some sense. (We will consider "logspace-uniform" circuits, where the function $n \mapsto C_n$ can be computed in space $\log n$.) P and NL (and many other uniform complexity classes) have natural definitions in terms of uniform circuits; for instance, NL can be characterized in terms of switching-and-rectifier networks (see, e.g. [Raz92, Raz90]) and skew circuits [Ven92]. Uniform complexity classes can be used to give characterizations of the nonuniform classes, too, using a formalism presented in [KL82]: Given any complexity class $\mathcal{C}$, $\mathcal{C}$/poly is the class of languages A for which there exists a sequence of "advice strings" $\{\alpha(n) \mid n \in \mathbf{N}\}$ and a language $B \in \mathcal{C}$ such that $x \in A$ if and only if $(x, \alpha(|x|)) \in B$.

Our main result is that NL/poly is equal to UL/poly.

(It is worth emphasizing that, in showing the equality UL/poly = NL/poly, we must show that for every B in NL/poly, there is a nondeterministic logspace machine M that never has more than one accepting path on any input, and there is an advice sequence $\alpha(n)$ such that $M(x, \alpha(|x|))$ accepts if and only if $x \in B$. This is stronger than merely saying that there is an advice sequence $\alpha(n)$ and a nondeterministic logspace machine such that $M(x, \alpha(|x|))$ never has more than one accepting path, and it accepts if and only if $x \in B$.)

Our work extends the earlier work of Wigderson and Gál. Motivated in part by the question of whether a space-bounded analogue of the result of [VV86] could be proved, Wigderson [Wig94, GW96] proved the inclusion NL/poly $\subseteq \oplus$L/poly. (An alternative proof of this inclusion is sketched in [Reg97, p. 284].) This is a weaker statement than NL $\subseteq \oplus$L, which is still not known to hold. $\oplus$L is the class of languages A for which there is a nondeterministic logspace bounded machine M such that $x \in A$ if and only if M has an odd number of accepting computation paths on input x.

In the proof of the main result of [Wig94, GW96], Wigderson observed that a simple modification of his construction produces graphs in which the shortest distance between every pair of nodes is achieved by a unique path. We will refer to such graphs in the following as *min-unique graphs.* Wigderson wrote: "We see no application of this observation." The proof of our main result is just such an application.

2. Nondeterministic logspace.

The s-t connectivity problem takes as input a directed graph with two distinguished vertices s and t and determines if there is a path in the graph from s to t. It is well known that this is a complete problem for NL [Jon75].

The following lemma is implicit in [Wig94, GW96], but for completeness we make it explicit here.

LEMMA 2.1. *There is a logspace-computable function f and a sequence of "advice strings" $\{\alpha(n) \mid n \in \mathbf{N}\}$ (where $|\alpha(n)|$ is bounded by a polynomial in n) with the following properties:*

- *For any directed acyclic graph G on n vertices, $f(G, \alpha(n)) = \langle G_1, \ldots, G_{n^2}\rangle$.*
- *For each i, the directed acyclic graph G_i has an s-t path if and only if G has an s-t path.*
- *There is some i such that G_i is a min-unique graph.*

Proof. We first observe that a standard application of the isolation lemma technique of [MVV87] shows that, if each edge in G is assigned a weight in the range $[1, 4n^4]$ uniformly and independently at random, then with probability at least $\frac{3}{4}$, for any two vertices x and y such that there is a path from x to y, there is only one path having minimum weight. (Sketch: The probability that there is more than one minimum weight path from x to y is bounded by the sum, over all edges e, of the probability of the event $\text{BAD}(e, x, y)$ ::= "e occurs on one minimum-weight path from x to y and not on another." Given any weight assignment w' to the edges in G other than e, there is at most one value z with the property that, if the weight of e is set to be z, then $\text{BAD}(e, x, y)$ occurs. Thus the probability that there are two minimum-weight paths between two vertices is bounded by $\sum_{x,y,e}\sum_{w'} \text{BAD}(e, x, y|w')\text{Prob}(w') \leq \sum_{x,y,e}\sum_{w'} 1/(4n^4)\text{Prob}(w') = \sum_{x,y,e} 1/(4n^4) \leq 1/4$.)

Our advice string α will consist of a sequence of n^2 weight functions, where each weight function assigns a weight in the range $[1, 4n^4]$ to each edge. (There are $A(n) = 2^{O(n^5)}$ such advice strings possible for each n.) Our logspace-computable function f takes as input a digraph G and a sequence of n^2 weight functions and produces as output a sequence of graphs $\langle G_1, \ldots, G_{n^2}\rangle$, where graph G_i is the result of replacing each directed edge $e = (x, y)$ in G by a directed path of length j from x to y, where j is the weight given to e by the ith weight function in the advice string. Note that, if the ith weight function satisfies the property that there is at most one minimum weight path between any two vertices, then G_i is a min-unique graph. To see this, it suffices to observe that, for any two vertices x and y of G_i, either (a) there exist vertices u and v such that x and y were both added in replacing the edge (u, v) (in which case there is exactly one path connecting u to v, or (b) there are vertices x' and y' such that

- x' and y' are vertices of the original graph G, and they lie on every path between x and y,
- there is only one path from x to x' and only one path from y' to y, and
- the minimum weight path from x' to y' is unique.

Let us call an advice string "bad for G" if none of the graphs G_i in the sequence $f(G)$ is a min-unique graph. For each G, the probability that a randomly-chosen advice string α is bad is bounded by (probability that G_i is not min-unique)$^{n^2} \leq (1/4)^{n^2} = 2^{-2n^2}$. Thus the total number of advice strings that are bad for some G is at most $2^{n^2}(2^{-2n^2}A(n)) < A(n)$. Thus there is some advice string $\alpha(n)$ that is not bad for *any* G. ∎

THEOREM 2.2. *NL⊆UL/poly.*

Proof. It suffices to present a UL/poly algorithm for the s-t connectivity problem.

We show that there is a nondeterministic logspace machine M that takes as input a sequence of digraphs $\langle G_1, \ldots, G_r\rangle$ and processes each G_i in sequence, with the following properties:

```
Input $(G, k, c_k, \Sigma_k, v)$
$count := 0$; $sum := 0$; $path.to.v :=$ false;
for each $x \in V$ do
    Guess nondeterministically if $d(x) \leq k$.
    if the guess is $d(x) \leq k$, then
        begin
         Guess a path of length $l \leq k$ from $s$ to $x$ (If this fails, then halt and reject).
         $count := count + 1$; $sum := sum + l$;
         if $x = v$, then $path.to.v :=$ true;
        end
endfor
if $count = c_k$ and $sum = \Sigma_k$,
    then return the Boolean value of $path.to.v$
    else halt and reject
end.procedure
```

FIG. 1. *An unambiguous routine to determine if $d(v) \leq k$.*

- If G_i is not min-unique, M has a unique path that determines this fact and goes on to process G_{i+1};[1] all other paths are rejecting.
- If G_i is a min-unique graph with an s-t path, then M has a unique accepting path.
- If G_i is a min-unique graph with no s-t path, then M has no accepting path.

Combining this routine with the construction of Lemma 2.1 yields the desired UL/poly algorithm.

Our algorithm is an enhancement of the inductive counting technique of [Imm88] and [Sze88]. We call this the *double counting* technique since in each stage we count not only the number of vertices having distance at most k from the start vertex, but also the sum of the lengths of the shortest path to each such vertex. In the following description of the algorithm, we denote these numbers by c_k and Σ_k, respectively.

Let us use the notation $d(v)$ to denote the length of the shortest path in a graph G from the start vertex to v. (If no such path exists, then $d(v) = n + 1$.) Thus, using this notation, $\Sigma_k = \sum_{\{x \mid d(x) \leq k\}} d(x)$.

A useful observation is that *if the subgraph of G induced by vertices having distance at most k from the start vertex is min-unique* (and if the correct values of c_k and Σ_k are provided), then an unambiguous logspace machine can, on input (G, k, c_k, Σ_k, v), compute the Boolean predicate "$d(v) \leq k$". This is achieved with the routine shown in Figure 1.

To see that this routine truly is unambiguous if the preconditions are met, note the following:

- If the routine ever guesses incorrectly for some vertex x that $d(x) > k$, then the variable *count* will never reach c_k and the routine will reject. Thus the only paths that run to completion guess correctly exactly the set $\{x \mid d(x) \leq k\}$.
- If the routine ever guesses incorrectly the length l of the shortest path to x, then if $d(x) > l$ no path of length l will be found, and if $d(x) < l$, then the

[1]More precisely, our routine will check if, for every vertex x, there is at most one minimal-length path from the start vertex to x. This is sufficient for our purposes. A straightforward modification of our routine would provide an unambiguous logspace routine that would determine if the entire graph G_i is a min-unique graph.

```
Input $(G, k, c_{k-1}, \Sigma_{k-1})$
Output $(c_k, \Sigma_k)$, and also the flag $BAD.GRAPH$

$c_k := c_{k-1}$; $\Sigma_k := \Sigma_{k-1}$;
for each vertex $v$ do
  if $\neg(d(v) \leq k-1)$, then
    for each $x$ such that $(x, v)$ is an edge do
      if $d(x) \leq k-1$, then
        begin
        $c_k := c_k + 1$; $\Sigma_k := \Sigma_k + k$;
        for $x' \neq x$ do
          if $(x', v)$ is an edge and $d(x') \leq k-1$, then $BAD.GRAPH :=$ true
        endfor
        end
    endfor
endfor
At this point, the values of $c_k$ and $\Sigma_k$ are correct.
```

FIG. 2. *Computing c_k and Σ_k.*

```
Input $(G)$
$BAD.GRAPH :=$ false; $c_0 := 1; \Sigma_0 := 0; k := 0$;
repeat
    $k := k + 1$;
    compute $c_k$ and $\Sigma_k$ from $(c_{k-1}, \Sigma_{k-1})$;
until $c_{k-1} = c_k$ or $BAD.GRAPH =$ true.
If $BAD.GRAPH =$ false, then there is an $s$-$t$ path in $G$ if and only if $d(t) \leq k$.
```

FIG. 3. *Finding an s-t path in a min-unique graph.*

variable *sum* will be increased by a value greater than $d(x)$. In the latter case, at the end of the routine, *sum* will be greater than Σ_k, and the routine will reject.

Clearly, the subgraph having distance at most 0 from the start vertex is min-unique, and $c_0 = 1$ and $\Sigma_0 = 0$. A key part of the construction involves computing c_k and Σ_k from c_{k-1} and Σ_{k-1}, at the same time checking that the subgraph having distance at most k from the start vertex is min-unique. It is easy to see that c_k is equal to c_{k-1} plus the number of vertices having $d(v) = k$. Note that $d(v) = k$ if and only if there is some edge (x, v) such that $d(x) \leq k-1$ and it is not the case that $d(v) \leq k-1$. (Note that both of these latter conditions can be determined in UL, as discussed above.) The subgraph having distance at most k from the start vertex *fails* to be a min-unique graph if and only if there exist some v and x as above, as well as some other $x' \neq x$ such that $d(x') \leq k-1$ and there is an edge (x', v). The code shown in Figure 2 formalizes these considerations.

Recall that we are building an algorithm that takes as input a sequence of graphs $\langle G_1, \dots, G_r \rangle$ and processes each graph G in the sequence in turn, as outlined at the start of this proof. Searching for an s-t path in a graph G in the sequence is now expressed by the routine shown in Figure 3.

We complete the proof by describing how our algorithm processes the sequence $\langle G_1, \dots, G_r \rangle$, as outlined at the start of the proof. Each G_i is processed in turn. If

G_i is not min-unique (or more precisely, if the subgraph of G_i that is reachable from the start vertex is not a min-unique graph), then one unique computation path of the routine returns the value *BAD.GRAPH* and goes on to process G_{i+1}; all other computation paths halt and reject. Otherwise, if G_i is min-unique, the routine has a unique accepting path if G_i has an s-t path, and if this is not the case, the routine halts with no accepting computation paths. ∎

COROLLARY 2.3. *NL/poly = UL/poly.*

Proof. Clearly UL/poly is contained in NL/poly. It suffices to show the converse inclusion. Let A be in NL/poly. By definition, there is a language $B \in$ NL and there is an advice sequence α_n such that x is in A if and only if $(x, \alpha_{|x|})$ is in B. By the preceding theorem, B is in UL/poly, and thus there is a C in UL and an advice sequence β_n such that (x, α_n) is in B if and only if $((x, \alpha_{|x|}), \beta_{|x|+|\alpha_{|x|}|})$ is in C. It is now obvious how to construct the desired advice sequence from α_n and $\beta_{n+|\alpha_n}$. ∎

3. LogCFL.

3. LogCFL. LogCFL is the class of problems logspace-reducible to a context-free language. Two important and useful characterizations of this class are summarized in the following proposition. (SAC^1 and AuxPDA$(\log n, n^{O(1)})$ are defined in the following paragraphs.)

PROPOSITION 3.1 (see [Sud78, Ven91]). *LogCFL* $=$ *AuxPDA*$(\log n, n^{O(1)})$ $=$ *SAC*^{1}.

An auxiliary pushdown automaton (AuxPDA) is a nondeterministic Turing machine with a read-only input tape, a space-bounded worktape, and a pushdown store that is not subject to the space-bound. The class of languages accepted by auxiliary pushdown automata in space $s(n)$ and time $t(n)$ is denoted by AuxPDA$(s(n), t(n))$. If an AuxPDA satisfies the property that, on every input x, there is at most one accepting computation, then the AuxPDA is said to be *unambiguous.* This gives rise to the class UAuxPDA$(s(n), t(n))$.

SAC^1 is the class of languages accepted by logspace-uniform semiunbounded circuits of depth $O(\log n)$; a circuit family is semiunbounded if the AND gates have fan-in 2 and the OR gates have unbounded fan-in.

Not long after NL was shown to be closed under complementation [Imm88, Sze88], LogCFL was also shown to be closed under complementation in a proof that also used the inductive counting technique [BCD$^+$89]. A similar history followed a few years later: not long after it was shown that NL is contained in $\oplus$L/poly [Wig94, GW96], the isolation lemma was again used to show that LogCFL is contained in $\oplus$SAC^1/poly [Gál95, GW96]. (As is noted in [GW96], this was independently shown by H. Venkateswaran.)

In this section, we show that the same techniques that were used in section 2 can be used to prove an analogous result about LogCFL. (In fact, it would also be possible to derive the result of section 2 from a modification of the proof of this section. Since some readers may be more interested in NL than LogCFL, we have chosen to present a direct proof of NL/poly = UL/poly.) The first step is to state the analogue to Lemma 2.1. Before we can do that, we need some definitions.

A *weighted circuit* is a semiunbounded circuit together with a *weighting function* that assigns a nonnegative integer weight to each wire connecting any two gates in the circuit.

Let C be a weighted circuit, and let g be a gate of C. A *certificate for* $g(x) = 1$ *(in* C*)* is a list of gates, corresponding to a depth-first search of the subcircuit of C rooted at g. The *weight of a certificate* is the sum of the weights of the edges traversed in the depth-first search. This informal definition is made precise by the following inductive definition. (It should be noted that this definition differs in some unimportant ways from the definition given in [Gál95, GW96].)

- If g is a constant 1 gate or an input gate evaluating to 1 on input x, then the only certificate for g is the string g. This certificate has weight 0.
- If g is an AND gate of C with inputs h_1 and h_2 (where h_1 lexicographically precedes h_2), then any string of the form gyz is a certificate for g, where y is any certificate for h_1, and z is any certificate for h_2. If w_i is the weight of the edge connecting h_i to g, then the weight of the certificate gyz is $w_1 + w_2$ plus the sum of the weights of certificates y and z.
- If g is an OR gate of C, then any string of the form gy is a certificate for g, where y is any certificate for a gate h that is an input to g in C. If w is the weight of the edge connecting h to g, then the weight of the certificate gy is w plus the weight of certificate y.

Note that if C has logarithmic depth d, then any certificate has length bounded by a polynomial in n and has weight bounded by 2^d times the maximum weight of any edge. Every gate that evaluates to 1 on input x has a certificate, and no gate that evaluates to 0 has a certificate.

We will say that a weighted circuit C *is min-unique on input* x if, for every gate g that evaluates to 1 on input x, the minimal-weight certificate for $g(x) = 1$ is unique.

LEMMA 3.2. *For any language A in LogCFL, there is a sequence of advice strings $\alpha(n)$ (having length polynomial in n) with the following properties:*

- *Each $\alpha(n)$ is a list of weighted circuits of logarithmic depth $\langle C_1, \dots, C_n\rangle$.*
- *For each input x and for each i, $x \in A$ if and only if $C_i(x) = 1$.*
- *For each input x, there is some i such that C_i is min-unique on input x.*

Lemma 3.2 is in some sense implicit in [Gál95, GW96]. We include a proof for completeness.

Proof. Let A be in LogCFL, and let C be the semiunbounded circuit of size n^l (i.e., having at most n^l gates) and depth $d = O(\log n)$ recognizing A on inputs of length n.

As in [Gál95, GW96], a modified application of the isolation lemma technique of [MVV87] shows that, for each input x, if each wire in C is assigned a weight in the range $[1, 4n^{3l}]$ uniformly and independently at random, then with probability at least $\frac{3}{4}$, C is min-unique on input x. (Sketch: The probability that there is more than one minimum weight certificate for $g(x) = 1$ is bounded by the sum, over all wires e, of the probability of the event $\text{BAD}(e, g)$ ::= "e occurs in one minimum-weight certificate for $g(x) = 1$ and not in another." Given any weight assignment w' to the edges in C other than e, there is at most one value z with the property that, if the weight of e is set to be z, then $\text{BAD}(e, g)$ occurs. Thus the probability that there are two minimum-weight certificates for any gate in C is bounded by $\sum_{g,e}\sum_{w'} \text{BAD}(e, g|w')\text{Prob}(w')$ $\le \sum_{g,e}\sum_{w'} 1/(4n^{3l})\text{Prob}(w') = \sum_{g,e} 1/(4n^{3l}) \le 1/4$.)

Now consider sequences β consisting of n weight functions $\langle w_1, \dots, w_n\rangle$, where each weight function assigns a weight in the range $[1, 4n^{3l}]$ to each edge of C. (There are $B(n) = 2^{n^{O(1)}}$ such sequences possible for each n.) There must exist a string β such that, for each input x of length n, there is some $i \le n$ such that the weighted circuit C_i that results by applying weight function w_i to C is min-unique on input x. (Sketch of proof: Let us call a sequence β "bad for x" if none of the circuits C_i in the sequence is min-unique on input x. For each x, the probability that a randomly chosen β is bad is bounded by (probability that C_i is not min-unique)^n $\le (1/4)^n = 2^{-2n}$. Thus the total number of sequences that are bad for some x is at

```
Input $(C, x, k, c_k, \Sigma_k, g)$
$count := 0$; $sum := 0$; $a := \infty$;
for each gate $h$ do
    Guess nondeterministically if $W(h) \leq k$.
    if the guess is $W(h) \leq k$, then
        begin
          Guess a certificate of size $l \leq k$ for $h$ (If this fails, then halt and reject).
          $count := count + 1$; $sum := sum + l$;
          if $h = g$, then $a := l$;
        end
endfor
if $count = c_k$ and $sum = \Sigma_k$,
    then return $a$
    else halt and reject
end.procedure
```

FIG. 4. *An unambiguous routine to calculate $W(g)$ if $W(g) \leq k$ and return ∞ otherwise.*

most $2^n(2^{-2n}B(n)) < B(n)$. Thus there is some sequence β that is not bad for *any* x.)

The desired advice sequence $\alpha(n) = \langle C_1, \ldots, C_n \rangle$ is formed by taking a good sequence $\beta = \langle w_1, \ldots, w_n \rangle$ and letting C_i be the result of applying weight function w_i to C. ∎

THEOREM 3.3. *LogCFL $\subseteq$ UAuxPDA$(\log n, n^{O(1)})$/poly.*

Proof. Let A be a language in LogCFL. Let x be a string of length n, and let $\langle C_1, \ldots, C_n \rangle$ be the advice sequence guaranteed by Lemma 3.2.

We show that there is an unambiguous auxiliary pushdown automaton M that runs in polynomial time and uses logarithmic space on its worktape that, given a sequence of circuits as input, processes each circuit in turn, and has the following properties:

- If C_i is not min-unique on input x, then M has a unique path that determines this fact and goes on to process C_{i+1}; all other paths are rejecting.
- If C_i is min-unique on input x and evaluates to 1 on input x, then M has a unique accepting path.
- If C_i is min-unique on input x but evaluates to zero on input x, then M has no accepting path.

Our construction is similar in many respects to that of section 2. Given a circuit C, let c_k denote the number of gates g that have a certificate for $g(x) = 1$ of weight at most k, and let Σ_k be the sum, over all gates g having a certificate for $g(x) = 1$ of weight at most k, of the minimum-weight certificate of g. (Let $W(g)$ denote the weight of the minimum-weight certificate of $g(x) = 1$, if such a certificate exists, and let this value be ∞ otherwise.)

A useful observation is that *if all gates of C having certificates of weight at most k have unique minimal-weight certificates* (and if the correct values of c_k and Σ_k are provided), then on input $(C, x, k, c_k, \Sigma_k, g)$, an unambiguous AuxPDA can determine if $W(g) > k$, and if $W(g) \leq k$, the AuxPDA can compute the value of $W(g)$. This is achieved with the routine shown in Figure 4.

To see that this routine truly is unambiguous if the preconditions are met, note the following:

- If the routine ever guesses incorrectly for some gate h that $W(h) > k$, then the variable *count* will never reach c_k and the routine will reject. Thus the only paths that run to completion guess correctly exactly the set $\{h \mid W(h) \leq k\}$.
- For each gate h such that $W(h) \leq k$, there is exactly one minimal-weight certificate that can be found. An UAuxPDA will find this certificate using its pushdown to execute a depth-first search (using nondeterminism at the OR gates and using its $O(\log n)$ workspace to compute the weight of the certificate), and only one path will find the minimal-weight certificate. If, for some gate h, a certificate of weight greater than $W(h)$ is guessed, then the variable *sum* will not be equal to Σ_k at the end of the routine, and the path will halt and reject.

Clearly, all gates at the input level have unique minimal-weight certificates (and the only gates g with $W(g) = 0$ are at the input level). Thus we can set $c_0 = n + 1$ (since each input bit and its negation are provided, along with the constant 1) and $\Sigma_0 = 0$. A key part of the construction involves computing c_k and Σ_k from (c_{k-1}, Σ_{k-1}), at the same time checking that no gate has two minimal-weight certificates of weight k. Consider each gate g in turn. If g is an AND gate with inputs h_1 and h_2 and weights w_1 and w_2 connecting g to these inputs, then $W(g) \leq k$ if and only if $(W(g) = l \leq k-1)$ or $((W(g) > k-1)$ and $(W(h_1) + W(h_2) + w_1 + w_2 = k))$. If g is an OR gate, then it suffices to check, for each gate h that is connected to g by an edge of weight w, if $(W(g) = l \leq k-1)$ or $((W(g) > k-1)$ and $(W(h)+w = k))$; if one such gate is found, then $W(g) = k$; if two such gates are found, then the circuit is not min-unique on input x. If no violations of this sort are found for any k, then C is min-unique on input x. The code shown in Figure 5 formalizes these considerations.

Evaluating a given circuit C_i is now expressed by the routine shown in Figure 6.

We complete the proof by describing how our algorithm processes the sequence $\langle C_1, \ldots, C_n\rangle$, as outlined at the start of the proof. Given the sequence $\langle C_1, \ldots, C_n\rangle$, the algorithm processes each C_i in turn. If C_i is not min-unique on input x, then one unique computation path of the routine returns the value *BAD.CIRCUIT* and goes on to process C_{i+1}; all other computation paths halt and reject. Otherwise, the routine has a unique accepting path if $C_i(x) = 1$, and if this is not the case the routine halts with no accepting computation paths. □

Corollary 3.4. *LogCFL/poly = UAuxPDA*$(\log n, n^{O(1)})$*/poly.*

4. Discussion and open problems.

Rytter [Ryt87] (see also [RR92]) showed that any unambiguous context-free language can be recognized in logarithmic time by a concurrent-read, exclusive-write parallel random access machine (CREW-PRAM). In contrast, no such CREW algorithm is known for any problem complete for NL, even in the nonuniform setting, although one might initially suspect that our results, combined with those of [Ryt87], would yield such algorithms, because of the following considerations:

- NL is the class of languages reducible to linear context-free languages [Sud75].
- The class of languages accepted by deterministic AuxPDAs in logarithmic space and polynomial time coincides with the class of languages logspace-reducible to deterministic context-free languages.
- LogCFL coincides with AuxPDA$(\log n, n^{O(1)})$.

```
Input $(C, x, k, c_{k-1}, \Sigma_{k-1})$
Output $(c_k, \Sigma_k)$, and also the flag $BAD.CIRCUIT$

$c_k := c_{k-1}; \Sigma_k := \Sigma_{k-1}$;
for each gate $g$ do
   if $W(g) > k-1$, then
      begin
      if $g$ is an AND gate with inputs $h_1, h_2$, connected
          to $g$ with edges weighted $w_1, w_2$ and
          $W(h_1) + W(h_2) + w_1 + w_2 = k$, then
        $c_k := c_k + 1$; $\Sigma_k := \Sigma_k + k$
      if $g$ is an OR gate, then
          for each $h$ connected to $g$ by an edge weighted $w$ do
              if $W(h) = k - w$, then
                  begin
                  $c_k := c_k + 1$; $\Sigma_k := \Sigma_k + k$
                  for $h' \neq h$ connected to $g$ by an edge of weight $w'$ do
                      if $W(h') = k - w'$,
                          then $BAD.CIRCUIT$ := true:
                  endfor
                  end
          endfor
      end
endfor
At this point, if $BAD.CIRCUIT$ = false, the values of $c_k$ and $\Sigma_k$ are correct.
```

FIG. 5. *Computing c_k and Σ_k.*

```
Input $(C_i)$
$BAD.CIRCUIT$ := false; $c_0 := n + 1; \Sigma_0 := 0$;
for $k = 1$ to $2^d 4n^{3l}$
     compute $(c_k, \Sigma_k)$ from $c_{k-1}, \Sigma_{k-1}$;
     if $BAD.CIRCUIT$ = true, then exit the for loop.
endfor
If $BAD.CIRCUIT$ = false, then the output gate $g$ evaluates to 1 if and only if $W(g) < \infty$.
```

FIG. 6. *Evaluating a circuit.*

That is, there is a close connection between deterministic and nondeterministic context-free languages, and related deterministic and nondeterministic complexity classes. Shouldn't similar relationships hold for the unambiguous classes? Unfortunately, it is *not* known that UAuxPDA$(\log n, n^{O(1)})$ or UL is reducible to unambiguous context-free languages. The work of Niedermeier and Rossmanith does an excellent job of explaining the subtleties and difficulties here [NR95]. CREW algorithms are closely associated with a version of unambiguity called *strong unambiguity.* In terms of Turing-machine based computation, strong unambiguity means that, not only is there at most one path from the start vertex to the accepting configuration, but in fact there is at most one path between *any two configurations of the machine.*

Strongly unambiguous classes have more efficient algorithms than are known for general NL or UL problems. It is shown in [AL98] that problems in strongly unambiguous logspace have deterministic algorithms using less than $\log^2 n$ space, and it is shown in [BJLR91] that this class is also in LogDCFL (and hence has logarithmic-time concurrent-read, owner-write parallel random access machine (CROW-PRAM) algorithms and is in SC^2). For more information on this connection to CROW-PRAM algorithms, see [FLR96].

The reader is encouraged to note that, in a min-unique graph, the shortest path between *any two vertices* is unique. This bears a superficial resemblance to the property of strong unambiguity. We see no application of this observation.

It is natural to ask if the randomized aspect of the construction can be eliminated using some sort of derandomization technique to obtain the equality UL = NL. In more recent work [ARZ], we observe that if DSPACE(n) contains a language with sufficiently high circuit complexity, then the techniques of [NW94] can be used to build pseudorandom generators of sufficiently high quality, so that the results of this paper would also hold in the uniform setting.

A corollary of our work is that UL/poly is closed under complement. It remains an open question if UL is closed under complement, although some of the unambiguous logspace classes that can be defined using strong unambiguity are known to be closed under complement [BJLR91]. Similarly, UL/poly has a complete set under the natural types of reducibility to consider (nonuniform logspace reductions or even nonuniform projections). In contrast, UL itself is not known to have any complete sets under logspace reducibility. In this regard, note that Lange has shown that one of the other unambiguous logspace classes does have complete sets [Lan97].

It is disappointing that the techniques used in this paper do not seem to provide any new information about complexity classes such as NSPACE(n) and NSPACE(2^n). It is straightforward to show that NSPACE($s(n)$) is contained in the advice class USPACE($s(n)$)/$2^{O(s(n))}$, but this is interesting only for sublinear $s(n)$. (In a personal communication, Fortnow [FOR] has pointed out that our argument does show that NSPACE(n) = USPACE(n) relative to a random oracle.)

There is a natural class of functions associated with NL, denoted FNL [AJ93]. This can be defined in several equivalent ways, such as

- the class of functions computable by NC^1 circuits with oracle gates for problems in NL,
- the class of functions f such that $\{(x, i, b) \mid \text{the } i\text{th bit of } f(x) \text{ is } b\}$ is in NL,
- the class of functions computable by logspace-bounded machines with oracles for NL.

Another important class of problems related to NL is the class #L, which counts the number of accepting paths of an NL machine. #L characterizes the complexity of computing the determinant [Vin91]. (See also [Tod, Dam, MV97, Val92, AO96].) It was observed in [AJ93] that if NL = UL, then FNL is contained in #L. Thus a corollary of the result in this paper is that FNL/poly $\subseteq$ #L/poly.

Many questions about #L remain unanswered. Two interesting complexity classes related to #L are PL (probabilistic logspace) and $\mathrm{C_=L}$ (which characterizes the complexity of singular matrices, as well as questions about computing the rank). It is known that some natural hierarchies defined using these complexity classes collapse:

- $\mathrm{AC}^0(\mathrm{C_=L}) = \mathrm{C_=L}^{\mathrm{C_=L}^{\cdot^{\cdot^{\mathrm{C_=L}}}}} = \mathrm{NC}^1(\mathrm{C_=L}) = \mathrm{L}^{\mathrm{C_=L}}$ [AO96, ABO96].
- $\mathrm{AC}^0(\mathrm{PL}) = \mathrm{PL}^{\mathrm{PL}^{\cdot^{\cdot^{\mathrm{PL}}}}} = \mathrm{NC}^1(\mathrm{PL}) = \mathrm{PL}$ [AO96, Ogi98, BF97].

In contrast, the corresponding #L hierarchy (equal to the class of problems AC^0 reducible to computing the determinant) $\mathrm{AC}^0(\#\mathrm{L}) = \mathrm{FL}^{\#\mathrm{L}^{\cdot^{\cdot^{\#\mathrm{L}}}}}$ is not known to collapse to any fixed level. Does the equality UL/poly = NL/poly provide any help in analyzing this hierarchy in the nonuniform setting?

It is instructive to view our results in terms of arithmetic circuits. An equivalent definition of the class of functions #L results by taking the Boolean circuit characterization of NL (see [Ven92]) and replacing each Boolean AND and OR gate by integer multiplication and addition, respectively. The class $\#\mathrm{SAC}^1$ can be defined similarly. This notion of arithmetic circuit complexity has been investigated in a series of papers including [Vin91, CMTV96, AAD97, All97]. Our results say that the zero-one valued characteristic function of any language in NL (or LogCFL) can be computed by the corresponding (nonuniform) class of arithmetic circuits. Note that, although the output gate is producing a value in {0,1}, some of the interior gates will be producing larger values. Are there equivalent arithmetic circuits where *all* gates take values in {0,1}? (This is essentially the notion of strong unambiguity.) Note that each such gate is itself defining a language in NL (or LogCFL), and thus there is a zero-one valued arithmetic circuit for it—but this circuit may itself have gates that produce large values.

Acknowledgments. We thank Klaus-Jörn Lange for helpful comments, for drawing our attention to min-unique graphs, and for arranging for the second author to spend some of his sabbatical time in Tübingen. We also thank V. Vinay and Lance Fortnow for insightful comments.

REFERENCES

[AAD97] M. Agrawal, E. Allender, and S. Datta, *On TC^0, AC^0, and arithmetic circuits*, in Proceedings 12th Annual IEEE Conference on Computational Complexity, 1997, pp. 134–148.

[ABO96] E. Allender, R. Beals, and M. Ogihara, *The complexity of matrix rank and feasible systems of linear equations*, in ACM Symposium on Theory of Computing (STOC), 1996.

[AJ93] C. Álvarez and B. Jenner, *A very hard log-space counting class*, Theoret. Comput. Sci., 107 (1993), pp. 3–30.

[AL98] E. Allender and K.-J. Lange, *RUSPACE*$(\log n)$ *is contained in DSPACE*$(\log^2 n/\log\log n)$, Theory Comput. Syst., 31 (1998), pp. 539–550.

[All97] E. Allender, *Making computation count: Arithmetic circuits in the nineties*, SIGACT News, 28 (1997), pp. 2–15.

[AO96] E. Allender and M. Ogihara, *Relationships among PL, #L, and the determinant*, RAIRO Theoret. Informat. Appl., 30 (1996), pp. 1–21.

[AR98] E. Allender and K. Reinhardt, in Proceedings 13th Annual IEEE Conference on Computational Complexity, 1998, pp. 92–100.

[ARZ] E. Allender, K. Reinhardt, and S. Zhou, *Isolation, matching, and counting: Uniform and nonuniform upper bounds*, J. Comput. System Sci.; a preliminary version appeared as [AR98], to appear.

[BCD$^+$89] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa, *Two applications of inductive counting for complementation problems*, SIAM J. Comput., 18 (1989), pp. 559–578.

[BF97] R. Beigel and B. Fu, *Circuits over PP and PL*, in IEEE Conference on Computational Complexity, 1997, pp. 24–35.

[BJLR91] G. Buntrock, B. Jenner, K.-J. Lange, and P. Rossmanith, *Unambiguity and fewness for logarithmic space*, in Proceedings 8th International Conference on Fundamentals of Computation Theory (FCT '91), Lecture Notes in Comput. Sci. 529, Springer-Verlag, New York, NY, 1991, pp. 168–179.

[CMTV96] H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer, *Nondeterministic NC^1 computation*, in Proceedings 11th Annual IEEE Conference on Computational Complexity, pp. 12–21.

[Dam] C. Damm, *$DET = L^{\#1}$?*, Informatik-Prepri//nt 8, Fachbereich Informatik der Humboldt-Universität zu Berlin, 1991.

[FLR96] H. Fernau, K.-J. Lange, and K. Reinhardt, *Advocating ownership*, in Proceedings 16th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 1180, V. Chandru, ed., Springer-Verlag, New York, NY, 1996, pp. 286–297.

[FOR] L. Fortnow, Private communication, University of Chicago, Chicago, IL, 1997.

[Gál95] A. Gál, *Semi-unbounded fan-in circuits: Boolean vs. arithmetic*, in IEEE Structure in Complexity Theory Conference, IEEE, Piscataway, NJ, 1995, pp. 82–87.

[GS88] J. Grollmann and A. Selman, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–335.

[GW96] A. Gál and A. Wigderson, *Boolean vs. arithmetic complexity classes: Randomized reductions*, Random Structures Algorithms, 9 (1996), pp. 99–111.

[Imm88] N. Immerman, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17 (1988), pp. 935–938.

[Jon75] N. D. Jones, *Space bounded reducibility among combinatorial problems*, J. Comput. System Sci., 11 (1975), pp. 68–85.

[KL82] R. Karp and R. Lipton, *Turing machines that take advice*, Enseign. Math., 28 (1982), pp. 191–209.

[Lan97] K.-J. Lange, *An unambiguous class possessing a complete set*, in Proceedings 14th Symposium on Theoretical Aspects of Computer Science (STACS '97), Lecture Notes in Comput. Sci. 1200, Springer-Verlag, New York, 1997, pp. 339–350.

[MV97] M. Mahajan and V. Vinay, *Determinant: Combinatorics, algorithms, and complexity*, Chicago J. Theoret. Comput. Sci., 1997, available online at http://cs-www.uchicago.edu/publications/cjtcs.

[MVV87] K. Mulmuley, U. Vazirani, and V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[NR95] R. Niedermeier and P. Rossmanith, *Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits*, Inform. and Comput., 118 (1995), pp. 227–245.

[NW94] N. Nisan and A. Wigderson, *Hardness vs. randomness*, J. Comput. System Sci., 49 (1994), pp. 149–167.

[Ogi98] M. Ogihara, *The PL hierarchy collapses*, SIAM J. Comput., 27 (1998), pp. 1430–1437.

[Raz90] A. Razborov, *Lower bounds on the size of switching-and-rectifier networks for symmetric Boolean functions*, Mathematical Notes of the Academy of Sciences of the USSR, 48 (1990), pp. 79–91.

[Raz92] A. Razborov, *Lower bounds for deterministic and nondeterministic branching programs*, in Proceedings 8th International Conference on Fundamentals of Computation Theory (FCT '91), Lecture Notes in Comput. Sci. 529, Springer-Verlag, New York, 1992, pp. 47–60.

[Reg97] K. Regan, *Polynomials and combinatorial definitions of languages*, in Complexity Theory Retrospective II, L. Hemaspaandra and A. Selman, eds., Springer-Verlag, New York, 1997, pp. 261–293.

[RR92] P. Rossmanith and W. Rytter, *Observations on $\log n$ time parallel recognition of unambiguous context-free languages*, Inform. Process. Lett., 44 (1992), pp. 267–272.

[Ryt87] W. Rytter, *Parallel time $O(\log n)$ recognition of unambiguous context-free languages*, Inform. and Comput., 73 (1987), pp. 75–86.

[Sud75] I. H. Sudborough, *A note on tape-bounded complexity classes and linear context-free languages*, J. ACM, 22 (1975), pp. 499–500.

[Sud78] I. H. Sudborough, *On the tape complexity of deterministic context-free languages*, J. ACM, 25 (1978), pp. 405–414.

[Sze88] R. Szelepcsényi, *The method of forced enumeration for nondeterministic automata*, Acta Inform., 26 (1988), pp. 279–284.

[Tod] S. Toda, *Counting Problems Computationally Equivalent to the Determinant*, Technical report CSIM 91-07, Department of Computer Science and Information Mathematics, University of Electro-Communications, Tokyo, 1991.

[Val76] L. Valiant, *The relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.

[Val92] L. Valiant, *Why is Boolean complexity theory difficult?*, in Boolean Function Complexity, M. Paterson, ed., London Math. Soc. Lecture Notes Ser. 169, Cambridge University Press, Cambridge, 1992, pp. 84–94.

[Ven91] H. Venkateswaran, *Properties that characterize LOGCFL*, J. Comput. System Sci., 43 (1991), pp. 380–404.

[Ven92] H. Venkateswaran, *Circuit definitions of nondeterministic complexity classes*, SIAM J. Comput., 21 (1992), pp. 655–670.

[Vin91] V. Vinay, *Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits*, in Proceedings of the 6th Structure in Complexity Theory Conference, IEEE, Piscataway, NJ, 1991, pp. 270–284.

[VV86] L. Valiant and V. Vazirani, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.

[Wig94] A. Wigderson, *NL/poly* $\subseteq \oplus$ *L/poly*, in Proceedings of the 9th IEEE Structure in Complexity Conference, IEEE, Piscataway, NJ, 1994, pp. 59–62.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1132–1154

A COMBINATORIAL CONSISTENCY LEMMA WITH APPLICATION TO PROVING THE PCP THEOREM[*]

ODED GOLDREICH[†] AND SHMUEL SAFRA[‡]

Abstract. The current proof of the probabilistically checkable proofs (PCP) theorem (i.e., $\mathcal{NP} = \mathcal{PCP}(\log, O(1))$) is very complicated. One source of difficulty is the technically involved analysis of low-degree tests. Here, we refer to the difficulty of obtaining *strong* results regarding low-degree tests; namely, results of the type obtained and used by Arora and Safra [*J. ACM*, 45 (1998), pp. 70–122] and Arora et al. [*J. ACM*, 45 (1998), pp. 501–555].

In this paper, we eliminate the need to obtain such strong results on low-degree tests when proving the PCP theorem. Although we do not remove the need for low-degree tests altogether, using our results it is now possible to prove the PCP theorem using a simpler analysis of low-degree tests (which yields weaker bounds). In other words, we replace the strong algebraic analysis of low-degree tests presented by Arora and Safra and Arora et al. by a combinatorial lemma (which does not refer to low-degree tests or polynomials).

Key words. parallelization of probabilistic proof systems, probabilistically checkable proofs (PCP), NP, low-degree tests

AMS subject classifications. 68Q15

PII. S0097539797315744

1. Introduction.

The characterization of $\mathcal{NP}$ in terms of probabilistically checkable proofs (PCP systems) [AS, ALMSS], hereafter referred to as the PCP characterization theorem, is one of the more fundamental achievements of complexity theory. Loosely speaking, this theorem states that membership in any NP-language can be verified probabilistically by a polynomial-time machine which inspects a constant number of bits (in random locations) in a "redundant" NP-witness. Unfortunately, the current proof of the PCP characterization theorem is very complicated and, consequently, has not been fully assimilated into complexity theory. Clearly, changing this state of affairs is highly desirable.

There are two aspects of the current proof (of the PCP characterization theorem) which are difficult. One difficult aspect is the complicated conceptual structure of the proof (most notably the acclaimed "proof composition" paradigm). Yet, with time, this part seems easier to understand and explain than when it was first introduced. Furthermore, the proof composition paradigm turned out to be very useful and played a central role in subsequent works in this area (cf. [BGLR, BS, BGS, H96]). The other difficult aspect is the technically involved analysis of low-degree tests. Here we refer to the difficulty of obtaining *strong* results regarding low-degree tests, namely, results of the type obtained and used in [AS] and [ALMSS].

In this paper, we eliminate the latter difficulty. Although we do not get rid of low-degree tests altogether, using our results it is now possible to prove the PCP characterization theorem using only the weaker and simpler analysis of low-degree

[*]Received by the editors January 27, 1997; accepted for publication (in revised form) February 4, 1999; published electronically February 1, 2000.

http://www.siam.org/journals/sicomp/29-4/31574.html

[†]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel (oded@wisdom.weizmann.ac.il). The research of this author was supported in part by grant 92-00226 from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel.

[‡]Computer Science Department, Sackler Faculty of Exact Sciences, Tel-Aviv University, Ramat-Aviv, Israel (safra@math.tau.ac.il).

tests presented in [GLRSW, RS92, RS96]. In other words, we replace the complicated algebraic analysis of low-degree tests presented in [AS, ALMSS] by a combinatorial lemma (which does not refer to low-degree tests or even to polynomials). We believe that this combinatorial lemma is very intuitive and find its proof much simpler than the algebraic analysis of [AS, ALMSS]. (However, simplicity may be a matter of taste.)

Loosely speaking, our combinatorial lemma provides a method of generating sequences of pairwise independent random points so that any assignment of values to the sequences either induces essentially consistent values on the individual elements or is detected as inconsistent. This is achieved by a "consistency test" which samples a constant number of sequences (and obtains the values assigned to these sequences). We stress that the length of the sequences as well as the domain from which the elements are chosen are parameters, which may grow while the number of samples remains fixed.

1.1. Two combinatorial consistency lemmas. The following problem arises frequently when trying to design PCP systems, and in particular when proving the PCP characterization theorem. For some sets S and V, one has a procedure which, given (bounded) oracle access to any function $f : S \mapsto V$, tests whether f has some desired property. The procedure should always accept a function having the property and should reject with "noticeable" probability any function which is far from having the property (i.e., differs from any function having the property on a significant fraction of the domain). For example, the property may be that of being a proof-oracle in a basic PCP system which we want to utilize (as an ingredient in the composition of PCP systems). Our goal is to increase the detection probability (equivalently, to reduce the error probability) without increasing the number of queries, but rather allowing more informative queries. For example, we are willing to allow queries in which one supplies a sequence of elements in S and expects to obtain the corresponding sequence of values of f applied to these elements. The problem is that the sequences of values obtained may not be consistent with any function $f : S \mapsto V$.

We can now phrase a simple problem of testing consistency. One is given access to a function $F : S^\ell \mapsto V^\ell$ and is asked whether there exists a function $f : S \mapsto V$ so that for most sequences $(x_1, \ldots, x_\ell) \in S^\ell$,

$$F(x_1, \ldots, x_\ell) = (f(x_1), \ldots, f(x_\ell)).$$

Loosely speaking, we prove that querying F on a constant number of related random sequences suffices for testing a relaxation of the above.

LEMMA 1.1 (combinatorial consistency—simple case). *For every $\delta > 0$, there exist a constant $c = \mathrm{poly}(1/\delta)$ and a probabilistic oracle machine, T, which on input $(\ell, |S|)$ runs for $\mathrm{poly}(\ell \cdot \log |S|)$-time and makes at most c queries to an oracle $F : S^\ell \mapsto V^\ell$, such that*

- *If there exists a function $f : S \mapsto V$ such that $F(x_1, \ldots, x_\ell) = (f(x_1), \ldots, f(x_\ell))$, for all $(x_1, \ldots, x_\ell) \in S^\ell$, then T always accepts when given access to oracle F.*
- *If T accepts with probability at least $\frac{1}{2}$, when given access to oracle F, then there exists a function $f : S \mapsto V$ such that the sequences $F(x_1, \ldots, x_\ell)$ and $(f(x_1), \ldots, f(x_\ell))$ agree on at least $\ell - \sqrt{\ell}$ positions, for at least a $1-\delta$ fraction of all possible $(x_1, \ldots, x_\ell) \in S^\ell$.*

Specifically, the test examines the value of the function F on random pairs of sequences $((r_1, \ldots, r_\ell), (s_1, \ldots, s_\ell))$, where $r_i = s_i$ for $\sqrt{\ell}$ of the i's, and checks that the corresponding values (on these r_i's and s_i's) are indeed equal. For details, see section 4.

Unfortunately, this relatively simple consistency lemma does not suffice for the PCP applications. The reason being that, in that application, error reduction (see above) is done via randomness-efficient procedures such as pairwise-independent sequences (since we cannot afford to utilize $\ell \cdot \log_2 |S|$ random bits as above). Consequently, the function F is not defined on the entire set S^ℓ but rather on a very sparse subset, denoted $\mathbf{S}$. Thus, one is given access to a function $F : \mathbf{S} \mapsto V^\ell$ and is asked whether there exists a function $f : S \mapsto V$ so that for most sequences $(x_1, \ldots, x_\ell) \in \mathbf{S}$, the sequences $F(x_1, \ldots, x_\ell)$ and $(f(x_1), \ldots, f(x_\ell))$ agree on most (contiguous) subsequences of length $\sqrt{\ell}$. The main result of this paper is the following lemma.

Lemma 1.2 (combinatorial consistency—sparse case). *For every two of integers $s, \ell > 1$, there exists a set $\mathbf{S}_{s,\ell} \subset [s]^\ell$, where $[s] \overset{\text{def}}{=} \{1, \ldots, s\}$, so that the following holds:*

1. *For every $\delta > 0$, there exist a constant $c = \text{poly}(1/\delta)$ and a probabilistic oracle machine, T, which on input (ℓ, s) runs for $\text{poly}(\ell \cdot \log s)$-time and makes at most c queries to an oracle $F : \mathbf{S}_{s,\ell} \mapsto V^\ell$, such that*
   - *if there exists a function $f : [s] \mapsto V$ such that $F(x_1, \ldots, x_\ell) = (f(x_1), \ldots, f(x_\ell))$, for all $(x_1, \ldots, x_\ell) \in \mathbf{S}_{s,\ell}$, then T always accepts when given access to oracle F.*
   - *if T accepts with probability at least $\frac{1}{2}$, when given access to oracle F, then there exists a function $f : [s] \mapsto V$ such that for at least a $1 - \delta$ fraction of all possible $(x_1, \ldots, x_\ell) \in \mathbf{S}_{s,\ell}$ the sequences $F(x_1, \ldots, x_\ell)$ and $(f(x_1), \ldots, f(x_\ell))$ agree on at least a $1 - \delta$ fraction of the (contiguous) subsequences of length $\sqrt{\ell}$.*
2. *The individual elements in a uniformly selected sequence in $\mathbf{S}_{s,\ell}$ are uniformly distributed in $[s]$ and are pairwise-independent. Furthermore, the set $\mathbf{S}_{s,\ell}$ has cardinality $\text{poly}(s)$ and can be constructed in $\text{poly}(s, \ell)$-time.*

Specifically, the test examines the value of the function F on related random pairs of sequences $((r_1, \ldots, r_\ell), (s_1, \ldots, s_\ell)) \in \mathbf{S}_{s,\ell}$. These sequences are viewed as $\sqrt{\ell} \times \sqrt{\ell}$ matrices, and, loosely speaking, they are chosen to be random extensions of the same random row (or column). For details, see section 2.

In particular, the presentation in section 2 axiomatizes properties of the set of sequences, $\mathbf{S}_{s,\ell}$, for which the above tester works. Thus, we provide a "parallel repetition theorem" which holds for random but nonindependent instances (rather than for independent random instances as in other such results). However, our parallel repetition theorem applies only to the case where a single query is asked in the basic system (rather than a pair of related queries as in other results). Due to this limitation, we could not apply our parallel repetition theorem directly to the error-reduction of generic proof systems. Instead, as explained below, we applied our parallel repetition theorem to derive a relatively strong low-degree test from a weaker low-degree test.

We believe that the combinatorial consistency lemma of section 2 may play a role in subsequent developments in the area.

1.2. Application to the PCP characterization theorem. The currently known proof of the PCP characterization theorem [ALMSS] composes proof systems in which the verifier makes a constant number of multivalued queries. Such verifiers are constructed by "parallelization" of simpler verifiers, and thus the problem of "consistency" arises. This problem is solved by use of low-degree multivariant polynomials, which in turn requires "high-quality" low-degree testers. Specifically, given a function

$f : \mathrm{GF}(p)^n \mapsto \mathrm{GF}(p)$, where p is prime, one needs to test whether f is close to some low-degree polynomial (in n variables over the finite field $\mathrm{GF}(p)$). It is required that any function f which disagrees with every d-degree polynomial on at least, say, 1% of the inputs be rejected with, say, a probability of 99%. The test is allowed to use auxiliary proof oracles (in addition to f), but it may only make a *constant* number of queries and the answers must have length bounded by $\mathrm{poly}(n, d, \log p)$. Using a technical lemma due to Arora and Safra [AS], Arora et al. [ALMSS] proved such a result.[1] The full proof is quite complex and is algebraic in nature. A weaker result due to Gemmel et al. [GLRSW] (see [RS96]) asserts the existence of a d-degree test which, using $d+2$ queries, rejects such bad functions with probability at least $\Omega(1/d^2)$. Their proof is much simpler. Combining the result of Gemmel et al. [GLRSW, RS96] with our combinatorial consistency lemma (i.e., Lemma 1.2), we obtain an alternative proof of the following result.

LEMMA 1.3 (low-degree tester). *For every $\delta > 0$, there exist a constant c and a probabilistic oracle machine, T, which on input n, p, d runs for $\mathrm{poly}(n, d, \log p)$-time and makes at most c queries to both f and to an auxiliary oracle F, such that*

- *if f is a degree-d polynomial, then there exists a function F so that T always accepts.*
- *if T accepts with probability at least $\frac{1}{2}$, when given access to the oracles f and F, then f agrees with some degree-d polynomial on at least a $1 - O(1/d^2)$ fraction of the domain.*[2]

Furthermore, the test uses $O(n \log p)$ coin tosses, and makes queries of length $O(n \log p)$.

We stress that in contrast to [ALMSS] our proof of the above lemma is mainly combinatorial. Our only reference to algebra is in relying on the result of Gemmel et al. [GLRSW, RS96] (which is weaker and has a simpler proof than that of [ALMSS]). Our tester works by performing many (pairwise-independent) instances of the [GLRSW] test in parallel and by guaranteeing the consistency of the answers obtained in these tests via our combinatorial consistency test (i.e., of Lemma 1.2). In contrast, prior to our work the only way to guarantee the consistency of these answers resulted in the need to perform a low-degree test of the type asserted in Lemma 1.3 (and using [ALMSS], which was the only alternative known; this meant losing the advantage of utilizing a low-degree test with a simpler algebraic analysis).

1.3. Related work. We refrain from an attempt to provide an account of the developments which have culminated in the PCP characterization theorem. Works which should certainly be mentioned include [GMR, BGKW, FRS, LFKN, Sha, BFL, BFLS, FGLSS, AS, ALMSS] as well as [BF, BLR, LS, RS92]. For detailed accounts, see surveys by Babai [B94] and Goldreich [G97].

This paper reports work completed in the Spring of 1994 and announced at the *Weizmann Institute Workshop on Randomness and Computation* (January 1995). Hastad's recent work [H96] contains a combinatorial consistency lemma which is related to our Lemma 1.1 (i.e., the "simple case" lemma). However, Hastad's lemma (which is harder to establish) refers to the case where the test accepts with very low probability (i.e., a weaker hypothesis) and guarantees the existence of a small set of "piecewise-consistent" assignments (i.e., a weaker conclusion). Raz and Safra [RaSa] claim to have been inspired by our Lemma 1.2 (i.e., the "sparse case" lemma).

[1] An improved analysis was later obtained by Friedl and Sudan [FS].

[2] Actually, [ALMSS] only prove agreement on an (arbitrarily large) constant fraction of the domain.

1.4. Organization. The (basic) sparse case consistency lemma is presented in section 2. The application to the PCP characterization theorem is presented in section 3. Section 4 contains a proof of Lemma 1.1 (which refers to sequences of totally independent random points).

2. The consistency lemma (for the sparse case).

In this section we present our main result—a combinatorial consistency lemma which refers to sequences of bounded independence. Specifically, we considered k^2-long sequences viewed as k-by-k matrices. To emphasize the combinatorial nature of our lemma and its proof, we adopt an abstract presentation in which the properties required from the set of matrices are explicitly stated (as axioms). We comment that the set of all k-by-k matrices over S satisfies these axioms. A more important case is given in Construction 2.2— it is based on a standard construction of pairwise-independent sequences (i.e., the matrix is a pairwise-independent sequence of rows, where each row is a pairwise-independent sequence of elements).

General notation. For a positive integer k, let $[k] \stackrel{\text{def}}{=} \{1, \ldots, k\}$. For a finite set A, the notation $a \in_{\mathrm{R}} A$ means that a is uniformly selected in A. In case A is a multiset, each element is selected with probability proportional to its multiplicity.

2.1. The setting. Let S be some finite set, and let k be an integer. Though both S and k are parameters, they will be implicit in all subsequent notations.

Rows and columns. Let $\mathbf{R}$ be a multiset of sequences of length k over S so that every $e \in S$ appears in some sequence of $\mathbf{R}$. For the sake of simplicity, think of $\mathbf{R}$ as being a set (i.e., each sequence appears with multiplicity 1). Similarly, let $\mathbf{C}$ be another set of sequences (of length k over S). We neither assume $\mathbf{R} = \mathbf{C}$ nor $\mathbf{R} \neq \mathbf{C}$. We consider matrices having rows in $\mathbf{R}$ and columns in $\mathbf{C}$ (thus, we call the members of $\mathbf{R}$ row-sequences, and those in $\mathbf{C}$ column-sequences). We denote by $\mathbf{M}$ a multiset of k-by-k matrices with rows in $\mathbf{R}$ and columns in $\mathbf{C}$.

AXIOM 1. *For every $m \in \mathbf{M}$ and $i \in [k]$, the* ith *row of m is an element of $\mathbf{R}$ and the* ith *column of m is an element of $\mathbf{C}$.*

For every $i \in [k]$ and $\bar{r} \in \mathbf{R}$, we denote by $\mathbf{M}_i(\bar{r})$ the set of matrices (in $\mathbf{M}$) having $\bar{r}$ as the ith row. Similarly, for $j \in [k]$ and $\bar{c} \in \mathbf{C}$, we denote by $\mathbf{M}^j(\bar{c})$ the set of matrices (in $\mathbf{M}$) having $\bar{c}$ as the jth column. For every $\bar{r} = (r_1, \ldots, r_k) \in \mathbf{R}$ and every $\bar{c} = (c_1, \ldots, c_k) \in \mathbf{C}$, so that $r_j = c_i$, we denote by $\mathbf{M}_i^j(\bar{r}, \bar{c})$ the set of matrices having $\bar{r}$ as the ith row and $\bar{c}$ as the jth column (i.e., $\mathbf{M}_i^j(\bar{r}, \bar{c}) = \mathbf{M}_i(\bar{r}) \cap \mathbf{M}^j(\bar{c})$).

Shifts. We assume that $\mathbf{R}$ is "closed" under the shift operator.

AXIOM 2. *For every $\bar{r} = (r_1, \ldots, r_k) \in \mathbf{R}$ there exists a unique $\bar{s} = (s_1, \ldots, s_k) \in \mathbf{R}$ satisfying $s_i = r_{i-1}$, for every $2 \leq i \leq k$. We denote this right-shifted sequence by $\sigma(\bar{r})$. Similarly, we assume that there exists a unique $\bar{s} = (s_1, \ldots, s_k) \in \mathbf{R}$ satisfying $s_i = r_{i+1}$, for every $1 \leq i \leq k-1$. We denote this left-shifted sequence by $\sigma^{-1}(\bar{r})$. Furthermore,*[3] *we assume that shifting each of the rows of a matrix $m \in \mathbf{M}$ to the same direction yields a matrix m' that is also in $\mathbf{M}$.*

Axiom 2 implies that if $\bar{r}$ is uniformly distributed in $\mathbf{R}$, then so is $\sigma(\bar{r})$ (resp., $\sigma^{-1}(\bar{r})$). For every (nonnegative) integer i, the notations $\sigma^i(\bar{r})$ and $\sigma^{-i}(\bar{r})$ are defined in the natural way (e.g., $\sigma^i(\bar{r}) = \sigma^{i-1}(\sigma(\bar{r}))$ and $\sigma^0(\bar{r}) = \bar{r}$). Note that we do not assume that $\mathbf{C}$ is closed under shifts (in an analogous manner).

[3] The extra axiom is not really necessary; see the remark following the definition of the consistency test.

Distribution. We now turn to axioms concerning the distribution of rows and columns in a uniformly chosen matrix. We assume that the rows (and columns) of a uniformly chosen matrix are uniformly distributed in $\mathbf{R}$ (and $\mathbf{C}$, respectively).[4] In addition, we assume that the rows (but not necessarily the columns) are also pairwise-independent.

AXIOM 3. *Let m be uniformly selected in $\mathbf{M}$. Then,*

1. *For every $i \in [k]$, the* ith *column of m is uniformly distributed in $\mathbf{C}$.*
2. *For every $i \in [k]$, the* ith *row of m is uniformly distributed in $\mathbf{R}$.*
3. *Furthermore, for every $j \neq i$ and $\bar{r} \in \mathbf{R}$, conditioned that the* ith *row of m equals $\bar{r}$, the* jth *row of m is uniformly distributed over $\mathbf{R}$.*

Finally, we assume that the columns in a uniformly chosen matrix containing a specific row-sequence are distributed identically to uniformly selected columns with the corresponding entry.

AXIOM 4. *For every $i, j \in [k]$ and $\bar{r} = (r_1, \ldots, r_k) \in \mathbf{R}$, the* jth *column in a matrix that is uniformly selected among those having $\bar{r}$ as its* ith *row* (*i.e., $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$*) *is uniformly distributed among the column-sequences that have r_j as their* ith *element.*

Clearly, if the jth element of $\bar{r} = (r_1, \ldots, r_k)$ differs from the ith element of $\bar{c} = (c_1, \ldots, c_k)$, then $\mathbf{M}_i^j(\bar{r}, \bar{c})$ is empty. Otherwise (i.e., $r_j = c_i$), by the above axiom, $\mathbf{M}_i^j(\bar{r}, \bar{c})$ is not empty. Furthermore, the above axiom implies that (in case $r_j = c_i$) for a uniformly chosen $m \in \mathbf{M}$

$$\begin{aligned}\mathrm{Prob}(m \in \mathbf{M}_i^j(\bar{r}, \bar{c})) &= \mathrm{Prob}(m \in \mathbf{M}_i(\bar{r})) \cdot \mathrm{Prob}(m \in \mathbf{M}^j(\bar{c}) \mid m \in \mathbf{M}_i(\bar{r})) \\ &= \frac{1}{|\mathbf{R}|} \cdot \frac{1}{|C_i(r_j)|} > 0,\end{aligned}$$

where $C_i(e)$ denotes the set of column-sequences having e as their ith element, and the second equality is obtained by Axiom 4.

2.2. The test. Let Γ be a function assigning matrices in $\mathbf{M}$ (which may be a proper subset of all possible k-by-k matrices over S) values which are k-by-k matrices over some set of values V (i.e., $\Gamma : \mathbf{M} \mapsto V^{k \times k}$). The function Γ is *supposed* to be "consistent" (i.e., assign each element, e, of S the same value, independently of the matrix in which e appears). The purpose of the following test is to check that this property holds in some approximate sense.

CONSTRUCTION 2.1 (consistency test).

1. Column test. *Select a column-sequence $\bar{c}$ uniformly in $\mathbf{C}$, and $i, j \in_{\mathrm{R}} [k]$. Select two random extensions of this column, namely, $m_1 \in_{\mathrm{R}} \mathbf{M}^i(\bar{c})$ and $m_2 \in_{\mathrm{R}} \mathbf{M}^j(\bar{c})$, and test if the* ith *column of $\Gamma(m_1)$ equals the* jth *column of $\Gamma(m_2)$.*
2. Row test (analogous to the column test). *Select a row-sequence $\bar{r}$ uniformly in $\mathbf{R}$, and $i, j \in_{\mathrm{R}} [k]$. Select two random extensions of this row, namely, $m_1 \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$ and $m_2 \in_{\mathrm{R}} \mathbf{M}_j(\bar{r})$, and test if the* ith *row of $\Gamma(m_1)$ equals the* jth *row of $\Gamma(m_2)$.*
3. Shift test. *Select a matrix m uniformly in $\mathbf{M}$ and an integer $t \in [k-1]$. Let m' be the matrix obtained from m by shifting each row by t; namely, the* ith *row of m' is $\sigma^t(\bar{r})$, where $\bar{r}$ denotes the* ith *row of m. We test if the $k - t$ first columns of $\Gamma(m)$ match the $k - t$ last columns of $\Gamma(m')$.*

[4] This, in fact, implies Axiom 1.

The test accepts *if all three* (sub)tests *succeed.*

Remark. Actually, it suffices to use a seemingly weaker test in which the row-test and shift-test are combined into the following *generalized row-test*:

> Select a row-sequence $\bar{r}$ uniformly in $\mathbf{R}$, integers $i, j \in_{\mathrm{R}} [k]$ and $t \in_{\mathrm{R}} \{0, 1, \ldots, k-1\}$. Select a random extension of this row and its shift; namely, $m_1 \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$ and $m_2 \in_{\mathrm{R}} \mathbf{M}_j(\sigma^t(\bar{r}))$, and test if the $(k-t)$-long suffix of the ith row of $\Gamma(m_1)$ equals the $(k-t)$-long prefix of the jth row of $\Gamma(m_2)$.

Our main result asserts that Construction 2.1 is a "good consistency test": If it accepts Γ with high probability, then not only ALMOST ALL ENTRIES *in almost all matrices* are assigned in a consistent manner (which is obvious), but ALL ENTRIES IN ALMOST ALL ROWS *of almost all matrices* are assigned in a consistent manner.

LEMMA 2.1. *Suppose* $\mathbf{M}$ *satisfies Axioms* 1–4. *Then, for every constant* $\delta > 0$, *there exists a constant* $\epsilon > 0$ *so that if a function* $\Gamma : \mathbf{M} \mapsto V^{k \times k}$ *passes the consistency test with probability at least* $1 - \epsilon$, *then there exists a function* $\tau : S \mapsto V$ *so that, with probability at least* $1 - \delta$, *the value assigned by* Γ *to a uniformly chosen matrix matches the values assigned by* τ *to the elements of a uniformly chosen row in this matrix. Namely,*

$$\mathrm{Prob}_{i,m}(\forall j \;:\; \Gamma(m)_{i,j} = \tau(m_{i,j})) \;\; \geq \;\; 1 - \delta,$$

where $m \in_{\mathrm{R}} \mathbf{M}$ *and* $i \in_{\mathrm{R}} [k]$. *The constant* ϵ *does not depend on* k *and* S. *Furthermore, it is polynomially related to* δ.

As a corollary, we get part 1.1 of Lemma 1.2. Part 1.2 follows from Proposition 2.2 (below).

2.3. Proof of Lemma 2.1. As a motivation towards the proof of Lemma 2.1, consider the following mental experiment. Let $m \in \mathbf{M}$ be an arbitrary matrix and e be its (i, j)th entry. First, uniformly select a random matrix, denoted m_1, containing the ith row of m. Next, uniformly select a random matrix, denoted m_2, containing the jth column of m_1. One can show that m_2 is uniformly distributed among the matrices containing the element e. Thus, if Γ passes steps 1 and 2 in the consistency test, then it must assign consistent values to almost all elements in almost all matrices. Yet, this falls short of even proving that there exists an assignment which matches all values assigned to the elements of some row in some matrix. Indeed, consider a function Γ which assigns 0 to all elements in the first ϵk columns of each matrix and 1's to all other elements. Clearly, Γ passes the row-test with probability 1 and the column-test with probability greater than $1 - \epsilon$; yet, there is no $\tau : S \mapsto V$ so that for a random matrix the values assigned by Γ to some row match τ. It is easy to see that the shift-test takes care of this special counterexample. Furthermore, it may be telling to see what is wrong with some naive arguments. A main issue these arguments tend to ignore is that for an "adversarial" choice of Γ and a candidate choice of $\tau : S \mapsto V$, we have no handle on the (column) *location* of the elements in a random matrix on which τ disagrees with Γ. The shift-test plays a central role in circumventing this problem; see subsection 2.3.2 and Claim 2.1.14 (below).

Recommendation. The reader may want to skip the proofs of all claims on the first reading. We believe that all the claims are quite believable and that their proofs (though slightly tedious in some cases) are quite straightforward. In contrast, we believe that the ideas underlying the proof of the lemma are to be found in its high level structure; namely, the definitions and the claims made.

Notation. The following notation will be used extensively throughout the proof. For a k-by-k matrix, m, we denote by $\mathrm{row}_i(m)$ the ith row of m and by $\mathrm{col}^j(m)$ the jth column of m. Restating the conditions of the lemma, we have (from the hypothesis that Γ passes the column test)

$$\mathrm{Prob}_{\bar{c},i,j,m_1,m_2}(\mathrm{col}^i(\Gamma(m_1))=\mathrm{col}^j(\Gamma(m_2))) \geq 1-\epsilon, \tag{2.1}$$

where $\bar{c}, i, j, m_1,$ and m_2 are uniformly selected in the corresponding sets (i.e., $\bar{c}\in\mathbf{C}$, $i,j\in[k]$, $m_1 \in \mathbf{M}^i(\bar{c})$, and $m_2 \in \mathbf{M}^j(\bar{c})$). Similarly, from the hypothesis that Γ passes the row test, we have

$$\mathrm{Prob}_{\bar{r},i,j,m_1,m_2}(\mathrm{row}_i(\Gamma(m_1)) = \mathrm{row}_j(\Gamma(m_2))) \geq 1-\epsilon, \tag{2.2}$$

where $\bar{r} \in_{\mathrm{R}} \mathbf{R}$, $i, j \in_{\mathrm{R}} [k]$, $m_1 \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$, and $m_2 \in_{\mathrm{R}} \mathbf{M}_j(\bar{r})$. It will be convenient to extend the shift notation to matrices in the obvious manner; namely, $\sigma^t(m)$ is defined as the matrix m' satisfying $\mathrm{row}_i(m') = \sigma^t(\mathrm{row}_i(m))$ for every $i \in [k]$. From the hypothesis that Γ passes the shift-test, we obtain

$$\mathrm{Prob}_{m,t}(\forall j \leq k-t \quad \mathrm{col}^j(\Gamma(m)) = \mathrm{col}^{j+t}(\Gamma(\sigma^t(m)))) \geq 1-\epsilon, \tag{2.3}$$

where $m \in_{\mathrm{R}} \mathbf{M}$ and $t \in_{\mathrm{R}} [k-1]$. Finally, denoting by $\mathrm{entry}_{i,j}(m)$ the (i, j)th entry in the matrix m, we restate the conclusion of the lemma as follows:

$$\mathrm{Prob}_{i,m}(\exists j \text{ so that } \mathrm{entry}_{i,j}(\Gamma(m)) \neq \tau(\mathrm{entry}_{i,j}(m))) \leq \delta, \tag{2.4}$$

where $m \in_{\mathrm{R}} \mathbf{M}$ and $i \in_{\mathrm{R}} [k]$.

2.3.1. Stable rows and columns – part 1. For each $\bar{r} \in \mathbf{R}$ and $\bar{\alpha} \in V^k$, we denote by $p_{\bar{r}}(\bar{\alpha})$ the probability that Γ assigns to the row-sequence $\bar{r}$ the value-sequence $\bar{\alpha}$; namely,

$$p_{\bar{r}}(\bar{\alpha}) \overset{\mathrm{def}}{=} \mathrm{Prob}_{i,m}(\mathrm{row}_i(\Gamma(m)) = \bar{\alpha}),$$

where $i \in_{\mathrm{R}} [k]$ and $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$. Equation (2.2) implies that for almost all row-sequences there is a "typical" sequence of values; see Claim 2.1.3 (below).

DEFINITION 2.1.1 (consensus). *The* consensus *of a row-sequence $\bar{r} \in \mathbf{R}$, denoted* $\mathrm{con}(\bar{r})$, *is defined as the value $\bar{\alpha}$ for which $p_{\bar{r}}(\bar{\alpha})$ is maximum. Namely, $\mathrm{con}(\bar{r}) = \bar{\alpha}$ if $\bar{\alpha}$ is the (lexicographically first) value-sequence for which $p_{\bar{r}}(\bar{\alpha}) = \max_{\bar{\beta}}\{p_{\bar{r}}(\bar{\beta})\}$.*

DEFINITION 2.1.2 (stable sequences). *Let $\epsilon_2 \overset{\mathrm{def}}{=} \sqrt{\epsilon}$. We say that the row-sequence $\bar{r}$ is* stable *if $p_{\bar{r}}(\mathrm{con}(\bar{r})) \geq 1-\epsilon_2$. Otherwise, we say that $\bar{r}$ is* unstable.

Clearly, almost all row-sequences are stable.

CLAIM 2.1.3. *All but at most an ϵ_2 fraction of the row-sequence are stable.*

Proof. For each fixed $\bar{r}$ we have

$$\mathrm{Prob}_{i,j,m_1,m_2}(\mathrm{row}_i(\Gamma(m_1))=\mathrm{row}_j(\Gamma(m_2))) = \sum_{\bar{\alpha}} p_{\bar{r}}(\bar{\alpha})^2,$$

where $i, j \in_{\mathrm{R}} [k]$, $m_1 \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$, and $m_2 \in_{\mathrm{R}} \mathbf{M}_j(\bar{r})$. Taking the expectation over $\bar{r} \in_{\mathrm{R}} \mathbf{R}$, and using (2.2), we get

$$\begin{aligned} 1-\epsilon &\leq \mathrm{Prob}_{\bar{r},i,j,m_1,m_2}(\mathrm{row}_i(\Gamma(m_1)) = \mathrm{row}_j(\Gamma(m_2))) \\ &= \mathrm{Exp}_{\bar{r}}\left(\sum_{\bar{\alpha}} p_{\bar{r}}(\bar{\alpha})^2\right) \\ &\leq \mathrm{Exp}_{\bar{r}}(p_{\bar{r}}^{\max}), \end{aligned}$$

where $p_{\bar{r}}^{\max} \overset{\text{def}}{=} \max_{\bar{\alpha}}\{p_{\bar{r}}(\bar{\alpha})\}$. Using the Markov inequality, we get

$$\text{Prob}_{\bar{r}}(p_{\bar{r}}^{\max} \le 1 - \sqrt{\epsilon}) < \sqrt{\epsilon}$$

and the claim follows. □

By definition, almost all matrices containing a particular *stable* row-sequence assign this row-sequence the same sequence of values (i.e., its consensus value). We say that such matrices are conforming for this row-sequence.

Definition 2.1.4 (conforming matrix). *Let $i \in [k]$. A matrix $m \in \mathbf{M}$ is called* i-conforming (or conforming for row-position i) *if Γ assigns the* ith *row of m its consensus value; namely, if* $\text{row}_i(\Gamma(m)) = \text{con}(\text{row}_i(m))$. *Otherwise, the matrix is called* i-nonconforming (or nonconforming for row-position i).

Claim 2.1.5. *The probability that for a uniformly chosen $i \in [k]$ and $m \in \mathbf{M}$, the matrix m is i-nonconforming is at most $\epsilon_3 \overset{\text{def}}{=} 2\epsilon_2$. Furthermore, the bound holds also if we require that the* ith *row of m is stable.*

Proof. The stronger bound (on probability) equals the sum of the probabilities of the following two events. The first event is that the ith row of the matrix is unstable; whereas the second event is that the ith row of the matrix is stable and yet the matrix is i-nonconforming. To bound the probability of the first event (by ϵ_2), we fix any $i \in [k]$ and combine Axiom 3 with Claim 2.1.3. To bound the probability of the second event, we fix any stable $\bar{r}$ and use the definition of a stable row. □

Remark. Clearly, an analogous treatment can be applied to column-sequences. In the following, we freely refer to the above notions and to the above claims also when discussing column-sequences.

2.3.2. Stable rows – part 2 (shifts). Now we consider the relation between the consensus of row-sequences and the consensus of their (short) shifts. By a short shift of the row-sequence $\bar{r}$, we mean any row-sequence $\bar{s} = \sigma^d(\bar{r})$ obtained with $d \in \{-(k-1), \dots, +(k-1)\}$. Our aim is to show that the consensus (as well as stability) is usually preserved under short shifts.

Definition 2.1.6 (very-stable row). *Let $\epsilon_4 = \sqrt{\epsilon_2}$. We say that a row-sequence $\bar{r}$ is* very-stable *if it is stable, and for all but an ϵ_4 fraction of $d \in \{-(k-1), \dots, +(k-1)\}$, the row-sequence $\bar{s} \overset{\text{def}}{=} \sigma^d(\bar{r})$ is also stable.*

Claim 2.1.7. *All but at most an ϵ_4 fraction of the row-sequence are very-stable.*

Proof. By a simple counting argument (using the fact that the uniform distribution over $\mathbf{R}$ is preserved under shifts). □

Definition 2.1.8 (superstable row). *Let $\epsilon_5 = \sqrt[3]{\epsilon}$ and $\epsilon_6 = 2(\epsilon_4 + \epsilon_5)$. We say that a row-sequence $\bar{r}$ is* superstable *if it is very-stable, and, for every $j \in [k]$, the following holds: for all but an ϵ_6 fraction of the $t \in [k]$, the row-sequence $\bar{s} \overset{\text{def}}{=} \sigma^{t-j}(\bar{r})$ is stable and* $\text{con}_j(\bar{r}) = \text{con}_t(\bar{s})$, *where* $\text{con}_i(\bar{r})$ *is the* ith *element of* $\text{con}(\bar{r})$.

Note that the tth element of $\sigma^{t-j}(\bar{r})$ is $r_{t-(t-j)} = r_j$. Thus, a row-sequence is superstable if the consensus value of each of its elements is preserved under almost all (short) shifts.

Claim 2.1.9. *All but at most an ϵ_6 fraction of the row-sequence are superstable.*

Proof. We start by proving that almost all row-sequences and almost all of their shifts have approximately matching statistics, where the *statistics vector* of $\bar{r} \in \mathbf{R}$ is defined as the k-long sequence (of functions), $p_{\bar{r}}^1(\cdot), \dots, p_{\bar{r}}^k(\cdot)$, so that $p_{\bar{r}}^j(v)$ is the probability that Γ assigns the value v to the jth element of the row $\bar{r}$. Namely,

$$p_{\bar{r}}^j(v) \overset{\text{def}}{=} \text{Prob}_{i,m}(\text{entry}_{i,j}(\Gamma(m)) = v),$$

where $i \in_{\mathrm{R}} [k]$ and $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$. By the definition of consensus, we know that for every stable row-sequence $\bar{r} \in \mathbf{R}$, we have $p^j_{\bar{r}}(\mathrm{con}_j(\bar{r})) \geq 1 - \epsilon_2$, for every $j \in [k]$. Thus, if both $\bar{r}$ and its shift $\bar{s} = \sigma^t(\bar{r})$ are stable and have approximately matching statistics (i.e., the corresponding $(k-t)$-long statistics subvectors are close), then their consensus must match (i.e., the corresponding $(k-t)$-long subsequences of the consensus are equal).

Subclaim 2.1.9.1. For all but an ϵ_5 fraction of the row-sequences $\bar{r}$, all but an ϵ_5 fraction of the values $d \in [k-1]$ satisfy

$$\sum_v |p^j_{\bar{r}}(v) - p^{j+d}_{\sigma^d(\bar{r})}(v)| < 2\epsilon_5 \quad \text{for every } j \leq k - d.$$

Proof of subclaim. Let *pref*$\mathrm{row}_{i,j}(m)$ denote the j-long prefix of $\mathrm{row}_i(m)$ and *suff*$\mathrm{row}_{i,j}(m)$ its j-long suffix. By the shift-test (see (2.3) and recall $\epsilon = \epsilon_5^3$)

$$\mathrm{Prob}_{m,i,d}(\mathit{pref}\mathrm{row}_{i,k-d}(\Gamma(m)) = \mathit{suff}\mathrm{row}_{i,k-d}(\Gamma(m'))) \geq 1 - \epsilon_5^3,$$

where $i \in_{\mathrm{R}} [k]$, $m \in_{\mathrm{R}} \mathbf{M}$, $d \in_{\mathrm{R}} [k-1]$, and $m' = \sigma^d(m)$. Using Axiom 3 (part 2) and an averaging argument, we get that for all but an ϵ_5 fraction of the $\bar{r} \in \mathbf{R}$, and for all but an ϵ_5 fraction of $d \in [k-1]$,

$$\mathrm{Prob}_{i,m}(\mathit{pref}\mathrm{row}_{i,k-d}(\Gamma(m)) = \mathit{suff}\mathrm{row}_{i,k-d}(\Gamma(m'))) \geq 1 - \epsilon_5, \tag{2.5}$$

where $i \in_{\mathrm{R}} [k]$, $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$, and $m' = \sigma^d(m)$. We fix a pair $\bar{r}$ and d satisfying (2.5), thus fixing also $\bar{s} = \sigma^d(\bar{r})$. A matrix pair (m, m') for which the equality *pref*$\mathrm{row}_{i,k-d}(\Gamma(m)) =$ *suff*$\mathrm{row}_{i,k-d}(\Gamma(m'))$ holds contributes equally to the (appropriate $(k-d)$-long portion of) statistic vectors of the row-sequences $\bar{r}$ and $\bar{s}$. The contribution of a matrix pair, for which the equality does not hold for the difference $\sum_v |p^j_{\bar{r}}(v) - p^{j+d}_{\bar{s}}(v)|$, is at most $\frac{2}{k \cdot |\mathbf{M}_i(\bar{r})|}$ per each relevant j. Thus, the total difference for such $\bar{r}$ and $\bar{s}$ (i.e., satisfying (2.5)) is at most $2\epsilon_5$. The subclaim follows. □

As a corollary we get the following.

Subclaim 2.1.9.2. Let us call a row-sequence, $\bar{r}$, *infective* if for every $j \in [k]$ all but an $2\epsilon_5$ fraction of the $t \in [k]$ satisfy $\sum_v |p^j_{\bar{r}}(v) - p^t_{\bar{s}}(v)| \leq 2\epsilon_5$, where $\bar{s} = \sigma^{t-j}(\bar{r})$. Then, all but a $2\epsilon_5$ fraction of the row-sequences are infective.

Proof of subclaim. We say that $\bar{r}$ is *rightwards-fine* if for all but an ϵ_5 fraction of the $d \in [k]$ and for every $j \leq k - d$, we have $\sum_v |p^j_{\bar{r}}(v) - p^{j+d}_{\sigma^d(\bar{r})}(v)| \leq 2\epsilon_5$. (Indeed, Subclaim 2.1.9.1 asserts that all but an ϵ_5 fraction of the row-sequences are rightwards-fine.) If $\bar{r}$ is rightwards-fine, then for every j there are at most $\epsilon_5 k$ positions $t \in \{j+1, \ldots, k\}$ so that $\sum_v |p^j_{\bar{r}}(v) - p^t_{\sigma^{t-j}(\bar{r})}(v)| > 2\epsilon_5$. Similarly, $\bar{r}$ is *leftwards-fine* if for all but an ϵ_5 fraction of the $d \in [k]$ and for every $j > d$, we have $\sum_v |p^j_{\bar{r}}(v) - p^{j-d}_{\sigma^{-d}(\bar{r})}(v)| \leq 2\epsilon_5$, and whenever $\bar{r}$ is leftwards-fine, then for every j there are at most $\epsilon_5 k$ positions $t \in \{1, \ldots, j-1\}$ so that $\sum_v |p^j_{\bar{r}}(v) - p^t_{\sigma^{t-j}(\bar{r})}(v)| > 2\epsilon_5$. Thus, if a row-sequence $\bar{r}$ is both rightwards-fine and leftwards-fine, then for every $j \in [k]$ all but a $2\epsilon_1$ fraction of the positions $t \in [k]$ satisfy $\sum_v |p^j_{\bar{r}}(v) - p^t_{\sigma^{t-j}(\bar{r})}(v)| \leq 2\epsilon_5$. Now, by Subclaim 2.1.9.1, all but an ϵ_5 fraction of the row-sequences are rightwards-fine. A similar statement holds for leftwards-fine (since the shift-test can be rewritten as selecting $m' \in_{\mathrm{R}} \mathbf{M}$ and $d \in_{\mathrm{R}} [k-1]$ and setting $m = \sigma^{-d}(m')$). Combining all these trivialities, the subclaim follows. □

Clearly, a row-sequence $\bar{r}$ that is both very-stable and infective satisfies, for every $j \in [k]$ and all but at most $\epsilon_4 \cdot (2k-1) + 2\epsilon_5 \cdot k$ of the $t \in [k]$, both

- $\bar{s} \overset{\text{def}}{=} \sigma^{t-j}(\bar{r})$ is stable; it follows that $p^t_{\bar{s}}(\text{con}_t(\bar{s})) \geq 1-\epsilon_2$ and $p^t_{\bar{s}}(u) \leq \epsilon_2$ for all $u \neq \text{con}_t(\bar{s})$.
- $p^t_{\bar{s}}(v) \geq p^j_{\bar{r}}(v) - 2\epsilon_5$, for every v and in particular for $v = \text{con}_j(\bar{r})$.

It follows that $p^t_{\bar{s}}(\text{con}_j(\bar{r})) \geq p^j_{\bar{r}}(\text{con}_j(\bar{r})) - 2\epsilon_5 \geq 1-\epsilon_2-2\epsilon_5$ which (for sufficiently small ϵ) is strictly greater than ϵ_2, and therefore $\text{con}_j(\bar{r}) = \text{con}_t(\bar{s})$ must hold. Thus, such an $\bar{r}$ is superstable. Combining the lower bounds on the fractions of very-stable and infective row-sequences (given by Claim 2.1.7 and Subclaim 2.1.9.2, respectively), the current claim follows. (Actually, we get a better bound; i.e., $\epsilon_4 + 2\epsilon_5$.) □

Summary. Before proceeding let us summarize our state of knowledge. The key definitions regarding row-sequences are of stable, very-stable, and superstable row-sequences (i.e., Definitions 2.1.2, 2.1.6, and 2.1.8, respectively). Recall that a stable row-sequence is assigned the same value in almost all matrices in which it appears. Furthermore, most prefixes (resp., suffices) of a superstable row-sequence are assigned the same values in almost all matrices containing these portions (as part of some row). Regarding matrices, we defined a matrix to be i-conforming if it assigns its ith row the corresponding consensus value (i.e., it conforms with the consensus of that row-sequence); cf. Definitions 2.1.4 and 2.1.1. We have seen that almost all row-sequences are superstable and that almost all matrices are conforming for most of their rows. Actually, we will use the latter fact with respect to columns; that is, almost all matrices are conforming for most columns (cf. Claim 2.1.5 and the remark following it).

2.3.3. Deriving the conclusion of the lemma. We are now ready to derive the conclusion of the lemma. Loosely speaking, we claim that the function τ, defined so that $\tau(e)$ is the value most frequently assigned (by Γ) to e, satisfies (2.4). Actually, we use a slightly different definition for the function τ.

DEFINITIONS 2.1.10 (the function τ). *For a column-sequence $\bar{c}$, we denote by $\text{con}_i(\bar{c})$ the values that $\text{con}(\bar{c})$ assigns to the i*th *element in $\bar{c}$. We denote by $\mathbf{C}_i(e)$ the set of column-sequences having e as the i*th *component. Let $q_e(v)$ denote the probability that the consensus of a uniformly chosen column-sequence, containing e, assigns to e the value v. Namely,*

$$q_e(v) \overset{\text{def}}{=} \text{Prob}_{i,\bar{c}}(\text{con}_i(\bar{c})=v),$$

where $i \in_{\text{R}} [k]$ and $\bar{c} \in_{\text{R}} \mathbf{C}_i(e)$. We consider $\tau : S \mapsto V$ so that $\tau(e) \overset{\text{def}}{=} v$ if $q_e(v) = \max_u\{q_e(u)\}$, with ties broken arbitrarily.

Assume, contrary to our claim, that (2.4) does not hold (for this τ). Namely, for a uniformly chosen $m \in \mathbf{M}$ and $i \in [k]$, the following holds with probability greater that δ

$$\exists j \text{ so that } \text{entry}_{i,j}(\Gamma(m)) \neq \tau(\text{entry}_{i,j}(m)). \tag{2.6}$$

The notion of an annoying row-sequence, defined below, plays a central role in our argument. Using the above (contradiction) hypothesis, we first show that many row-sequences are annoying. Next, we show that lower bounds on the number of annoying row-sequences translate to lower bounds on the probability that a uniformly chosen matrix is nonconforming for a uniformly chosen column position. This yields a contradiction to Claim 2.1.5.

DEFINITIONS 2.1.11 (row-annoying elements). *An element r_j in $\bar{r} = (r_1, \ldots, r_k) \in \mathbf{R}$, is said to be* annoying for the row-sequence $\bar{r}$ *if the* jth *element in* $\text{con}(\bar{r})$ *differs* *from $\tau(r_j)$. A row-sequence $\bar{r}$ is said to be* annoying *if $\bar{r}$ contains an element that is annoying for it.*

Using Claim 2.1.9, we get the following claim.

CLAIM 2.1.12. *Suppose that* (2.4) *does not hold* (for τ). *Then, at least a $\delta_1 \stackrel{\text{def}}{=} \delta - \epsilon_6 - \epsilon_2$ fraction of the row-sequences are both superstable and annoying.*

Proof. Axiom 3 (part 2) is extensively used throughout this proof (with no explicit reference). Combining (2.6) and Claim 2.1.9, with probability at least $\delta - \epsilon_6 - \epsilon_2 = \delta_1$, a uniformly chosen pair $(m, i) \in \mathbf{M} \times [k]$ satisfies the following:

1. there exists a j so that $\tau(\text{entry}_{i,j}(m))$ is different from $\text{entry}_{i,j}(\Gamma(m))$;
2. $\text{row}_i(m)$ is superstable;
3. matrix m is i-conforming; i.e., $\text{entry}_{i,j}(\Gamma(m))$ equals $\text{con}_j(\text{row}_i(m))$, for every $j \in [k]$.

Combining conditions (1) and (3), we get that $e = \text{entry}_{i,j}(m)$ is annoying for the ith row of m. The current claim follows. □

A key observation is that each stable row-sequence which is annoying yields many matrices which are nonconforming for the "annoying column position" (i.e., for the column position containing the element which annoys this row-sequence).

CLAIM 2.1.13. *Suppose that a row-sequence $\bar{r} = (r_1, \ldots, r_k)$ is stable and that r_j is annoying for $\bar{r}$. Then at least a $\frac{1}{2} - \epsilon_2$ fraction of the matrices containing the row-sequence $\bar{r}$ are nonconforming for column-position j.*

We stress that the row-sequence $\bar{r}$ in the above claim is *not* necessarily very-stable (let alone superstable).

Proof. Let us denote by v the value assigned to r_j by the consensus of $\bar{r}$ (i.e., $v \stackrel{\text{def}}{=} \text{con}_j(\bar{r})$). Since r_j annoys $\bar{r}$ it follows that v is different from $\tau(r_j)$. Consider the probability space defined by uniformly selecting $i \in [k]$ and $m \in \mathbf{M}_i(\bar{r})$. Since $\bar{r}$ is stable it follows that in almost all of these matrices the value assigned to r_j by the matrix equals v. Namely,

$$\text{Prob}_{i,m}(\text{entry}_{i,j}(\Gamma(m)) = v)) \quad \geq \quad 1 - \epsilon_2, \tag{2.7}$$

where $i \in_{\text{R}} [k]$ and $m \in_{\text{R}} \mathbf{M}_i(\bar{r})$. By Axiom 4, the jth column of m is uniformly distributed in $\mathbf{C}_i(r_j)$, and thus we may replace $\bar{c} \in_{\text{R}} \mathbf{C}_i(r_j)$ by the jth column of $m \in_{\text{R}} \mathbf{M}_i(\bar{r})$. Now, using the definition of the function τ and the accompanying notations, we get

$$\text{Prob}_{i,m}(\text{con}_i(\text{col}^j(m)) = v) \quad = \quad q_{r_j}(v) \quad \leq \quad \frac{1}{2}\ , \tag{2.8}$$

where, again, $i \in_{\text{R}} [k]$ and $m \in_{\text{R}} \mathbf{M}_i(\bar{r})$. The inequality holds since $v \neq \tau(r_j)$ and by τ's definition $q_{r_j}(v) \leq q_{r_j}(\tau(r_j))$. Combining (2.7) and (2.8), we get

$$\begin{aligned}
&\text{Prob}_{i,m}(\text{entry}_{i,j}(\Gamma(m)) \neq \text{con}_i(\text{col}^j(m))) \\
&\qquad\qquad \geq \text{Prob}_{i,m}(\text{entry}_{i,j}(\Gamma(m)) = v \;\&\; \text{con}_i(\text{col}^j(m)) \neq v) \\
&\qquad\qquad \geq 1 - \epsilon_2 - \frac{1}{2} \;=\; \frac{1}{2} - \epsilon_2
\end{aligned}$$

and the claim follows. □

Another key observation is that superstable row-sequences which are annoying have the property of "infecting" almost all their shifts with their annoying positions, thus spreading the "annoyance" over all column positions.

CLAIM 2.1.14. *Suppose that a row-sequence $\bar{r}$ is both superstable and annoying. In particular, suppose that the jth element of $\bar{r} = (r_1, \ldots, r_k)$ is annoying for $\bar{r}$. Then, for all but at most an ϵ_6 fraction of the $t \in [k]$, the row-sequence $\bar{s} = \sigma^{t-j}(\bar{r})$ is stable and its tth element* (which is indeed r_j) *is annoying for $\bar{s}$.*

Proof. Since $\bar{r}$ is superstable, we know that for all but an ϵ_6 fraction of the t's, $\mathrm{con}_j(\bar{r}) = \mathrm{con}_t(\bar{s})$ and $\bar{s}$ is stable (as well), where $\bar{s} = (s_1, \ldots, s_k) = \sigma^{t-j}(\bar{r})$. Since r_j is annoying for $\bar{r}$, we have $\mathrm{con}_j(\bar{r}) \neq \tau(r_j)$ and $\mathrm{con}_t(\bar{s}) \neq \tau(r_j) = \tau(s_t)$ follows (recall $r_j = s_t$). □

Combining Claims 2.1.12 and 2.1.14, we derive, for almost all positions $t \in [k]$, a lower bound for the number of stable row-sequences that are annoyed by their tth element.

CLAIM 2.1.15. *Suppose that* (2.4) *does not hold* (for τ). *Then, there exists a set $T \subseteq [k]$ so that $|T| \geq (1 - 2\epsilon_6) \cdot k$ and for every $t \in T$ there is a set of at least $\frac{\delta_1}{2k} \cdot |\mathbf{R}|$ stable row-sequences so that the tth position is annoying for each of these sequences.*

Proof. Combining Claims 2.1.12 and 2.1.14, we get that there is a set of super stable row-sequences $A \subseteq \mathbf{R}$ so that

1. A contains at least a δ_1 fraction of $\mathbf{R}$; and
2. for every $\bar{r} \in A$ there exists a $j_{\bar{r}} \in [k]$ so that for all but an ϵ_6 of the $t \in [k]$, the row-sequence $\bar{s} \overset{\mathrm{def}}{=} \sigma^{t-j_{\bar{r}}}(\bar{r})$ is stable and the tth position is annoying for it (i.e., for $\bar{s}$).

By a counting argument it follows that there is a set T so that $|T| \geq (1 - 2\epsilon_6) \cdot k$, and for every $t \in T$ at least half of the $\bar{r}$'s in A satisfy item 2 above for this t (i.e., $\bar{s} \overset{\mathrm{def}}{=} \sigma^{t-j_{\bar{r}}}(\bar{r})$ is stable and the tth position is annoying for $\bar{s}$). Fixing such a $t \in T$, we consider the set, denoted A_t, containing these $\bar{r}$'s; namely, for every $\bar{r} \in A_t$ the row-sequence $\bar{s} \overset{\mathrm{def}}{=} \sigma^{t-j_{\bar{r}}}(\bar{r})$ is stable and the tth position is annoying for it (i.e., for $\bar{s}$). Thus, we have established a mapping from A_t to a set of stable row-sequences which are annoyed by their tth position; specifically, $\bar{r}$ is mapped to $\sigma^{t-j_{\bar{r}}}(\bar{r})$. Each row-sequence in the range of this mapping has at most k preimages (corresponding to the k possible shifts which maintain its tth element). Recalling that A_t contains at least $\frac{|A|}{2} \geq \frac{\delta_1}{2} \cdot |\mathbf{R}|$ sequences, we conclude that the mapping's range must contain at least $\frac{\delta_1}{2k} \cdot |\mathbf{R}|$ sequences, and the claim follows. □

Combining Claims 2.1.15 and 2.1.13, we get a lower bound on the number of matrices which are nonconforming for the jth column, for all $j \in T$ (where T is as in Claim 2.1.15).

CLAIM 2.1.16. *Let T be as guaranteed by Claim* 2.1.15 *and suppose that $j \in T$. Then, at least a $\frac{\delta_1}{6}$ fraction of the matrices are nonconforming for column-position j.*

Proof. By Claim 2.1.15, there are at least $\frac{\delta_1}{2k} \cdot |\mathbf{R}|$ stable row-sequences that are annoyed by their jth position. Out of these row-sequences, we consider a subset, denoted A, containing exactly $\frac{\delta_1}{2k} \cdot |\mathbf{R}|$ row-sequences. By Claim 2.1.13, for each $\bar{r} \in A$, at least a $\frac{1}{2} - \epsilon_2$ fraction of the matrices containing the row-sequence $\bar{r}$ are nonconforming for column-position j. We claim that almost all of these matrices do not contain another row-sequence in A (here we use the fact that A isn't too large); this will allow us to add up the matrices guaranteed by each $\bar{r} \in A$ without worrying about multiple counting.

Subclaim 2.1.16.1. For every $\bar{r} \in \mathbf{R}$

$$\mathrm{Prob}_{i,m}(\exists i' \neq i \text{ s.t. } \mathrm{row}_{i'}(m) \in A) < \frac{\delta_1}{2},$$

where $i \in_{\mathrm{R}} [k]$ and $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$.

Proof of subclaim. By Axiom 3 (part 3), we get that for every $i' \neq i$ the i'th row of $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$ is uniformly distributed in $\mathbf{R}$. Thus, for every $i' \neq i$

$$\mathrm{Prob}_m(\mathrm{row}_{i'}(m) \in A) = \frac{\delta_1}{2k},$$

where $m \in_{\mathrm{R}} \mathbf{M}_i(\bar{r})$. The subclaim follows. □

Using the subclaim, we conclude that for each $\bar{r} \in A$, at least a $\frac{1}{2} - \epsilon_2 - \frac{\delta_1}{2} > \frac{1}{3}$ fraction of the matrices containing the row-sequence $\bar{r}$ are nonconforming for column-position j and do not contain any other row-sequence in A. The desired lower bound now follows. Namely, let B denote the set of matrices which are nonconforming for column-position j, let $B_i(\bar{r}) \stackrel{\mathrm{def}}{=} B \cap \mathbf{M}_i(\bar{r})$ and $B'_i(\bar{r})$ denote the set of matrices in $B_i(\bar{r})$ which do not contain any row in A except for the ith row; then

$$\begin{aligned}|B| &\geq |\cup_{\bar{r}\in A} \cup_{i=1}^{k} B'_i(\bar{r})| \\ &= \sum_{\bar{r}\in A} \sum_{i=1}^{k} |B'_i(\bar{r})| \\ &> \sum_{\bar{r}\in A} \sum_{i=1}^{k} \frac{|\mathbf{M}_i(\bar{r})|}{3} \\ &= |A| \cdot \left(\frac{1}{3} \cdot \frac{k \cdot |\mathbf{M}|}{|\mathbf{R}|}\right) \\ &= \frac{\delta_1}{6} \cdot |\mathbf{M}|\end{aligned}$$

The claim follows. □

The combination of Claims 2.1.15 and 2.1.16 yields that a uniformly chosen matrix is nonconforming for a uniformly chosen column position with probability at least $(1 - 2\epsilon_6) \cdot \frac{\delta_1}{6}$. For a suitable choice of constants (e.g., $\epsilon = (\delta/30)^4$), this yields a contradiction to Claim 2.1.5 (which asserts that this probability is at most ϵ_3).[5] Thus, (2.4) must hold for τ as defined in Definition 2.1.10, and the lemma follows. □

2.4. A construction that satisfies the axioms. Clearly, the set of all k-by-k matrices over S satisfies Axioms 1–4.[6] A more interesting and useful set of matrices is defined as follows.

CONSTRUCTION 2.2 (basic construction). *We associate the set S with a finite field of characteristic at least k. Furthermore, $[k]$ is associated with k elements of the field so that* 1 *is the multiplicative unit and $i \in [k]$ is the sum of i such units. Let* $\mathbf{M}$ *be the set of matrices defined by four field elements as follows. The matrix associated with the quadruple (x, y, x', y') has the (i, j)th entry equal to $(x + jy) + i(x' + jy')$.*

Remark. The column-sequences correspond to the standard pairwise-independent sequences $\{r + is : i \in [k]\}$, where $r, s \in S$. Similarly, the row-sequences are expressed as $\{r + js : j \in [k]\}$, where $r, s \in S$.

PROPOSITION 2.2. *The basic construction satisfies Axioms* 1–4.

[5] Specifically, contradiction follows when $(1 - 2\epsilon_6) \cdot \frac{\delta_1}{6} > \frac{\delta_1}{12} > \epsilon_3 = 2\epsilon_2$. Using $\delta_1 = \delta - \epsilon_6 - \epsilon_2$, we need to have $\epsilon_6 \leq 1/4$ and $\delta > 25\epsilon_2 + \epsilon_6$. Using $\epsilon_2 = \sqrt{\epsilon}$ and $\epsilon_6 = 2(\sqrt[4]{\epsilon} + \sqrt[3]{\epsilon}) < 4\sqrt[4]{\epsilon}$, it suffices to have $\epsilon \leq 2^{-16}$ and $\delta > 29\sqrt[4]{\epsilon}$, which holds for $\epsilon = \min\{2^{-16}, (\delta/30)^4\}$ ($= (\delta/30)^4$ as $\delta \leq 1$).

[6] To see that Axiom 2 holds, one should specify the right shift of $\bar{r} = (r_1, \ldots, r_k)$. A natural choice is to have $\sigma(\bar{r}) = (r_k, r_1, \ldots, r_{k-1})$.

Proof. Axiom 1 as well as the first two items of Axiom 3 are obvious from the above remark. The right-shift of the sequence $\{r+js : j \in [k]\}$ is $\{(r+s)+js : j \in [k]\}$ and Axiom 2 follows. To prove that the third item of Axiom 3 holds, we rewrite the ith row as $\{s_i + j \cdot r_i : j \in [k]\}$, where $s_i = x + ix'$ and $r_i = y + iy'$. Now, for every $i \neq i' \in [k]$, when $x, y, x', y' \in_{\mathrm{R}} S$, the pairs (s_i, r_i) and $(s_{i'}, r_{i'})$ are pairwise-independent and uniformly distributed in $S \times S$, which corresponds to the set of row-sequences. It remains to prove that Axiom 4 holds. We start by proving the following.

FACT 2.2.1. *Consider any $i, j \in [k]$ and two sequences $\bar{r} = (r_1, \ldots, r_k) \in \mathbf{R}$ and $\bar{c} = (c_1, \ldots, c_k) \in \mathbf{C}$ so that $r_j = c_i$. Then, $|\mathbf{M}_i^j(\bar{r}, \bar{c})|$ equals $|S|$.*

Proof of fact. By the construction there exists a unique pair $(a, b) \in S \times S$ so that $a + j'b = r_{j'}$ for every $j' \in [k]$ (existence is obvious and uniqueness follows by considering any two equations; e.g., $a + b = r_1$ and $a + 2b = r_2$). Similarly, there exists a unique pair (α, β) so that $\alpha + i'\beta = c_{i'}$ for every $i' \in [k]$. We get a system of four linear equations in $x, x', y,$ and y' (i.e., $x + ix' = a$, $y + iy' = b$, $x + jy = \alpha$, and $x' + jy' = \beta$). This system has rank 3 and thus $|S|$ solutions, each defining a matrix in $\mathbf{M}_i^j(\bar{r}, \bar{c})$. ◻

Using Fact 2.2.1, Axiom 4 follows since

$$\begin{aligned}
\frac{|\mathbf{M}_i^j(\bar{r}, \bar{c})|}{|\mathbf{M}_i(\bar{r})|} &= \frac{|S|}{|S \times S|} \\
&= \frac{1}{|S|} \\
&= \frac{1}{|\mathbf{C}_i(r_j)|}
\end{aligned}$$

and so does the proposition. ◻

3. A stronger consistency test and the PCP application.

To prove Lemma 1.3, we need a slightly stronger consistency test than the one analyzed in Lemma 2.1. This new test is given access to three related oracles, each supplying assignments to certain classes of sequences over S, and is supposed to establish the consistency of these oracles with one function $\tau : S \mapsto V$. Specifically, one oracle assigns values to k^2-long sequences viewed as 2-dimensional arrays (as before). The other two oracles assign values to k^3-long sequences viewed as 3-dimensional arrays, whose slices (along a specific coordinate) correspond to the 2-dimensional arrays of the first oracle. Using Lemma 2.1 (and the auxiliary oracles) we will present a test which verifies that the first oracle is consistent in an even stronger sense than established in Lemma 2.1. Namely, not only that ALL ENTRIES IN ALMOST ALL ROWS *of almost all* 2*-dimensional arrays* are assigned in a consistent manner, but ALL ENTRIES *in almost all* 2*-dimensional arrays* are assigned in a consistent manner.

3.1. The setting.

Let S, k, $\mathbf{R}$, $\mathbf{C}$, and $\mathbf{M}$ be as in the previous section. We now consider a family, $\mathcal{M}_{\mathtt{c}}$, of k-by-k matrices with entries in $\mathbf{C}$. The family $\mathcal{M}_{\mathtt{c}}$ will satisfy Axioms 1–4 of the previous section. In addition, its induced multiset of row-sequences, denoted $\mathcal{R}$, will correspond to the multiset $\mathbf{M}$; namely, each row of a matrix in $\mathcal{M}_{\mathtt{c}}$ will form a matrix in $\mathbf{M}$ (i.e., the sequence of elements of $\mathbf{C}$ corresponding to a row in a $\mathcal{M}_{\mathtt{c}}$-matrix will correspond to a $\mathbf{M}$-matrix).

AXIOM 5. *For every $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ and every $i \in [k]$, there exists $m \in \mathbf{M}$ so that for every $j \in [k]$, the (i,j)*th *entry of* $\mathtt{m}$ *equals the j*th *column of m* (i.e., $\mathrm{entry}_{i,j}(\mathtt{m}) =$

$\mathrm{col}^j(m)$, or, equivalently, $\mathrm{row}_i(\mathtt{m}) \cong m$). *Furthermore, this matrix m is unique.*[7] Analogously, we consider also a family, $\mathcal{M}_{\mathtt{r}}$, of k-by-k matrices, the entries of which are elements in $\mathbf{R}$ so that the rows[8] of each $\mathtt{m} \in \mathcal{M}_{\mathtt{r}}$ correspond to matrices in $\mathbf{M}$.

3.2. The test. As before, Γ is a function assigning (k-by-k) matrices in $\mathbf{M}$ values which are k-by-k matrices over some set of values V (i.e., $\Gamma : \mathbf{M} \mapsto V^{k\times k}$). Let $\Gamma_{\mathtt{c}}$ (resp., $\Gamma_{\mathtt{r}}$) be (the supposedly corresponding) function assigning k-by-k matrices over $\mathbf{C}$ (resp., $\mathbf{R}$) values which are k-by-k matrices over $\overline{V} \overset{\text{def}}{=} V^k$ (i.e., $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto \overline{V}^{k\times k}$).

CONSTRUCTION 3.1 (extended consistency test):

1. Consistency for sequences. *Apply the consistency test of Construction* 2.1 *to* $\Gamma_{\mathtt{c}}$. *Same for* $\Gamma_{\mathtt{r}}$.
2. Correspondence test. *Uniformly select a matrix* $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ *and a row* $i \in [k]$, *and compare the* ith *row in* $\Gamma_{\mathtt{c}}(\mathtt{m})$ *to* $\Gamma(m)$, *where* $m \in \mathbf{M}$ *is the matrix formed by the* $\mathbf{C}$*-elements in the* ith *row of* $\mathtt{m}$. *The same goes for* $\Gamma_{\mathtt{r}}$.

The test accepts if both (sub)tests *succeed.*

LEMMA 3.1. *Suppose* $\mathbf{M}, \mathcal{M}_{\mathtt{c}}$, and $\mathcal{M}_{\mathtt{r}}$ *satisfy Axioms* 1–5. *Then, for every constant* $\gamma > 0$, *there exists a constant* ϵ *so that if a function* $\Gamma : \mathbf{M} \mapsto V^{k\times k}$ (together with some functions $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto \overline{V}^{k\times k}$ and $\Gamma_{\mathtt{r}} : \mathcal{M}_{\mathtt{r}} \mapsto \overline{V}^{k\times k}$) *passes the extended consistency test with probability at least* $1-\epsilon$, *then there exists a function* $\tau : S \mapsto V$ *so that, with probability at least* $1-\gamma$, *the value assigned by* Γ *to a uniformly chosen matrix* $m \in \mathbf{M}$ *matches the values assigned by* τ *to each of the elements of* m. *Namely,*

$$\mathrm{Prob}_m\left(\forall i,j \ \ \mathrm{entry}_{i,j}(\Gamma(m)) = \tau(\mathrm{entry}_{i,j}(m))\right) \geq 1-\gamma,$$

where $m \in_{\mathrm{R}} \mathbf{M}$. *The constant* ϵ *does not depend on* k *and* S. *Furthermore, it is polynomially related to* γ.

The proof of the lemma starts by applying Lemma 2.1 to derive assignments to $\mathbf{C}$ (resp., $\mathbf{R}$) which are consistent with $\Gamma_{\mathtt{c}}$ (resp., $\Gamma_{\mathtt{r}}$) on almost all rows of almost all k^3-dimensional arrays (i.e., $\mathcal{M}_{\mathtt{c}}$ and $\mathcal{M}_{\mathtt{r}}$, resp.). It proceeds by applying a degenerate argument of the kind applied in the proof of Lemma 2.1. Again, the reader may want to skip the proofs of all claims in first reading.

3.3. Proof of Lemma 3.1. We start by considering step 1 in the extended consistency test. By Lemma 2.1, there exists a function $\tau_{\mathtt{c}} : \mathbf{C} \mapsto V^k$ (resp., $\tau_{\mathtt{r}} : \mathbf{R} \mapsto V^k$) so that the value assigned by $\Gamma_{\mathtt{c}}$ (resp., $\Gamma_{\mathtt{r}}$), to a uniformly chosen row in a uniformly chosen matrix $\mathcal{M}_{\mathtt{c}}$ (resp., $\mathcal{M}_{\mathtt{r}}$) matches, with high probability, the values assigned by $\tau_{\mathtt{c}}$ (resp., $\tau_{\mathtt{r}}$) to each of the $\mathbf{C}$-elements (resp., $\mathbf{R}$-elements) appearing in this row. Here "with high probability" means with probability at least $1-\delta$, where $\delta > 0$ is a constant, related to ϵ as specified by Lemma 2.1. Namely,

$$\mathrm{Prob}_{i,\mathtt{m}}(\forall j \ \ \mathrm{entry}_{i,j}(\Gamma_{\mathtt{c}}(\mathtt{m})) = \tau_{\mathtt{c}}(\mathrm{entry}_{i,j}(\mathtt{m}))) \geq 1-\delta, \tag{3.1}$$

where $i \in_{\mathrm{R}} [k]$ and $\mathtt{m} \in_{\mathrm{R}} \mathcal{M}_{\mathtt{c}}$.

[7] Uniqueness is an issue only in case $\mathbf{M}$ is a multiset. In such a case, $\mathcal{M}_{\mathtt{c}}$ will be a multiset too, and the furthermore clause establishes a 1-1 correspondence betwen the rows of $\mathcal{M}_{\mathtt{c}}$ and $\mathbf{M}$.

[8] Alternatively, one can consider a family, $\mathcal{M}_{\mathtt{r}}$, of k-by-k matrices, the entries of which are elements in $\mathbf{R}$, so that the columns of each $\mathtt{m} \in \mathcal{M}_{\mathtt{r}}$ correspond to matrices in $\mathbf{M}$. However, this would require modifying the basic consistency test (of Construction 2.1) for these matrices, so that it shifts columns instead of rows.

3.3.1. Perfect matrices and typical sequences. Equation (3.1) relates $\tau_{\mathtt{c}}$ to $\Gamma_{\mathtt{c}}$ (resp., $\tau_{\mathtt{c}}$ to $\Gamma_{\mathtt{c}}$). Our next step is to relate $\tau_{\mathtt{c}}$ (resp., $\tau_{\mathtt{r}}$) to Γ. This is done easily by referring to step 2 in the extended consistency test. Specifically, it follows that the value assigned by Γ, to a uniformly chosen matrix $m \in \mathbf{M}$, matches, with high probability, the values assigned by $\tau_{\mathtt{c}}$ (resp., $\tau_{\mathtt{r}}$) to each of the columns (resp., rows) of m.

Definition 3.1.1 (perfect matrices). *A matrix $m \in \mathbf{M}$ is called* perfect (for columns) *if for every $j \in [k]$, the jth column of $\Gamma(m)$ equals the value assigned by $\tau_{\mathtt{c}}$ to the jth column of m (i.e., $\mathrm{col}^j(\Gamma(m)) = \tau_{\mathtt{c}}(\mathrm{col}^j(m))$). Similarly, $m \in \mathbf{M}$ is called* perfect (for rows) *if $\mathrm{row}_i(\Gamma(m)) = \tau_{\mathtt{r}}(\mathrm{row}_i(m))$, for every $i \in [k]$.*

Claim 3.1.2 (perfect matrices). *Let $\delta_1 \stackrel{\mathrm{def}}{=} \delta + \epsilon$.*
(c) *All but a δ_1 fraction of the matrices in $\mathbf{M}$ are perfect for columns.*
(r) *All but a δ_1 fraction of the matrices in $\mathbf{M}$ are perfect for rows.*

Proof. It will be convenient to view the rows of $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ as elements of $\mathbf{M}$ (although, formally, we only have a correspondence between the ith row of $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ and a matrix $m \in \mathbf{M}$ so that $\mathrm{entry}_{i,j}(\mathtt{m}) = \mathrm{col}^j(m)$, for all j's). By the correspondence (sub)test, with probability at least $1-\epsilon$, a uniformly chosen row in a uniformly chosen $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ is given the same values by $\Gamma_{\mathtt{c}}$ and by Γ (i.e., $\mathrm{row}_i(\Gamma_{\mathtt{c}}(\mathtt{m})) = \Gamma(\mathrm{row}_i(\mathtt{m}))$, for $i \in_{\mathrm{R}} [k]$). In other words, for uniformly chosen $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ and $i \in_{\mathrm{R}} [k]$

$$\mathrm{entry}_{i,j}(\Gamma_{\mathtt{c}}(\mathtt{m})) = \mathrm{col}^j(\Gamma(\mathrm{row}_i(\mathtt{m}))) \quad \text{for every } j \in [k].$$

On the other hand, by (3.1), with probability at least $1-\delta$, a uniformly chosen row in a uniformly chosen $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ is given the same values by $\Gamma_{\mathtt{c}}$ and by $\tau_{\mathtt{c}}$ (i.e., $\mathrm{entry}_{i,j}(\Gamma_{\mathtt{c}}(\mathtt{m})) = \tau_{\mathtt{c}}(\mathrm{entry}_{i,j}(\mathtt{m}))$, for $i \in_{\mathrm{R}} [k]$ and all $j \in [k]$). Thus, with probability at least $1-(\epsilon+\delta)$, a uniformly chosen row in a uniformly chosen $\mathtt{m} \in \mathcal{M}_{\mathtt{c}}$ is given the same values by Γ and by $\tau_{\mathtt{c}}$ (i.e., $\mathrm{col}^j(\Gamma(\mathrm{row}_i(\mathtt{m}))) = \tau_{\mathtt{c}}(\mathrm{entry}_{i,j}(\mathtt{m}))$, for $i \in_{\mathrm{R}} [k]$ and all $j \in [k]$). Using Axiom 3 (part 2—regarding $\mathcal{M}_{\mathtt{c}}$) and the "furthermore" part of Axiom 5, $\mathrm{row}_i(\mathtt{m})$ is uniformly distributed in $\mathbf{M}$ (for any $i \in [k]$ when $\mathtt{m} \in_{\mathrm{R}} \mathcal{M}_{\mathtt{c}}$). Part (c) of the claim follows (i.e., $\mathrm{col}^j(\Gamma(m)) = \tau_{\mathtt{c}}(\mathrm{col}^j(m))$, with high probability for $m \in_{\mathrm{R}} \mathbf{M}$ and all $j \in [k]$). A similar argument holds for part (r). ☐

A perfect (for columns) matrix "forces" *all* of its columns to satisfy some property Π (specifically, the value assigned by $\tau_{\mathtt{c}}$ to its column-sequences must match the value Γ of the matrix). Recall that we have just shown that almost all matrices are perfect and thus force all their columns to satisfy some property Π. Using a counting argument, one can show that all but at most a $\frac{1}{k}$ fraction of the column-sequences must satisfy Π in *almost all* matrices in which they appear.

Definition 3.1.3 (typical sequences). *Let $\delta_2 \stackrel{\mathrm{def}}{=} 2\sqrt{\delta_1}$. We say that the column-sequence $\bar{c}$ is* typical *if*

$$\mathrm{Prob}_{j,m}(\mathrm{col}^j(\Gamma(m)) = \tau_{\mathtt{c}}(\bar{c})) \geq 1 - \delta_2,$$

where $j \in_{\mathrm{R}} [k]$ and $m \in_{\mathrm{R}} \mathbf{M}^j(\bar{c})$. Otherwise, we say that $\bar{c}$ is nontypical. *Similarly, we say that the row-sequence $\bar{r}$ is* typical *if* $\mathrm{Prob}_{i,m}(\mathrm{row}_i(\Gamma(m)) = \tau_{\mathtt{r}}(\bar{r})) \geq 1 - \delta_2$, *where $i \in_{\mathrm{R}} [k]$ and $m \in_{\mathrm{R}} \mathbf{M}^i(\bar{r})$.*

Claim 3.1.4. *All but at most an $\frac{\delta_2}{2k}$ fraction of the column-sequence* (resp., row-sequences) *are typical.*

We will only use the bound for the fraction of typical row-sequences.

Proof. We mimic part of the counting argument of Claim 2.1.16. Let N be a set of nontypical row-sequences, containing exactly $\frac{\delta_2}{2k} \cdot |\mathbf{R}|$ sequences. Fix any $\bar{r} \in N$

and consider the set of matrices containing $\bar{r}$. By Axiom 3 (part 3—regarding $\mathbf{M}$), at most a $\frac{\delta_2}{2}$ fraction of these matrices contain some other row in N. On the other hand, by definition (of nontypical row-sequence), at least a δ_2 fraction of the matrices containing $\bar{r}$ have Γ disagree with $\tau_{\mathtt{r}}(\bar{r})$ on $\bar{r}$ and thus are nonperfect (for rows). It follows that at least a $\frac{\delta_2}{2}$ fraction of the matrices containing $\bar{r}$ are nonperfect (for rows) and contain no other row in N. Combining the bounds obtained for all $\bar{r} \in N$, we get that at least a $\frac{\delta_2}{2k} \cdot k \cdot \frac{\delta_2}{2} = \delta_1$ fraction of the matrices are not perfect (for rows).[9] This contradicts Claim 3.1.2(r), and so the current claim follows (for row-sequences and similarly for column-sequences). $\square$

3.3.2. Deriving the conclusion of the lemma. We are now ready to derive the conclusion of the Lemma. Loosely speaking, we claim that the function τ, defined so that $\tau(e)$ is the value most frequently assigned by $\tau_{\mathtt{c}}$ to e, satisfies the claim of the lemma.

Definition 3.1.5 (the function τ). *Let $\tau_{\mathtt{c}}(\bar{c})_i$ denote the value assigned by $\tau_{\mathtt{c}}$ to the* ith *element of $\bar{c} \in \mathbf{C}$. Define*

$$q_e(v) \overset{\text{def}}{=} \text{Prob}_{i,\bar{c}}(\tau_{\mathtt{c}}(\bar{c})_i = v),$$

where $i \in_{\text{R}} [k]$ and $\bar{c} \in_{\text{R}} \mathbf{C}_i(e)$ (recall that $\mathbf{C}_i(e)$ denotes the set of column-sequences having e as the ith component). *We consider $\tau : S \mapsto V$ so that $\tau(e) \overset{\text{def}}{=} v$ if $q_e(v) = \max_u\{q_e(u)\}$, with ties broken arbitrarily.*

The proof that τ satisfies the claim of Lemma 3.1 is a simplified version of the proof of Lemma 2.1.[10] We assume, contrary to our claim, that for a uniformly chosen $m \in \mathbf{M}$

$$\text{Prob}_m\left(\exists i, j \text{ so that entry}_{i,j}(\Gamma(m)) \neq \tau(\text{entry}_{i,j}(m))\right) > \gamma. \tag{3.2}$$

As in the proof of Lemma 2.1, we define a notion of an *annoying* row-sequence. Using the above (contradiction) hypothesis, we first show that many row-sequences are annoying. Next, we show that lower bounds on the number of annoying row-sequences translate to lower bounds on the probability that a uniformly chosen matrix is nonperfect (for columns). This yields a contradiction to Claim 3.1.2(c).

Definition 3.1.6 (a new definition of annoying rows). *A row-sequence $\bar{r} = (r_1, \ldots, r_k)$ is said to be* annoying *if there exists a $j \in [k]$ so that the* jth *element in $\tau_{\mathtt{r}}(\bar{r})$ differs from $\tau(r_j)$.*

Using Claim 3.1.2(r), we get the following claim.

Claim 3.1.7. *Suppose that* (3.2) *holds and let $\gamma_1 \overset{\text{def}}{=} \gamma - \delta_1$. Then, at least a $\frac{\gamma_1}{k}$ fraction of the row-sequences are annoying.*

Proof. Combining (3.2) and Claim 3.1.2(r), we get that with probability at least $\gamma - \delta_1 = \gamma_1$, a uniformly chosen matrix $m \in \mathbf{M}$ is perfect for rows and contains some entry, denoted (i, j), for which the Γ value is different from the τ value (i.e., $\text{entry}_{i,j}(\Gamma(m)) \neq \tau(\text{entry}_{i,j}(m))$). Since the $\tau_{\mathtt{r}}$-value of each row of a perfect (for rows)

[9] For each $\bar{r} \in N$, let $M_{\bar{r}}$ denote the number of nonperfect matrices containing $\bar{r}$ but not any other row in N. Then, $M_{\bar{r}} \geq \frac{\delta_2}{2} \cdot \sum_{i=1}^{k} |\mathbf{M}_i(\bar{r})| = \frac{\delta_2}{2} \cdot k \cdot \frac{|\mathbf{M}|}{|\mathbf{R}|}$ and the number of nonperfect matrices is at least $\sum_{\bar{r} \in N} M_{\bar{r}} \geq \frac{\delta_2 |\mathbf{R}|}{2k} \cdot \frac{\delta_2 k |\mathbf{M}|}{2|\mathbf{R}|}$.

[10] The reader may wonder how it is possible that a simpler proof yields a stronger result, as the claim concerning the current τ is stronger. The answer is that the current τ is defined based on a more restricted function over $\mathbf{C}$ and there are also stronger restrictions on Γ. Both restrictions are due to facts that we have inferred using Lemma 2.1 with regard to (w.r.t) $\Gamma_{\mathtt{c}}$ and $\Gamma_{\mathtt{r}}$.

matrix m matches the Γ values, it follows that the ith row of m is annoying. Thus, at least a γ_1 fraction of the matrices contain an annoying row-sequence. Using Axiom 3 (part 2—regarding $\mathbf{M}$), we conclude that the fraction of annoying row-sequences must be as claimed. ◻

A key observation is that each row-sequence that is both typical and annoying yields many matrices which are nonperfect for columns.

Claim 3.1.8. *Suppose that a row-sequence $\bar{r}$ is both typical and annoying. Then, at least a $\frac{1}{2}-\delta_2$ fraction of the matrices, containing the row-sequence $\bar{r}$, are nonperfect for columns.*

Proof. Since $\bar{r}=(r_1,\ldots,r_k)$ is annoying, there exists a $j\in[k]$ so that the jth component of $\tau_{\mathtt{r}}(\bar{r})$ (which is the value assigned to r_j) is different from $\tau(r_j)$. Let us denote by v the value $\tau_{\mathtt{r}}(\bar{r})$ assigns to r_j. Note that $v\neq\tau(r_j)$. Consider the probability space defined by uniformly selecting $i\in[k]$ and $m\in\mathbf{M}_i(\bar{r})$. Since $\bar{r}$ is typical it follows that in almost all of these matrices the value assigned to r_j by the Γ equals v; namely,

$$\mathrm{Prob}_{i,m}(\mathrm{entry}_{i,j}(\Gamma(m))=v) \;\; \geq \;\; 1-\delta_2. \tag{3.3}$$

By Axiom 4 (regarding $\mathbf{M}$), the jth column of m is uniformly distributed in $\mathbf{C}_i(r_j)$. Now, using the definition of the function τ and the accompanying notations, we get

$$\mathrm{Prob}_{i,m}(\tau_{\mathtt{c}}(\mathrm{col}^j(m))_i=v) \;\; = \;\; q_{r_j}(v) \;\; \leq \;\; \frac{1}{2}. \tag{3.4}$$

The inequality holds since $v\neq\tau(r_j)$ and by τ's definition $q_{r_j}(v)\leq q_{r_j}(\tau(r_j))$. Combining (3.3) and (3.4), we get

$$\mathrm{Prob}_{i,m}(\mathrm{entry}_{i,j}(\Gamma(m))\neq\tau_{\mathtt{c}}(\mathrm{col}^j(m))_i) \;\; \geq \;\; \frac{1}{2}-\delta_2,$$

and the claim follows. ◻

Combining Claims 3.1.7, 3.1.4, and 3.1.8, we get a lower bound on the number of matrices which are nonperfect for columns.

Claim 3.1.9. *Suppose that* (3.2) *holds and let $\gamma_2\stackrel{\mathrm{def}}{=}\gamma_1-\frac{\delta_2}{2}$. Then, at least a $\frac{\gamma_2}{3}$ fraction of the matrices are nonperfect for columns.*

Proof. By Claims 3.1.7 and 3.1.4, at least a $\frac{\gamma_1}{k}-\frac{\delta_2}{2k}$ $(=\frac{\gamma_2}{k})$ fraction of the row-sequences are both annoying and typical. Let us consider a set of exactly $\frac{\gamma_2}{k}\cdot|\mathbf{R}|$ such row-sequences, denoted A. Mimicking again the counting argument part of Claim 2.1.16, we bound, for each $\bar{r}\in A$, the fraction of nonperfect (for columns) matrices which contain $\bar{r}$ but no other row-sequence in A. Using an adequate setting of δ_2 and γ_2, this fraction is at least $\frac{1}{3}$. Summing the bounds achieved for all $\bar{r}\in A$, the claim follows. ◻

Using a suitable choice of γ (as a function of ϵ), Claim 3.1.9 contradicts Claim 3.1.2(c), and so (3.2) cannot hold. The lemma follows. ◻

3.4. Application to low-degree testing. Again, the set of all k-by-k-by-k arrays over S satisfies Axioms 1–5. A more useful set of 3-dimensional arrays is defined as follows.

Construction 3.2 (main construction). *Let $\mathbf{M}$ be as in the basic construction (i.e., Construction* 2.2*). We let $\mathcal{M}_{\mathtt{c}}=\mathcal{M}_{\mathtt{r}}$ be the set of matrices defined by applying the basic construction to the element-set $\mathbf{C}=\mathbf{R}$. Specifically, a matrix in $\mathcal{M}_{\mathtt{c}}$ is defined by the quadruple (x,y,x',y'), where each of the four elements is a pair over S,* *so that the* (i,j)th *entry in the matrix equals* $(x+jy)+i(x'+jy')$. *Here* x,y,x', and y' *are viewed as* 2*-dimensional vectors over the finite field* S *and* i,j *are scalars in* S. *The* (i,j)th *entry is a pair over* S *which represents a pairwise independent sequence* (*which equals an element in* $\mathbf{C}=\mathbf{R}$).

PROPOSITION 3.2. *Construction* 3.2 *satisfies Assumptions* 1–5.

Combining all the above with the low-degree test of [GLRSW, RS96] and using the results proved there,[11] we get a low-degree test which is sufficiently efficient to be used in the proof of the PCP-characterization of NP.

CONSTRUCTION 3.3 (low-degree test). *Let* $f : F^n \mapsto F$, *where* F *is a field of prime cardinality, and* d *be an integer so that* $|F| > 4(d+2)^2$. *Let* $\mathbf{M}$, $\mathcal{M}_{\mathtt{c}}$, *and* $\mathcal{M}_{\mathtt{r}}$ *be as in Construction* 3.2, *with* $S = F^n$, $V = F$ *and* $k \stackrel{\text{def}}{=} 4(d+2)^2$. *Let* $\Gamma : \mathbf{M} \mapsto F^{k\times k}$, $\Gamma_{\mathtt{r}} : \mathcal{M}_{\mathtt{r}} \mapsto F^{k^3}$, *and* $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto F^{k^3}$ *be auxiliary tables* (which should contain the corresponding f-values). *The low-degree test consists of the following three steps:*

1. *Apply the extended consistency test* (i.e., Construction 3.1) *to the functions* $\Gamma : \mathbf{M} \mapsto F^{k\times k}$, $\Gamma_{\mathtt{r}} : \mathcal{M}_{\mathtt{r}} \mapsto F^{k^3}$, *and* $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto F^{k^3}$.
2. *Select uniformly a matrix* $m \in \mathbf{M}$ *and test whether the* polynomial interpolation condition (cf. membership test of [GLRSW, p. 37]) *holds for each row; namely, we test that*

$$\sum_{j=1}^{d+2} \alpha_j \cdot \text{entry}_{i,j}(\Gamma(m)) = 0$$

   for all $i \in [k]$, *where* $\alpha_j = (-1)^j \cdot \binom{d+1}{j-1}$.
3. *Select uniformly a matrix in* $\mathbf{M}$ *and test whether* Γ *and* f *agree on a uniformly chosen element in the matrix. Namely, select uniformly* $m \in \mathbf{M}$ *and* $i,j \in [k]$, *and check whether* $\text{entry}_{i,j}(\Gamma(m)) = f(\text{entry}_{i,j}(m))$.

The test accepts if and only if all the above three subtests accept.

PROPOSITION 3.3. *Let* $f : F^n \mapsto F$, *where* F *is a field, and let* $\ell \stackrel{\text{def}}{=} n \cdot \log_2 |F|$. *Then, the low-degree test of Construction* 3.3 *satisfies the following conditions:*

Efficiency. *The test runs in* poly(ℓ)*-time, uses* $O(\ell)$ *random bits, and makes a constant number of queries each of length* $O(\ell)$. (*The queries are answered by strings of length* poly(ℓ).)

Completeness. *If* f *is a degree-*d *polynomial, then there exist* $\Gamma : \mathbf{M} \mapsto F^{k\times k}$, $\Gamma_{\mathtt{r}} : \mathcal{M}_{\mathtt{r}} \mapsto F^{k^3}$, *and* $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto F^{k^3}$ *so that the test always accepts.*

Soundness. *For every* $\delta > 3/(d+2)^2$ *there exists an* $\epsilon > 0$ *so that for every* f *which is at distance at least* δ *from any degree-*d *polynomial and for every* $\Gamma : \mathbf{M} \mapsto F^{k\times k}$, $\Gamma_{\mathtt{r}} : \mathcal{M}_{\mathtt{r}} \mapsto F^{k^3}$, *and* $\Gamma_{\mathtt{c}} : \mathcal{M}_{\mathtt{c}} \mapsto F^{k^3}$, *the test rejects with probability at least* ϵ. *Furthermore, the constant* ϵ *is a polynomial in* δ *which does not depend on* n, d, *and* F.

As a corollary, we get Lemma 1.3.

Proof. The efficiency requirement is immediate from the construction. Also, as usual, the completeness requirement is easy to establish. We thus turn to the soundness requirement. By Proposition 3.2, we may apply Lemma 3.1 to the first subtest and infer that either the first subtest fails with some constant probability (say ϵ_1) or there exists a function $\tau : F^n \mapsto F$ so that, with very high constant probability (say,

[11] Rather than using much stronger results obtained via a more complicated analysis, as in [ALMSS], which rely on the lemma of [AS].

$1 - \delta_1$),

$$\text{entry}_{i,j}(\Gamma(m)) = \tau(\text{entry}_{i,j}(m)) \tag{3.5}$$

holds for all $i \in [k]$ and $j \in [d+2]$. We assume from this point on that this is the case (or else the low-degree test rejects with probability at least ϵ_1). Now, by [GLRSW] (see also [Sud, Thm. 3.3] and [RS96, Thm. 5]), either

$$\text{Prob}_{x,y \in F^n}\left(\sum_{j=1}^{d+2} \alpha_j \cdot \tau(x+jy) \neq 0\right) > \frac{1}{2(d+2)^2} \tag{3.6}$$

or τ is very close (specifically at distance at most $1/(d+2)^2$) to some degree-d polynomial. A key observation is that the main construction (i.e., Construction 3.2) has the property that rows in $m \in_{\text{R}} \mathbf{M}$ are distributed identically to the distribution in (3.6). Thus, for every $i \in [k]$, either

$$\text{Prob}_{m \in \mathbf{M}}\left(\sum_{j=1}^{d+2} \alpha_j \cdot \tau(\text{entry}_{i,j}(m)) \neq 0\right) > \frac{1}{2(d+2)^2} \tag{3.7}$$

or τ is at distance at most $\delta_2 \stackrel{\text{def}}{=} 1/(d+2)^2$ from some degree-d polynomial. Now, we claim that in case (3.7) holds, the second subtest will reject with constant probability. The claim is proven by considering $k = 4(d+2)^2$ pairwise-independent copies of the GLRSW test (i.e., the test in (3.7)), and recalling that the rows in $m \in_{\text{R}} \mathbf{M}$ are distributed in a pairwise-independent manner. Using Chebyshev's inequality and the hypothesis that each copy rejects with probability at least $1/2(d+2)^2$, we conclude that the probability that none of these copies rejects is bounded above by $\frac{2(d+2)^2}{4(d+2)^2} = \frac{1}{2}$. Thus, the second subtest must reject with probability at least $\epsilon_2 \stackrel{\text{def}}{=} \frac{1}{2} - \delta_1$, where δ_1 accounts for the substitution of the τ values by the entries in $\Gamma(\cdot)$. We conclude that τ must be δ_2-close to a degree-d polynomial or else the test rejects with probability at least ϵ_2.

Next, we claim that if f disagrees with τ on a $\delta_3 > \delta_1$ fraction of the inputs then the third subtest rejects with probability at least $\epsilon_3 \stackrel{\text{def}}{=} \delta_3 - \delta_1$ (since the disagreement of f and τ is upper bounded by the sum of the disagreement of f and Γ and the disagreement of Γ and τ).

Thus, if the low-degree test rejects with probability smaller than $\epsilon = \min\{\epsilon_1, \epsilon_2, \epsilon_3\}$, then f disagrees with τ on at most δ_3 fraction of the inputs, where τ is δ_2-close to a degree d-polynomial. (So f is $(\delta_2 + \delta_3)$-close to a degree d-polynomial.) The proposition follows using arithmetic: specifically, we set $\delta_1 = \delta/3$, $\delta_3 = 2\delta/3$, $\epsilon_1 = \text{poly}(\delta_1)$ (where the polynomial is as in Lemma 3.1), and verify that $\delta_3 + \delta_2 \leq \delta$ (since $\delta_2 = (d+2)^{-2} < \delta/3$). Furthermore, $\epsilon = \min\{\epsilon_1, \epsilon_2, \epsilon_3\} = \text{poly}(\delta)$ (since $\epsilon_2 = 0.5 - \delta_1 \geq 0.5 - (1/3) = 1/6$ and $\epsilon_3 = \delta_3 - \delta_1 = \delta/3$). ☐

4. Proof of Lemma 1.1.

There should be an easier and more direct way of proving Lemma 1.1. However, having proved Lemma 2.1, we can apply it[12] to derive a short proof of Lemma 1.1. To this end we view ℓ-multisets over S as k-by-k matrices, where $k = \sqrt{\ell}$. Recall that the resulting set of matrices satisfies Axioms 1–4. Thus,

[12] This is indeed an overkill. For example, we can avoid all complications regarding shifts (in the proof of Lemma 2.1).

by Lemma 2.1 (applied to $\Gamma = F$), in case the test accepts with probability at least $1-\epsilon$, there exists a function $f : S \mapsto V$ such that

$$\mathrm{Prob}_{A\in_{\mathrm{R}} S^k, B\in_{\mathrm{R}} E_{k^2}(A)}(\forall e \in A,\ F(B)_e = f(e)) \geq 1-\delta,$$

where S^k is the set of all k-multisets over S and $E_l(A)$ is the set of all l-multisets extending A (and $F(B)_e$ denotes the value assigned by F to $e \in B$). We can think of this probability space as first selecting $B \in_{\mathrm{R}} S^{k^2}$ and next selecting a k-subset A in B. Thus,

$$\mathrm{Prob}_{B\in_{\mathrm{R}} S^{k^2}, A\in_{\mathrm{R}} C_k(B)}(\exists e \in A \text{ s.t. } F(B)_e \neq f(e)) \leq \delta, \tag{4.1}$$

where $C_k(B)$ denotes the set of all k-multisets contained in B. This implies

$$\mathrm{Prob}_{B\in_{\mathrm{R}} S^{k^2}}(|\{e \in B : F(B)_e \neq f(e)\}| > k) \leq 2\delta,$$

as otherwise (4.1) is violated. (The probability that a random k-subset hits a subset of density $\frac{1}{k}$ is at least $\frac{1}{2}$.) The lemma follows. $\square$

Comment. A previous version of this paper [GS96] has stated a stronger version of Lemma 1.1, where the sequences $F(x_1,\ldots,x_\ell)$ and $(f(x_1),\ldots,f(x_\ell))$ are claimed to be identical (rather than different on at most k locations), for a $1-\delta$ fraction of all possible $(x_1,\ldots,x_\ell) \in S^\ell$. Unfortunately, the proof given there was not correct— a mistake in the concluding lines of the proof of Claim 4.2.9 was found by Sudan [Sud]. Still we conjecture that the stronger version holds as well, and that it can be established by a test which examines two random $(2k-1)$-extensions of a random k-subset.

Acknowledgment. We are grateful to Madhu Sudan for pointing out an error in an earlier version, and for other helpful comments. We also thank the anonymous referees for their useful comments.

REFERENCES

[ALMSS] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and the hardness of approximation problems*, J. ACM, 45 (1998), pp. 501–555.

[AS] S. Arora and S. Safra, *Probabilistic checkable proofs: A new characterization of NP*, J. ACM, 45 (1998), pp. 70–122.

[B94] L. Babai, *Transparent Proofs and Limits to Approximation*, Technical report TR-94-07, Dept. of Computer Science, University of Chicago, Chicago, IL, 1994.

[BFL] L. Babai, L. Fortnow, and C. Lund, *Nondeterministic exponential time has two-prover interactive protocols*, Comput. Complexity, 1 (1991), pp. 3–40.

[BFLS] L. Babai, L. Fortnow, L. Levin, and M. Szegedy, *Checking computations in polylogarithmic time*, in 23rd ACM Symposium on the Theory of Computing, New Orleans, LA, 1991, pp. 21–31.

[BF] D. Beaver and J. Feigenbaum, *Hiding instances in multioracle queries*, in 7th Symposium on Theoretical Aspects of Computer Science, Rouen, France, Lecture Notes in Comput. Sci. 415, Springer-Verlag, New York, 1990, pp. 37–48.

[BGS] M. Bellare, O. Goldreich, and M. Sudan, *Free bits, PCPs, and non-approximability – Towards tight results*, SIAM J. Comput., 27 (1998), pp. 804–915.

[BGLR] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, *Efficient probabilistically checkable proofs and applications to approximation*, in 25th ACM Symposium on the Theory of Computing, San Diego, CA, 1993, pp. 294–304.

[BS] M. Bellare and M. Sudan, *Improved non-approximability results*, in 26th ACM Symposium on the Theory of Computing, Montreal, Canada, 1994, pp. 184–193.

[BGKW] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson, *Multi-prover interactive proofs: How to remove intractability*, in 20th ACM Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 113–131.

[BLR] M. Blum, M. Luby, and R. Rubinfeld, *Self-testing/correcting with applications to numerical problems*, J. Comput. System Sci., 47 (1993), pp. 549–545.

[FGLSS] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy, *Approximating Clique is almost NP-complete*, J. ACM, 43 (1996), pp. 268–292.

[FRS] L. Fortnow, J. Rompel, and M. Sipser, *On the power of multi-prover interactive protocols*, in Proceedings 3rd IEEE Symp. on Structure in Complexity Theory, Georgetown University, Washington, D.C., 1988, pp. 156–161.

[FS] K. Friedl and M. Sudan, *Some improvement to total degree tests*, in Proceedings 3rd Israel Symp. on Theory of Computing and Systems, Tel Aviv, Israel, 1995, pp. 190–198.

[GLRSW] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson, *Self-testing/correcting for polynomials and for approximate functions*, in 23th ACM Symposium on the Theory of Computing, New Orleans, LA, 1991, pp. 32–42.

[G97] O. Goldreich, *A taxonomy of proof systems*, in Complexity Theory Retrospective II, L.A. Hemaspaandra and A. Selman, eds., Springer-Verlag, New York, 1997, pp. 109–134.

[GS96] O. Goldreich and M. Safra, *A Combinatorial Consistency Lemma with Application to Proving the PCP Theorem*, Technical report ECCC TR96-047, Version 1, 1996.

[GMR] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof systems*, SIAM J. Comput., 18 (1989), pp. 186–208.

[H96] J. Hastad, *Clique is hard to approximate within* $n^{1-\epsilon}$, in 37th IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996.

[LS] D. Lapidot and A. Shamir, *Fully parallelized multi prover protocols for NEXP-time*, in 32nd IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1991, pp. 13–18.

[LFKN] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, J. ACM, 39 (1992), pp. 859–868.

[RaSa] R. Raz and S. Safra, *A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP*, in 29th ACM Symposium on the Theory of Computing, El Paso, TX, 1997, pp. 475–484.

[RS92] R. Rubinfeld and M. Sudan, *Testing polynomial functions efficiently and over rational domains*, in 3rd ACM–SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 23–32.

[RS96] R. Rubinfeld and M. Sudan, *Robust characterization of polynomials with application to program testing*, SIAM J. Comput., 25 (1996), 252–271.

[Sha] A. Shamir, *IP=PSPACE*, J. ACM, 39 (1992), pp. 869–877.

[Sud] M. Sudan, *Efficient checking of polynomials and proofs and the hardness of approximation problems*, ACM Distinguished Theses, Lecture Notes in Comput. Sci. 1001, Springer-Verlag, New York, 1995.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1155–1163

VERIFICATION OF IDENTITIES[*]

SRIDHAR RAJAGOPALAN[†] AND LEONARD J. SCHULMAN[‡]

Abstract. We provide an $O(n^2 \log \frac{1}{\delta})$ time randomized algorithm to check whether a given operation $\circ : S \times S \to S$ is associative (where $n = |S|$ and $\delta > 0$ is the error probability required of the algorithm). We prove that (for any constant δ) this performance is optimal up to a constant factor, even if the operation is "cancellative." No sub-n^3 time algorithm was previously known for this task.

More generally we give an $O(n^c)$ time randomized algorithm to check whether a collection of c-ary operations satisfy any given "read-once" identity.

Key words. computer aided verification, randomized algorithms

AMS subject classifications. Primary, 68Q25; Secondary, 20N02, 20N05

PII. S0097539797325387

1. Introduction.

Let a set S be given, along with a binary operation $\circ : S \times S \to S$. In this paper, we consider the complexity of checking whether $\circ$ is associative. We provide an algorithm for this problem and then show how our method extends without essential modification to provide a means of checking general "read-once" identities.

Throughout this paper, let $n = |S|$. We provide a randomized, one-sided error algorithm which in time $O(n^2 \log \frac{1}{\delta})$ computes whether the operation $\circ$ is associative. If $\circ$ is associative, then the algorithm does not err in its response, while if $\circ$ is not associative, then the error probability is bounded by δ. It is assumed that the operation $\circ$ is computable in unit time; the same assumption is made throughout the paper, and all runtimes scale linearly in the actual time required for computation of the given operations.

The techniques we develop are more general and can be used to check whether collections of operators satisfy various identities. For instance, given a finite state machine, $\delta : S \times \Sigma \to S$, verify that the machine is "asynchronous," namely $\delta(\delta(s, a), b) = \delta(\delta(s, b), a)\ \forall\ s \in S$ and $a, b \in \Sigma$. This test is sometimes called the diamond test.

In case the identity is not satisfied, our method also provides a witness (e.g., for associativity, a triple a, b, c such that $(a \circ b) \circ c \neq a \circ (b \circ c)$) with only a logarithmic slowdown in the time complexity.

Associativity: Prior work. Prior to our work, except in special cases, no method for verifying associativity was known that was better than the naive $O(n^3)$-time algorithm of examining whether $a \circ (b \circ c) = (a \circ b) \circ c\ \forall\ a, b, c \in S$.

F. W. Light observed in 1949 (see [3]) that, if $R \subseteq S$ is a set of generators of S, i.e., a collection such that every element of S is representable as a product of elements of R, then it suffices to test all triples a, b, c in which b is an element of R. This, however, does not result in a sub-n^3 algorithm for this problem, for two reasons: first, the operation may require a large set of generators, as much as n (and there are such

[*]Received by the editors August 4, 1997; accepted for publication (in revised form) March 10, 1999; published electronically February 1, 2000.
http://www.siam.org/journals/sicomp/29-4/32538.html

[†]IBM Almaden Research Center, San Jose, CA (sridhar@almaden.ibm.com). The research of this author was conducted while at a postdoctoral fellowship at DIMACS and Princeton University.

[‡]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280 (schulman@cc.gatech.edu).

examples which are associative, so the algorithm would in fact examine all triples)—see Example 1; second, even if the operation does have a small set of generators, we are not aware of any rapid method for obtaining a set of generators of (even close to) minimal size.

Associativity: Cancellative operations. An operation $\circ$ on a finite set S is left (resp., right) cancellative if for every a and b, there is an x (which must be unique) such that $x \circ a = b$ (resp., $a \circ x = b$). A cancellative operation is both left and right cancellative. In other words, every row and column of the table for the operation is a permutation of S.

In the special case that the operation is cancellative, Light's observation is quite useful. Only $O(n^2)$ time is required to test whether $\circ$ is cancellative. Moreover, we will show in section 5 that, given a cancellative multiplication table, one can deterministically compute a set of generators of size $\lfloor \log_2 n \rfloor + 1$ in $O(n^2)$ time. Thus, Light's observation results in an $O(n^2 \log n)$ deterministic algorithm for verifying associativity in the cancellative case. Consequently there is a deterministic $O(n^2 \log n)$ time algorithm to test whether a given set and operation form a group. (Alternatively, a random set of elements may be chosen; a set of $c \log_2 n$ elements will be a set of generators with probability at least $1 - \exp(-c)$. This implies a randomized algorithm analogous to the deterministic one.)

We show in section 5 that any cancellative nonassociative operation has at least $n-2$ nonassociative triples. Therefore, the procedure of checking $O(n^2 \log \frac{1}{\delta})$ random triples will succeed with probability $1 - \delta$, providing essentially the same guarantee as the randomized version of Light's method, while being even simpler. (Both run in time $O(n^2 \log \frac{1}{\delta})$.)

However, just as Light's observation fails to be of use for general operations, because small sets of generators may not exist or may be hard to find, the random sampling approach also fails to be of use for general operations, in this case because for every $n \geq 3$ there exists an operation (noncancellative, of course) with just one nonassociative triple. See Example 2.

Associativity: Lower bound. We show that any randomized algorithm requires time $\Omega(n^2)$ to verify associativity, even in the cancellative case. Thus the runtime of our randomized algorithm as a function of input size is tight up to a constant factor.

Comment on terminology. The pair $(S, \circ)$ is known in the algebra literature as a groupoid [1, 2, 3], a term which is unfortunately also used elsewhere in that literature to mean something entirely different [5, 7]. When $\circ$ is cancellative $(S, \circ)$ is referred to in [1, 2] as a quasi group.

2. Algorithm for checking associativity.

We define the structure $\mathcal{S}/2 = (\mathbb{Z}/2)[S]$ as follows. The elements of $\mathcal{S}/2$ are sums of elements of S, with coefficients in $\mathbb{Z}/2$ (i.e., $\sum_{s \in S} \alpha_s s$ for $\alpha_s \in \mathbb{Z}/2$). $\mathcal{S}/2$ is equipped with the following operations:

(1) Addition: $\sum_s \alpha_s s + \sum_s \beta_s s = \sum_s (\alpha_s + \beta_s) s$.

(2) Scalar multiplication: $\beta \sum_s \alpha_s s = \sum_s (\beta \alpha_s) s$ and $(\sum_s \alpha_s s)\beta = \sum_s (\beta \alpha_s) s$ for $\beta \in \mathbb{Z}/2$.

(3) The operation $\circ$: $(\sum_s \alpha_s s) \circ (\sum_s \beta_s s) = \sum_r \sum_s \alpha_r \beta_s (r \circ s)$.

Thus $\mathcal{S}/2$ is what would be known as the "group algebra" of $(S, \circ)$ over $\mathbb{Z}/2$, were $(S, \circ)$ a group. (In the present circumstance "groupoid algebra" may be an appropriate term.)

Henceforth, we will denote members of $\mathcal{S}/2$ by bold letters, and coefficients by the corresponding Greek lowercase, e.g., $\mathbf{a} = \sum_s \alpha_s s$.

Our method for checking associativity in $(S, \circ)$ is to repeat the following, $O(\log \frac{1}{\delta})$ times:

- *Check the associative identity for three random elements of* $\mathcal{S}/2$.

In other words, select random (uniformly i.i.d.) $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{S}/2$, and check that $(\mathbf{a} \circ \mathbf{b}) \circ \mathbf{c} = \mathbf{a} \circ (\mathbf{b} \circ \mathbf{c})$.

THEOREM 2.1. *The algorithm above, running in time $O(n^2 \log \frac{1}{\delta})$, will determine whether $\circ$ is associative, with error probability at most δ if it is not associative and no error if it is associative.*

First note that checking that $(\mathbf{a} \circ \mathbf{b}) \circ \mathbf{c} = \mathbf{a} \circ (\mathbf{b} \circ \mathbf{c})$ can be done in time $O(n^2)$ because $\mathbf{a} \circ \mathbf{b}$ can be computed by brute force in time $O(n^2)$; moreover, the result of this computation is another vector $\mathbf{d} \in \mathcal{S}/2$, so the subsequent computation $\mathbf{d} \circ \mathbf{c}$ can again be computed by brute force in time $O(n^2)$. For the same reason, $\mathbf{a} \circ (\mathbf{b} \circ \mathbf{c})$ can be computed in time $O(n^2)$; and finally, the two sides can be compared in time $O(n)$.

We now show that a single run of the above process succeeds in detecting nonassociativity with probability at least $1/8$; by repeating the process $\log_{8/7} \frac{1}{\delta}$ times, the dependence of the running time on δ follows.

It is easily verified that $\circ$ is associative on S if and only if it is associative on $\mathcal{S}/2$ as well (for a proof see Lemma 3.2).

Let S^k denote the k-wise Cartesian product of S with itself. A *minor* of S^k is a set $A_1 \times A_2 \times \cdots \times A_k \subseteq S^k$ (where each A_i is a subset of S). If H is a commutative group (expressed additively), g is a function $g : S^k \to H$, and T is a subset of S^k, then define $g(T) = \sum_{t \in T} g(t)$.

LEMMA 2.2. *Let H be any commutative group, and let $g : S^k \to H$ be a nonzero function. Then the fraction of minors T of S^k for which $g(T) \neq 0$ is at least 2^{-k}.*

Proof. Fix $\tau = (t_1, \ldots, t_k)$ such that $g(\tau) \neq 0$. Let $T_1 \subseteq S - \{t_1\}, \ldots, T_k \subseteq S - \{t_k\}$. For $1 \leq i \leq k$ define $T_i^0 = T_i$ and $T_i^1 = T_i \cup \{t_i\}$. Observe that $\{\tau\} = \cap_1^k((T_1^1 \times \cdots \times T_k^1) - (T_1^1 \times \cdots \times T_{i-1}^1 \times T_i^0 \times T_{i+1}^1 \times \cdots \times T_k^1))$ and that for $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_k)$ (with $\varepsilon_i \in \{0,1\}\ \forall i$), $T_1^{\varepsilon_1} \times \cdots \times T_k^{\varepsilon_k} = \bigcap_i T_1^1 \times \cdots \times T_{i-1}^1 \times T_i^{\varepsilon_i} \times T_{i+1}^1 \times \cdots \times T_k^1$. Hence inclusion-exclusion gives

$$g(\tau) = (-1)^k \sum_{\varepsilon} (-1)^{\sum \varepsilon_i} g(T_1^{\varepsilon_1} \times \cdots \times T_k^{\varepsilon_k}).$$

Since $g(\tau) \neq 0$, there exists a minor $T_1^{\varepsilon_1} \times \cdots \times T_k^{\varepsilon_k}$ such that $g(T_1^{\varepsilon_1} \times \cdots \times T_k^{\varepsilon_k}) \neq 0$.

Thus the minors of S^k are partitioned into sets of 2^k minors, each identified by some $T_1, \ldots, T_k$, and in each such set there is at least one minor T for which $g(T)$ is nonzero. $\square$

The error probability claimed in the theorem follows from the lemma by defining

$$g : S^3 \to \mathcal{S}/2,$$

$$g(i,j,k) = (i \circ j) \circ k - i \circ (j \circ k).$$

In the last line we have abbreviated notation by using the natural embedding $S \to \mathcal{S}/2$ sending $s \in S$ to $1s + \sum_{r \neq s} 0r$.

Remark. There is a similarity to the argument showing that Freivalds's [4] checker for matrix multiplication succeeds with probability at least 1/2. In that case, given matrices A, B, and C, a random vector v is multiplied by $AB - C$; if $AB \neq C$, then there is some vector u such that $u(AB - C) \neq 0$, and with every vector w such that $w(AB - C) = 0$, we associate the vector $w + u$ and note that $(w + u)(AB - C) \neq 0$.

3. Generalizations.

We have presented our method in the context of testing for associativity, that being a concrete and significant case, and the original problem we considered. However the method can be extended *mutatis mutandis* to more general situations.

Let $\{D_i\}_{i=1}^{c}$ be a collection of finite sets. If $\circ$ is a function on domain $\prod_{i=1}^{c} D_i$, we say that c is the *degree* of $\circ$. An *identity* is an equation involving one or more operations $\circ_j$, and which is required to hold for all instantiations of the variables, e.g., $\forall a, b, d, e \in D_1, c \in D_2$,

$$\circ_1(\circ_2(a,b), c, \circ_2(d,e)) = \circ_1(\circ_2(e,d), c, \circ_2(b,a)),$$

or, in the case of associativity,

$$\circ(\circ(a,b),c) = \circ(a, \circ(b,c)) \ \ \forall a,b,c \in S.$$

Note that each side of the equation may be viewed as a formula (in the circuit sense); we let ℓ be the total number of operations (internal nodes) occurring in the formulas. Thus $\ell = 6$ in the first example, and $\ell = 4$ for associativity. We let k be the total number of distinct variables occurring in the identity; thus $k = 5$ in the first example, and $k = 3$ for associativity.

A "read-once" identity is one in which every variable occurs exactly once on each side of the equation.

We have the following theorem.

THEOREM 3.1. *Let $\{D_i\}_{i=1}^{k}$ be finite sets, and let $\{\circ_j\}$ be a collection of operators, defined on various products of these sets; let the arguments to operator $\circ_j$ be drawn from the sets $D_{\sigma(j,1)}, \ldots, D_{\sigma(j,c_j)}$ where c_j is the degree of $\circ_j$. Let $M = \max_j \prod_{i=1}^{c_j} |D_{\sigma(j,i)}|$. A read-once identity with ℓ operations on k variables $\{x_i\}_{i=1}^{k}$ (each ranging in the corresponding D_i) can be verified in time $O(M \log(1/\delta) 2^k \ell)$ with failure probability at most δ.*

For comparison, supposing for simplicity that all the sets are of size n, this runtime is $O(n^{\max\{c_j\}} \log(1/\delta) 2^k \ell)$ whereas the naive method requires time $O(\ell n^k)$. The key gain is that the exponent of n is the maximum degree of the operations, rather than the number of variables in the identity.

This theorem relies upon the following.

LEMMA 3.2. *A read-once identity holds in S if and only if it holds in $\mathcal{S}/2$.*

Proof. If the identity holds in $\mathcal{S}/2$, then in particular it holds when each variable in the equation is a "singleton" element of the algebra, i.e., an element which has coefficient 1 on some element of S, and coefficient 0 elsewhere.

For the converse, note that each side of the identity can be expanded as a summation, over $(x_1, \ldots, x_k) \in D_1 \times \cdots \times D_k$, of terms each in exactly the same form as that side of the identity. Each corresponding pair of terms are equal in S, hence the summations are equal. ∎

Some examples:

Multiple domains. Let a finite state machine (S, Σ, δ) be given. Here S is the set of states of the machine (let $|S| = n$), Σ is the tape alphabet (let $|\Sigma| = m$), and $\delta : S \times \Sigma \to S$ is the transition function. Then the degree of the operator δ is 2, and it is possible to check in time $O(nm)$ (rather than the obvious $O(nm^2)$) that the operation of the machine is not dependent on the order of the input. (In other words, whether $\forall s \in S, a, b \in \Sigma, \delta(\delta(s,a),b) = \delta(\delta(s,b),a)$.) Testing for this property of the machine is sometimes called the diamond test in the CAD literature. We do this by extending $\mathbb{Z}/2$ with both S and Σ. Then we can extend δ implicitly to $\delta : (\mathbb{Z}/2)[S] \times (\mathbb{Z}/2)[\Sigma] \to (\mathbb{Z}/2)[S]$. We then verify the identity for a random $\mathbf{s}, \mathbf{a}$, and $\mathbf{b}$.

More generally, if $\circ$ is an operator from $X \times Y \to Z$, it can be naturally extended to an operator from $(\mathbb{Z}/2)[X] \times (\mathbb{Z}/2)[Y] \to (\mathbb{Z}/2)[Z]$, i.e.,

$$\left(\sum_x \alpha_x x\right) \circ \left(\sum_y \beta_y y\right) \doteq \sum_{xy} \alpha_x \beta_y x \circ y = \sum_z \gamma_z z,$$

where $\gamma_z = \sum_{x \circ y = z} \alpha_x \beta_y$. The algorithm extends to this setting. (In particular, if the identity is false, the probability of detecting this in one round is at least 1/8.)

Multiple operations. Consider two binary operations $\cup$ and $\cap$ and a unary operation $'$ over S (representing perhaps some kind of complementation). We wish to verify that for every $a, b, c \in S$, $(a \cap (b \cup c))' = a' \cup (b' \cap c')$. This is done in quadratic time by verifying the identity at random points over $(\mathbb{Z}/2)[S]$.

4. Witness identification.

The above method disproves identities without producing an explicit counterexample. However, if desired, it can be used easily in order to produce such a counterexample. (For example, for the purpose of debugging.) We illustrate this in the case of associativity.

Let $\mathbf{a}, \mathbf{b}, \mathbf{c}$ be a nonassociative triple in $\mathcal{S}/2$. Let A, B, and C be the sets indexed by $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$ (i.e., those elements occurring with coefficient 1). We know that there is a nonassociative triple in $A \times B \times C$. Because of Theorem 3.1, we can choose a new triple to be tested by the algorithm as follows: $\mathbf{a}', \mathbf{b}', \mathbf{c}'$ are i.i.d., with $\mathbf{a}'_i = 0$ if $\mathbf{a}_i = 0$, and otherwise $\mathbf{a}'_i$ is uniformly selected in $\{0, 1\}$. $\mathbf{b}'$ and $\mathbf{c}'$ are chosen similarly. The probability that $\mathbf{a}', \mathbf{b}', \mathbf{c}'$ is a nonassociative triple is at least 1/8. We repeat this process until a nonassociative triple $\mathbf{a}', \mathbf{b}', \mathbf{c}'$ is found, and then continue in like manner. Each of A, B, and C approximately halves in each successful round (analysis below), hence the number of rounds will be $O(\log n)$ with high probability. Thus, with a logarithmic increase in cost, we can pinpoint a nonassociative triple in S. Below we carry out the analysis up to the point that each of A, B, and C has been reduced to size $n^{2/3}$ or less; at that point all triples may be examined in time $O(n^2)$.

This argument, like Theorem 2.1, generalizes in all the ways described in the previous section.

THEOREM 4.1. *Beginning with sets A, B, C containing some nonassociative triple, the probability that more than $(1 + \frac{1}{\log^{1/3} n})\frac{8}{3}\log_2 n$ rounds of the above process are required to reduce all sets to size at most $n^{2/3}$ is at most* $\exp(-\Omega(\log^{1/3} n))$.

Proof. After one round we have new sets A', B', C'; if the round was successful (i.e., $\mathbf{a}', \mathbf{b}', \mathbf{c}'$ was a nonassociative triple), these will usually be strict subsets of A, B, and C, while if the round was unsuccessful, then $A' = A$, $B' = B$, and $C' = C$.

We are interested in the distribution of the random variables $|A'|/|A|$, $|B'|/|B|$, and $|C'|/|C|$ (in each round). With probability at most 7/8, the round is unsuccessful and the ratios equal 1.

We have shown earlier that for any $a \in A, b \in B, c \in C$ for which (a, b, c) is nonassociative, and for any $R \subseteq A - \{a\}$, $S \subseteq B - \{b\}$, $T \subseteq C - \{c\}$, at least one of the eight triples $\{R, R \cup \{a\}\} \times \{S, S \cup \{b\}\} \times \{T, T \cup \{c\}\}$ is a successful choice. In the worst case from the point of view of the sizes of A', B', and C', the successful triple is always $(R \cup \{a\}) \times (S \cup \{b\}) \times (T \cup \{c\})$. Therefore, as an upper bound on our analysis, we can suppose that the distribution of the sizes of A', B' and C' in successful rounds are independent and binomially distributed in the ranges $[1, |A|], [1, |B|], [1, |C|]$.

We focus on just one of these sequences of ratios, e.g., $|A'|/|A|$; the union bound accounts for a factor of 3 in the probability of failure. The probability that less than fraction $\frac{1}{8}(1 - \frac{1}{2\log^{1/3} n})$ of the $(1 + \frac{1}{\log^{1/3} n})\frac{8}{3}\log_2 n$ rounds are successful is at most $\exp(-\Omega(\log^{1/3} n))$. The probability that there exists any successful round in which $|A'|/|A| > (1 + \frac{1}{\log n})/2$ is at most $\log n \exp(-\Omega(n^{2/3}/\log^2 n))$. Therefore with probability at least $1 - \exp(-\Omega(\log^{1/3} n))$, there are at least $\frac{1}{3}(1 + \frac{1}{2\log^{1/3} n})\log_2 n$ rounds in which $|A'|/|A| \leq (1 + \frac{1}{\log n})/2$. The size of the set remaining after these rounds is at most $n((1 + \frac{1}{\log n})/2)^{\frac{1}{3}(1+\frac{1}{2\log^{1/3} n})\log_2 n} \leq n^{2/3}\exp(-\Omega(\log^{2/3} n))$.

5. The cancellative case.

We claim the following concerning this case.

THEOREM 5.1. *Let $\circ$ be cancellative.*

(1) *If $\circ$ is nonassociative, then it has at least $n - 2$ nonassociative triples.*

(2) *It is possible to compute a generating set of size $\lfloor \log_2 n \rfloor + 1$ in quadratic time.*

The first observation implies that picking a random triple and checking it for associativity works essentially optimally for cancellative operations. The second observation implies an $O(n^2 \log n)$ deterministic algorithm for verifying associativity for cancellative binary operations by taking into account Light's observation mentioned in the introduction.

Proof.

(1) Let (a, b, c) be nonassociative and let $a = a' \circ a''$. Consider the following *cycle:* $(a' \circ a'') \circ (b \circ c) = ((a' \circ a'') \circ b) \circ c = (a' \circ (a'' \circ b)) \circ c = a' \circ ((a'' \circ b) \circ c) = a' \circ (a'' \circ (b \circ c)) = (a' \circ a'') \circ (b \circ c)$.

Each of these equalities is an application of the associative identity, and since the first fails, one of the other four must fail as well. In other words, if (a, b, c) is nonassociative, then at least one of the following must be nonassociative:

(i) (a', a'', b),

(ii) $(a', a'' \circ b, c)$,

(iii) (a'', b, c),

(iv) $(a', a'', b \circ c)$.

Since $\circ$ is cancellative, a can be written as $a' \circ a''$ in n different ways. For each of these, associativity fails in at least one of the four categories above. Thus, there is a category for which there are $n/4$ failures. Each category identifies either a' or a'', so there can be no duplications among the nonassociative triples listed for that category.

To improve to a bound of $n - 2$, note that there must be a nonassociative triple in which not all three elements are equal. This follows once we know that there are at least two triples in one of the above categories. Now we suppose without loss of generality that there is a nonassociative triple (a, b, c) for which $b \neq c$; if this is not the case, then there is one for which $a \neq b$, and the remainder of the argument can be "reflected" accordingly.

For each of the n choices of a', a'' such that $a' \circ a'' = a$, fix a category (i-iv) in which the triple is nonassociative. Observe that among the triples listed in categories (i), (ii), and (iv), there can be no duplication, since a' is identified as the first element in each of those triples. The question that remains is how much duplication there can be between list (iii) and lists (i), (ii), and (iv).

Since $b \neq c$, there can be no duplication between lists (i) and (iii).

If there is a duplication between a triple $(a'_2, a''_2 \circ b, c)$ in list (ii) and a triple

(a_3'', b, c) in list (iii), then the value of a_2'' is implied by the value of $a_2'' \circ b$; this in turn implies the value of a_2', which must be equal to a_3'', and this in turn implies the value of a_3'. Hence there can be only one triple common to lists (ii) and (iii).

A common triple to lists (iii) and (iv) is possible only if $b \circ c = c$; in this case, if triples (a_3'', b, c) in list (iii) and $(a_4', a_4'', b \circ c)$ in list (iv) are equal, then a_4'' must equal b. This implies the value of a_4', which in turn is equal to a_3'', and this implies the value of a_3'. Hence there can be only one triple common to (iii) and (iv).

Hence there are at least $n - 2$ nonassociative triples.

The construction in section 6 will show that this bound is tight to within a constant factor.

(2) Let $A \subseteq S$ be arbitrary. Denote by $\langle A \rangle$ the closure of A under $\circ$. Thus a generating set G is one such that $\langle G \rangle = S$. Let A be any set such that $\langle A \rangle \neq S$ and let $b \in S - \langle A \rangle$. Then, $|\langle A \cup \{b\} \rangle| \geq 2|\langle A \rangle|$ because right cancellativity implies that the elements of $b \circ \langle A \rangle$ are all distinct, and left cancellativity and the fact that $\langle A \rangle$ is closed imply that each element of $b \circ \langle A \rangle$ is outside of $\langle A \rangle$. This implies the existence of a size $\lfloor \log_2 n \rfloor + 1$ generating set. Moreover, observe that we can keep track of the closure (in a greedy manner) in time $O(n^2)$. $\square$

If an operation $\circ$ on a finite set S is both cancellative and associative, then $(S, \circ)$ is a group. Hence the above process of testing first for cancellativity and then for the associative identity yields the following.

THEOREM 5.2. *There is a deterministic $O(n^2 \log n)$ time algorithm to test whether $(S, \circ)$ is a group.*

6. Lower bound for verifying associativity.

6. Lower bound for verifying associativity. An $\Omega(n^2)$ lower bound for checking for associativity is immediate: if some product of elements is not examined, it may be changed to destroy associativity. Indeed, this points out that a sub-$O(n^2)$ algorithm can be foiled by simply changing the product of a random pair of elements. Using Yao's lemma [8], the $\Omega(n^2)$ lower bound holds for randomized algorithms (even those tolerating constant error probability) as well.

With some more attention, we obtain an $\Omega(n^2)$ lower bound which holds even in case $\circ$ is assumed to be cancellative. (In the cancellative case, as we have pointed out, there are simpler quadratic time algorithms than ours to check associativity.) Again the method is to change just a few values of $\circ$; and again Yao's lemma implies that the lower bound holds for randomized algorithms.

The argument is as follows. Suppose the algorithm is deterministic. Let S be the hypercube $(\mathbb{Z}/2)^m$, and let $\circ$ be vector addition. Suppose there exist $a, b, c \in S$, with $a \neq 0^m$, such that the entries $b \circ c$, $b \circ (a+c)$, $(a+b) \circ c$, $(a+b) \circ (a+c)$ are not examined by the algorithm. Then the behavior of the algorithm on $\circ$ is indistinguishable from its behavior on the operation $\circ'$ which we obtain by modifying the following entries of $\circ$:

(1) $b \circ' c = a + b + c$,
(2) $b \circ' (a + c) = b + c$,
(3) $(a + b) \circ' c = b + c$,
(4) $(a + b) \circ' (a + c) = a + b + c$.

Observe that $\circ'$ is still cancellative, but it is not associative because

$$(c \circ' (a + b)) \circ' c = (a + b + c) \circ' c = a + b,$$

while

$$c \circ' ((a+b) \circ' c) = c \circ' (b+c) = b.$$

Now, for any fixed a, there is a set $R \subseteq S$ of size $|R| = |S|/2$ such that for all $r, r' \in R$, $r + a \neq r'$. Letting b and c range over R, we obtain $|S|^2/4$ disjoint quadruples $\{\{b, c\}, \{b, a+c\}, \{a+b, c\}, \{a+b, a+c\}\}$ in $S \times S$. If the algorithm does not examine at least one value of $\circ$ in each quadruple, then it cannot distinguish $\circ$ from $\circ'$. Hence the algorithm must perform at least $n^2/4$ operations.

We now note that Theorem 5.1(1) is tight up to a constant factor. If we pick a, b, c independently, uniformly at random, then since $\circ'$ is cancellative, each one of the four pairs $\{a, b\}$, $\{a \circ' b, c\}$, $\{b, c\}$, and $\{a, b \circ' c\}$ is uniformly distributed in $S \times S$. Therefore the probability that (a, b, c) is a nonassociative triple for $\circ'$ is at most $16/n^2$; in other words, the number of nonassociative triples is at most $16n$.

7. Grigni's modification. M. Grigni has pointed out that for large k (the number of variables in the identity) it is useful to run our algorithm using $\mathcal{S}/p = (\mathbb{Z}/p)[S]$ for prime p such that $p > k$ (in place of $\mathcal{S}/2$), where k is the number of variables occurring in the identity. At the modest price of keeping track of larger coefficients and using a little more randomness, the probability of detecting failure of the identity in one run of the process is now at least $1 - k/p$, rather than 2^{-k}. This is for the following reason: the result of the computation in our algorithm is a pair of elements of $\mathcal{S}/p$, one corresponding to the left-hand side of the identity being checked and another corresponding to the right-hand side. Subtracting one from the other, the algorithm obtains an element $\sum_s \omega_s s \in \mathcal{S}/p$ and reports nonassociativity if any ω_s is nonzero. Each such ω_s is defined by a polynomial over $\mathbb{Z}/p$ in the (random) variables (each in $\mathbb{Z}/p$) used in the algorithm. The degree of each of these polynomials is at most k, and if the operation is not associative, then at least one polynomial is nonzero. By a lemma of J. T. Schwartz [6], the fraction of assignments on which a nonzero polynomial is 0 is at most k/p.

8. Examples.

Example 1. Let $S = \{1, 2, \ldots, n\}$, with $x \circ y = x$ for every (x, y). The only set of generators is S. Note also that $\circ$ is associative.

Example 2. Let $S = \{1, 2, \ldots, n\}$, with n at least 3. Let $1 \circ 2 = 2$. Let $x \circ y = 3$ for every $(x, y) \neq (1, 2)$. The only nonassociative triple is $(1, 1, 2)$.

For $n = 2$, any nonassociative operation has at least two nonassociative triples.

9. Open questions. While a nonassociative operation has a short witness for this property—and we are able, in fact, to efficiently find such a witness—we do not know of any short witness for associativity or any sub-n^3 time (randomized or deterministic) algorithm which, if the given operation is associative, proves this fact. (An exception is cancellative operations which, from Theorem 5.2, we know to have associativity witnesses of length $O(n^2 \log n)$.)

It remains to make any progress on verification of identities which are not read-once. A key example is the "distributive" identity $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$; it is not known whether this can be verified in less than cubic time.

Finally, we recall the interesting question encountered in the context of Light's observation as to whether there is an efficient algorithm which, given a set S and a binary operation $\circ$ on S, finds a set of generators for S of optimal or nearly optimal size.

Acknowledgments. We wish to acknowledge helpful conversations with Manuel Blum, Michelangelo Grigni, Sampath Kannan, and Thomas Wilke. We also wish to thank Kousha Etessami for first bringing the question to our attention.

REFERENCES

[1] R. H. Bruck, *A Survey of Binary Systems*, Springer-Verlag, Berlin, 1958.

[2] R. H. Bruck, *What is a loop?* in Studies in Modern Algebra, Studies in Mathematics 2, The Mathematical Association of America, Washington, DC, 1963.

[3] A. H. Clifford and G. B. Preston, *The Algebraic Theory of Semigroups*, AMS, Providence, RI, 1961.

[4] R. Freivalds, *Probabilistic machines can use less running time*, in Information Processing 77, Proceedings of IFIP Congress 77, B. Gilchrist, ed., North-Holland, Amsterdam, 1977, pp. 839–842.

[5] N. Jacobson, *Basic Algebra* II, W. H. Freeman, San Francisco, CA, 1980.

[6] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. ACM, 27 (1980), pp. 701–717.

[7] A. Weinstein, *Groupoids: Unifying internal and external symmetry*, Notices Amer. Math. Soc., 43 (1996), pp. 744–752.

[8] A. C. Yao, *Probabilistic computations: Toward a unified measure of complexity*, in Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science, Providence, RI, 1977, pp. 222–227.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1164–1188

EXPLORING UNKNOWN ENVIRONMENTS*

SUSANNE ALBERS† AND MONIKA R. HENZINGER‡

Abstract. We consider exploration problems where a robot has to construct a complete map of an unknown environment. We assume that the environment is modeled by a directed, strongly connected graph. The robot's task is to visit all nodes and edges of the graph using the minimum number R of edge traversals. Deng and Papadimitriou [*Proceedings of the* 31*st Symposium on the Foundations of Computer Science*, 1990, pp. 356–361] showed an upper bound for R of $d^{O(d)}m$ and Koutsoupias (reported by Deng and Papadimitriou) gave a lower bound of $\Omega(d^2m)$, where m is the number of edges in the graph and d is the minimum number of edges that have to be added to make the graph Eulerian. We give the first subexponential algorithm for this exploration problem, which achieves an upper bound of $d^{O(\log d)}m$. We also show a matching lower bound of $d^{\Omega(\log d)}m$ for our algorithm. Additionally, we give lower bounds of $2^{\Omega(d)}m$, respectively, $d^{\Omega(\log d)}m$ for various other natural exploration algorithms.

Key words. directed graph, exploration algorithm

AMS subject classifications. 05C20, 68Q20, 68Q25, 68R10

PII. S009753979732428X

1. Introduction.

Suppose that a robot has to construct a complete map of an unknown environment using a path that is as short as possible. In many situations it is convenient to model the environment in which the robot operates by a graph. This allows us to neglect geometric features of the environment and to concentrate on combinatorial aspects of the exploration problem. Deng and Papadimitriou [12] formulated thus the following exploration problem. A robot has to explore all nodes and edges of an unknown, strongly connected directed graph. The robot *visits* an edge when it traverses the edge. A node or edge is *explored* when it is visited for the first time. The goal is to determine a *map*, i.e., the adjacency matrix of the graph, using the minimum number R of edge traversals. At any point in time the robot knows (1) all visited nodes and edges and can recognize them when reencountered, and (2) the number of unvisited edges leaving any visited node. The robot does not know the head of unvisited edges leaving a visited node or the unvisited edges leading into a visited node. At each point in time, the robot visits a *current node* and has the choice of leaving the current node by traversing a specific known or an arbitrary (i.e., given by an adversary) unvisited outgoing edge. An edge can be traversed only from tail to head, not vice versa.

If the graph is Eulerian, $2m$ edge traversals suffice [12], where m is the number of edges. This immediately implies that undirected graphs can be explored with at most $4m$ traversals. In fact, using depth-first-search they can be explored using $2m$ edge traversals. For a non-Eulerian graph, let the *deficiency* d be the minimum number of edges that have to be added to make the graph Eulerian. Deng and

*Received by the editors July 14, 1997; accepted for publication (in revised form) February 12, 1999; published electronically February 10, 2000. A preliminary version of this paper was presented at the 29th Annual ACM Symposium on the Theory of Computing (STOC), El Paso, TX, 1997.
http://www.siam.org/journals/sicomp/29-4/32428.html

†Lehrstuhl Informatik II, Universitaet Dortmund, 44221 Dortmund, Germany (albers@ls2.cs.uni-dortmund.de). The work of this author supported in part by the Deutsche Forschungsgemeinschaft, project Al 464/1-1.

‡Systems Research Center, Compaq Computer Corporation, 130 Lytton Ave, Palo Alto, CA 94301 (monika@pa.dec.com). The work was supported by the NSF CAREER Award, grant CCR-9501712.

Papadimitriou [12] suggested studying the dependence of R on m and d and showed the first upper and lower bounds: they gave a graph such that any algorithm needs $\Omega(d^2m/\log d)$ edge traversals, and they also presented an algorithm that achieves an upper bound of $d^{O(d)}m$. Koutsoupias [16] improved the lower bound to $\Omega(d^2m)$. Deng and Papadimitriou asked the question whether the exponential gap between the upper and lower bound can be closed. Our paper is a first step in this direction: we give an algorithm that is subexponential in d; namely, it achieves an upper bound of $d^{O(\log d)}m$. We also show a matching lower bound for our algorithm and exponential lower bounds for various other exploration algorithms.

Note that d arises also in the complexity of the "offline" version of the problem: Consider a directed cycle with one edge replaced by $d+1$ parallel edges. On this graph any Eulerian traversal requires $\Omega(dm)$ edge traversals. A simple modification of the Eulerian online algorithm solves the offline problem on any directed graph with $O(dm)$ edge traversals.

Related work. Exploration and navigation problems for robots have been studied extensively in the past. The exploration problem in this paper was formulated by Deng and Papadimitriou based on a learning problem proposed by Rivest [19]. Betke, Rivest, and Singh [8] and Awerbuch et al. [1] studied the problem of exploring an undirected graph and requiring additionally that the robot returns to its starting point every so often. Bender and Slonim [9] showed how two cooperating robots can learn a directed graph with indistinguishable nodes, where each node has the same number of outgoing edges. Subsequent to the work in [12], Deng, Kameda, and Papadimitriou [11] investigated a geometric exploration problem, whose goal is to explore a room with or without polygonal obstacles. Hoffmann et al. [15] gave an improved exploration strategy for rooms without obstacles. More generally, theoretical studies of exploration and navigation problems in unknown environments were initiated by Papadimitriou and Yannakakis [18]. They considered the problem of finding a shortest path from a point s to a point t in an unknown environment and presented many geometric and graph-based variants of this problem. Blum, Raghavan, and Schieber [7] investigated the problem of finding a shortest path in an unfamiliar terrain with convex obstacles. More work on this problem includes [2, 5, 6].

Our results. Our main result is a new robot strategy that explores an arbitrary graph with deficiency d and traverses each edge at most $(d+1)^7d^{2\log d}$ times; see section 3. The algorithm does not need to know d in advance. The total number of traversals needed by the algorithm is also $O(\min\{nm, dn^2+m\})$, where n is the number of nodes. At the end of section 3 we show that any exploration algorithm that fulfills two intuitive conditions achieves an upper bound of $O(\min\{nm, dn^2+m\})$. A depth-first search strategy obtaining this bound was independently developed by Kwek [17].

In section 4 we demonstrate that our analysis of the new robot strategy is tight: There exists a graph that is explored by our algorithm using $d^{\Omega(\log d)}m$ edge traversals. We also show that various variants of the algorithm have the same lower bound. In section 2, we present lower bounds of $2^{\Omega(d)}m$, respectively, $d^{\Omega(\log d)}m$ for various other natural exploration algorithms, to give some intuition for the problem.

Our exploration algorithm tries to explore new edges that have not been visited so far. That is, starting at some visited node x with unvisited outgoing edges, the robot explores new edges until it gets *stuck* at a node y, i.e., it reaches y on an unvisited incoming edge and y has no unvisited outgoing edge. Since the robot is not allowed to traverse edges in the reverse direction, an adversary can always force the robot to visit unvisited nodes until it finally gets stuck at a visited node.

The robot then relocates, using visited edges, to some visited node z with unexplored outgoing edges and continues the exploration. The *choice* of z is the only difference between various algorithms and the *relocation* to z is the only step where the robot traverses visited edges. To minimize R we have to minimize the total number of edges traversed during all relocations. It turns out that a locally greedy algorithm that tries to minimize the number of traversed edges during each relocation is not optimal: it has a lower bound of $2^{\Omega(d)}m$ (see section 2).

Instead, our algorithm uses a divide-and-conquer approach. The robot explores a graph with deficiency d by exploring d^2 subgraphs with deficiencies $d/2$ each and uses the same approach recursively on each of the subgraphs. To create subgraphs with small deficiencies, the robot keeps track of visited nodes that have more visited outgoing than visited incoming edges. Intuitively, these nodes are *expensive* because the robot, when exploring new edges, can get stuck there. The relocation strategy tries to keep portions of the explored subgraphs "balanced" with respect to their expensive nodes. If the robot gets stuck at some node, then it relocates to a node z such that "its" portion of the explored subgraph contains the minimum number of expensive nodes.

2. Lower bounds for various algorithms. In this section we prove a lower bound of $2^{\Omega(d)}m$ for a locally greedy, a depth-first, and a breadth-first algorithm. We also give a lower bound of $d^{\Omega(\log d)}m$ for a generalized greedy strategy.

A related problem, for which lower bounds have been studied extensively, is the *s–t connectivity problem* in directed graphs; see [3, 4, 14] and references therein. Given a directed graph, the problem is to decide whether there exists a path from a distinguished node s to a distinguished node t. Most of the results are developed in the JAG model by Cook and Rackoff [10]. The best time–space tradeoffs currently known [4, 14] only imply a polynomial lower bound on the computation time if no upper bounds are imposed in the space used by the computation. Given the current knowledge of the s–t connectivity problem it seems unlikely that one can prove super-polynomial lower bounds for a *general* class of graph exploration algorithms.

In the following let G be a directed, strongly connected graph and let v be a node of G. Let $in(v)$ and $out(v)$ denote, respectively, the number of incoming and outgoing edges of v. Let the *balance* $bal(v) = out(v) - in(v)$. For a graph with deficiency d there exist at most d nodes s_i, $1 \leq i \leq d$, such that $bal(s_i) < 0$. Every node s_i with $bal(s_i) < 0$ is called a *sink*. Note that $-\sum_{s,bal(s)<0} bal(s) = d$. We use the term *chain* to denote a path. A chain is a sequence of nodes and edges $x_1, (x_1, x_2), x_2, (x_2, x_3), \ldots, (x_{k-1}, x_k), x_k$ for $k > 1$.

Greedy: If stuck at a node y, move to the nearest node z that has new outgoing edges.

Generalized-Greedy: At any time, for each path in the subgraph explored so far, define a lexicographic vector as follows. For each edge on the path, determine its current *cost*, which is the number of times the edge was traversed so far. Sort these costs in nonincreasing order and assign this vector to the path. Whenever stuck at a node y, out of all paths to nodes with new outgoing edges traverse the path whose vector is lexicographic minimum.

Depth-First: If stuck at a node y, move to the most recently discovered node z that can be reached and that has new outgoing edges.

Breadth-First: Let v be the node where the exploration starts initially. If stuck at a node y, move to the node z that has the smallest distance from v among all nodes with new outgoing edges that can be reached from y.

THEOREM 1. *For Greedy, Depth-First, and Breadth-First, and for every d, there exist graphs of deficiency d that require $2^{\Omega(d)}m$ edge traversals.*

Proof (*Greedy*). Basically *Greedy* fails since it is easy to "hide" a subgraph (see Figure 1). Whenever *Greedy* discovers this subgraph, the adversary can force it to repeat all the work done so far.

The graph G consists of two parts: (1) a cycle C_0 of three edges and nodes v, $v^1(C_0)$, and $v^2(C_0)$, and (2) a recursively defined problem P^d. A *problem* P^δ, for any integer $\delta \geq 2$, is a subgraph that has two *incoming* edges whose startnodes do not belong to P^δ but whose endnodes do, and δ *outgoing* edges whose startnode belongs to P^δ but whose endnodes do not. A *problem* P^1 is defined in the same way as a problem P^δ, $\delta \geq 2$, except that P^1 has only one incoming edge. In the case of P^d, the two incoming edges start at $v^1(C_0)$ and $v^2(C_0)$, respectively; the d outgoing edges all point to v.

For the description of P^δ we also need recursively defined problems Q^δ. These problems are identical to P^δ except that, for $\delta > 2$, Q^δ has exactly δ incoming edges.

A problem P^δ, $\delta = 1, 2$, consists of δ chains of three edges each. The first edge of each chain is an incoming edge into P^δ; the last edge of each chain is an outgoing edge. A problem Q^δ, $\delta = 1, 2$, is the same as P^δ.

We proceed to define P^δ, for $\delta > 2$. One of the incoming edges of P^δ is the first edge of a chain D^δ consisting of three edges and the other incoming edge is the first edge of a long chain C^δ. For each of these chains C^δ and D^δ, the last edge is an outgoing edge of P^δ. If $\delta = 3$, the last interior node of each of the chains C^δ and D^δ has an additional outgoing edge pointing into a problem P^1. If $\delta \geq 4$, then (a) the last two interior nodes of C^δ each have an additional outgoing edge pointing into a subproblem $P^{\delta-2}$, and (b) the last two interior nodes of D^δ each have an additional outgoing edge pointing into a subproblem $Q^{\delta-2}$. There are $\delta - 2$ edges leaving $P^{\delta-2}$, exactly $\max\{0, \delta - 4\}$ of which point to nodes of $Q^{\delta-2}$ such that each node in $Q^{\delta-2}$ that has k more outgoing than incoming edges, for some $0 \leq k \leq \max\{0, \delta - 4\}$, receives k incoming edges from $P^{\delta-2}$. The remaining outgoing edges of $P^{\delta-2}$ point to the interior nodes of D^δ that have additional outgoing edges. The problem $Q^{\delta-2}$ has $\delta - 2$ outgoing edges all of which are outgoing edges of P^δ. The total number of edges in C^δ is 2 plus the number of edges of D^δ plus the total number of edges contained in the subproblem $Q^{\delta-2}$ below D^δ.

A problem Q^δ, $\delta > 2$, is the same as P^δ except that the subproblem $P^{\delta-2}$ is replaced by another $Q^{\delta-2}$ problem. That is, Q^δ is composed of chains C^δ, D^δ, and problems $Q_i^{\delta-2}$, $i = 1, 2$. As mentioned before, Q^δ has exactly δ incoming edges.

Greedy is started at node v and traverses first chain C_0. Then it either explores C^d or D^d. In either case, afterwards *Greedy* explores all edges of Q^{d-2} since C^d is prohibitively long. Thus, P^{d-2} is "hidden" from *Greedy*. We exploit this in the analysis: Let $N(\delta)$ be the number of times that *Greedy* explores edges of a problem P^δ or Q^δ, gets stuck at some node, and cannot relocate to a suitable node by using only edges in P^δ, respectively, Q^δ. We show that $N(\delta) \geq 2^{\delta/2}$. Since the edge leaving v is traversed every time the algorithm cannot relocate by using only edges in P^d, the bound follows.

A problem P^δ contains two subproblems $P^{\delta-2}$ and $Q^{\delta-2}$. Note that (a) because of chain D^δ, no node in $Q^{\delta-2}$ can reach a node of $P^{\delta-2}$ without leaving P^δ, and (b) $Q^{\delta-2}$ is completely explored when the exploration of $P^{\delta-2}$ starts and all paths starting in $P^{\delta-2}$ lead through D^δ or $Q^{\delta-2}$. Thus, every time *Greedy* gets stuck in

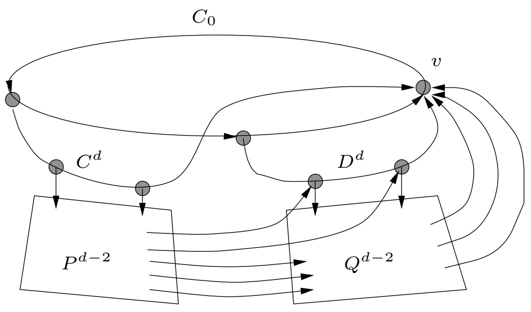


FIG. 1. *The graph for* Greedy.

a subproblem $P^{\delta-2}$ or $Q^{\delta-2}$ and has to leave $P^{\delta-2}$, respectively, $Q^{\delta-2}$ in order to resume exploration, it also has to leave P^{δ}. For $Q^{\delta-2}$ the statement follows from (a); for $P^{\delta-2}$ it follows from (a) and (b). In the same way, we can argue for a problem Q^{δ}. Thus, $N(\delta) \geq 2N(\delta-2)$. Since, for $\delta = 1, 2$, $N(\delta) \geq 1$, we obtain $N(\delta) \geq 2^{\delta/2}$.

This implies that the edge e on C_0 leaving v is traversed $2^{\Omega(d)}$ times. The desired bound follows by replacing e with a path consisting of $\Theta(m)$ edges.

Depth-First: We can use the same graph as in the case of the *Greedy* algorithm. *Depth-First* will explore all edges in Q^{d-2} before it will start exploring P^{d-2}.

Breadth-First: Again we can use the same graph as in the lower bound for *Greedy*. The last two interior nodes of C^d have a larger distance from the initial node v than all nodes on D^d and in Q^{d-2}. Thus Q^{d-2} is finished before *Breadth-First* starts exploring P^{d-2}. ∎

THEOREM 2. *For Generalized-Greedy and for every d, there exists a graph of deficiency d that requires $d^{\Omega(\log d)}m$ edge traversals.*

Proof. The graph used for the lower bound is outlined in Figure 2. The basic idea in the lower bound construction is as follows. *Generalized-Greedy* explores each subgraph Q_i^{γ} and its sibling R_i^{γ} "in parallel." Without loss of generality we can assume that the last chain traversed in the two subgraphs lies in Q_i^{γ} and the algorithm continues to explore Q_{i+1}^{γ} and R_{i+1}^{γ}. Let $N(\gamma)$ denote the number of times that the algorithm has to leave R_i^{γ} and traverse the root. We will show that $N(4\gamma) \geq \gamma N(\gamma)$, which implies that the root has to be traversed $N(d) \geq d^{\Omega(\log d)}$ times.

To be precise we show the bound for d being a power of 4. The bound for all values of d follows by rounding down to the largest power of 4 smaller than d. The graph G consists of two parts: (1) a cycle C_0 with nodes v, $v^1(C_0)$ and $v^2(C_0)$, and (2) a recursively defined subproblem P^d. Problem P^d has two incoming edges, one starting at $v^1(C_0)$ and one starting at $v^2(C_0)$. It also has d outgoing edges, all pointing to v. The subproblem P^d is a union of chains C, each of which consists of three edges, a startnode, an endnode, and two *interior nodes* $v^1(C)$ and $v^2(C)$. The interior nodes have at most one additional outgoing edge. We proceed to define P^{δ} and the "sibling" graphs Q^{δ} and R^{δ}, for all $\delta \leq d$ that are a power of 4, and then show the lower bound on this graph.

A *problem* P^{δ}, $\delta > 1$, is a graph with two incoming edges and exactly δ outgoing edges. A *problem* R^{δ}, $\delta > 1$, consists of P^{δ} with $\delta - 2$ additional incoming edges. The *problem* Q^{δ} consists of R^{δ} with two additional incoming and two additional outgoing edges.

$\delta = 1$: A problem P^1 consists of one chain. The incoming edge of P^1 is the first

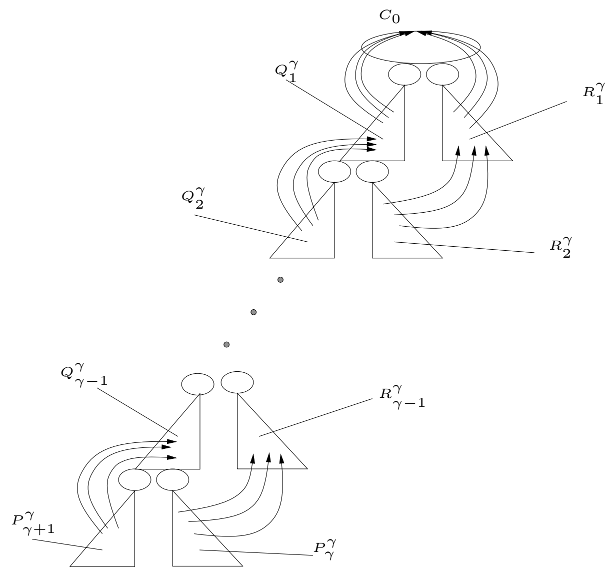


FIG. 2. *The graph for* Generalized-Greedy.

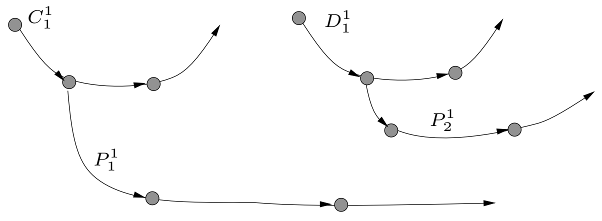


FIG. 3. *The subproblem* P^4.

edge of the chain and the outgoing edge of P^1 is the last edge of the chain. In P^1, the interior nodes of the chain have no additional outgoing edges; in Q^1, each interior node has one additional incoming and one additional outgoing edge. Problem R^1 is equal to P^1.

$\delta = 4$: A problem P^4 consists of two subproblems P_1^1 and P_2^1 and chains C_1^1 and D_1^1, whose first interior nodes have one additional outgoing edge (see Figure 3). The outgoing edge of C_1^1 is the incoming edge of P_1^1 and the corresponding edge of D_1^1 is the incoming edge of P_2^1. The last edge of C_1^1 and D_1^1 and the outgoing edges of P_1^1 and P_2^1 are outgoing edges of P^4. A problem R^4 is P^4 with two additional incoming edges, one at the startnode of P_1^1 and one at the startnode of P_2^1. A problem Q^4 is R^4 with two additional incoming and outgoing edges; each interior node of P_1^1 has an additional incoming and outgoing edge.

$\delta = 4^l$, for some $l \geq 2$: Let $\gamma = \delta/4$. It is simpler to describe Q^δ first. The construction is depicted in Figure 4. Every node has the same indegree as outdegree, i.e., there are no sinks. Problem Q^δ consists of subproblems Q_i^γ and R_i^γ, for $1 \leq i \leq \gamma$, connected by chains C_i^γ and D_i^γ, for $1 \leq i \leq \gamma$, whose interior nodes each have an additional outgoing edge.

The C-chains and Q-subproblems are interleaved as follows. The two edges leaving the interior nodes of C_1^γ point into Q_1^γ. In general, the edges leaving the interior nodes

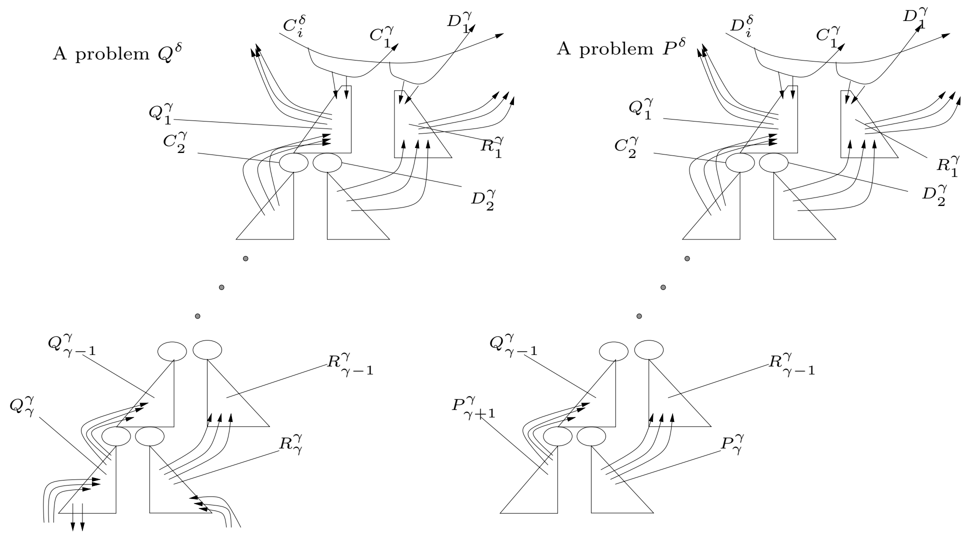


FIG. 4. *The subproblems Q^δ and P^δ.*

of C_i^γ point into Q_i^γ. The same holds for the D-chains and R-subproblems. The first edge of C_i^γ and of D_i^γ are incoming edges of Q^δ, for $i = 1$, and start in Q_{i-1}^γ, for $1 < i \leq \gamma$, on a node of the leftmost subproblem Q^1 contained in Q_{i-1}^γ. Recall that this problem consists of one chain with two additional incoming and outgoing edges. One of these outgoing edges is the first edge of C_i^γ and the second outgoing edge is the first edge of D_i^γ.

Additionally, the subproblems are connected as follows. Recall that γ edges leave R_i^γ. For $i = 1$, the edges leaving R_i^γ are outgoing edges of Q^δ. For $1 < i \leq \gamma$, two edges leaving R_i^γ point to the interior edges of D_{i-1}^γ. Additionally, there are $\gamma - 2$ edges leaving R_i^γ and pointing into R_{i-1}^γ such that every node in R_{i-1}^γ that has k more outgoing than incoming edges, for $k > 0$, receives k edges from R_i^γ. The same holds for Q_i^γ with C_{i-1}^γ. The problem Q_γ^γ has γ incoming edges which are incoming edges for Q^δ; the problem R_γ^γ has $\gamma - 2$ incoming edges which are incoming edges for Q^δ.

There are $4\gamma + 2 = \delta + 2$ outgoing edges in Q^δ: the last edge of C_i^γ and the last edge of D_i^γ, for $1 \leq i \leq \gamma$, all edges leaving R_1^γ, all but two edges leaving Q_1^γ (the other two are the incoming edges of D_2^γ and C_2^γ), and two edges leaving Q_γ^γ. There are also $\delta + 2$ incoming edges: the first edge of C_1^γ and of D_1^γ, the edges pointing to the two interior nodes of C_γ^γ and D_γ^γ, the γ incoming edges of Q_γ^γ, the $\gamma - 2$ incoming edges of R_γ^γ, and $2\gamma - 2$ incoming edges ending at the startnodes of C_i^γ and D_i^γ, for $2 \leq i \leq \gamma$.

A problem P^δ consists of 2γ chains C_i^γ and D_i^γ, $1 \leq i \leq \gamma$, as well as two subproblems P_i^γ, $\gamma \leq i \leq \gamma + 1$, and $2(\gamma - 1)$ subproblems Q_i^γ and R_i^γ, $1 \leq i \leq \gamma - 1$. These components are assembled in the same way as in Q^δ, except that Q_γ^γ is replaced by $P_{\gamma+1}^\gamma$ and R_γ^γ is replaced by P_γ^γ. Problems P_γ^γ and $P_{\gamma+1}^\gamma$ each have only two incoming edges from C_γ^γ and D_γ^γ, respectively.

There are $4\gamma = \delta$ outgoing edges in P^δ: the last edge of C_i^γ and the last edge of D_i^γ, for $1 \leq i \leq \gamma$, all but two edges leaving Q_1^γ (the other two are the incoming edges of D_2^γ and C_2^γ), all edges leaving R_1^γ. There are two incoming edges in P^δ. The first edge of C_1^γ and of D_1^γ are incoming edges in every problem P^δ. The following

$\delta - 2$ nodes are sources for P^δ: the two interior nodes of C_γ^γ and of D_γ^γ, the $2\gamma - 2$ startnodes of C_i^γ and D_i^γ, for $2 \le i \le \gamma$, the $\gamma - 2$ sources of P_γ^γ, and the $\gamma - 2$ sources of $P_{\gamma+1}^\gamma$.

A problem R^δ is a problem P^δ with an incoming edge into every source of P^δ. Thus there are δ incoming and δ outgoing edges.

We analyze *Generalized-Greedy* on G. For simplicity we only discuss the exploration of a problem Q^δ. The argument for P^δ and R^δ is analogous. As before, let $\gamma = \delta/4$. We show inductively that the symmetric construction of Q_i^γ and R_i^γ attached to C_i^γ and D_i^γ as well as the definition of *Generalized-Greedy* imply that Q_i^γ and R_i^γ are explored symmetrically. That is, during two consecutive traversals of C (in order to resume exploration in Q_i^γ or R_i^γ), *Generalized-Greedy* proceeds once into Q_i^γ and once into R_i^γ, where C is the chain at which chains C_i^γ and D_i^γ start. This obviously holds for $i = 1$. Assume it holds for i and we want to show it for $i + 1$. Note that Q_i^γ and R_i^γ differ only in the last chain that *Generalized-Greedy* explores in Q_i^γ, respectively, R_i^γ. Thus, until the traversal of the earlier of the last chain of Q_i^γ and the last chain of R_i^γ, *Generalized-Greedy* does not distinguish Q_i^γ from R_i^γ. Hence we can assume without loss of generality that *Generalized-Greedy* traverses first the last chain of R_i^γ, and afterwards the last chain of Q_i^γ. (Think of an adversary "giving" to *Generalized-Greedy* first the last chain of R_i^γ and then the last chain of Q_i^γ.) Then *Generalized-Greedy* explores C_{i+1}^γ and D_{i+1}^γ, and afterwards Q_{i+1}^γ and R_{i+1}^γ symmetrically. Thus, when *Generalized-Greedy* explores a subproblem R_i^γ, $1 \le i \le \gamma$, subproblems R_j^γ with $1 \le j < i$ are already finished.

Whenever *Generalized-Greedy* gets stuck in R_i^γ, $1 \le i \le \gamma$, and has to leave R_i^γ in order to resume exploration, it also has to leave the "parent problem" Q^δ (or P^δ, R^δ). This is because the chains D_i^γ, $1 \le i \le \gamma$, prevent the algorithm from reaching a chain in Q_j^γ, $1 \le j \le i$, from where unfinished chains in Q^δ, (P^δ, R^δ) can be reached. On the way from R_i^γ to an outgoing edge of the parent problem, *Generalized-Greedy* can traverse problems R_j^γ, $j \le i$. As shown in Figure 4, the subproblems are finished; no further exploration of R_j^γ is possible. The same arguments hold when the algorithm gets stuck in a problem P_γ^γ.

For any δ, $4 \le \delta \le d$, let $N(\delta)$ be the number of times *Generalized-Greedy* generates a chain in P^δ or R^δ, gets stuck, and has to leave P^δ or R^δ in order to continue exploration. Then $N(\delta) \ge \gamma N(\gamma) = \delta/4 N(\delta/4)$. Since $N(1) \ge 1$, we have $N(d) \ge d^{\Omega(\log d)}$ and hence the edge leaving node v is traversed $d^{\Omega(\log d)}$ times. $\square$

3. An algorithm for graphs with deficiency *d*.

3.1. The *Balance* algorithm. We present an algorithm that explores an unknown, strongly connected graph with deficiency d, without knowing d in advance. First we give some definitions. At the start of the algorithm, all edges are *unvisited* or *new*. An edge becomes *visited* whenever the robot traverses it. A node is *finished* whenever all its outgoing edges are visited. The robot is *stuck* at a node y if the robot enters a finished node y on an unvisited edge. A sink is *discovered* whenever the robot gets stuck at the sink for the first time. We assume that whenever the robot discovers a new sink, the subgraph of explored edges is strongly connected. This does not hold in general, but by properly restarting the algorithm, the problem can be reduced to the case described here. Details are given in section 3.2.

Assume the algorithm knew the d missing edges $(s_1, t_1), (s_2, t_2), \ldots, (s_d, t_d)$ and a path from each s_i to t_i. Then a modified version of the Eulerian algorithm could be executed: Whenever the original Eulerian algorithm traverses an edge (s_i, t_i), the modified Eulerian algorithm traverses the corresponding path from s_i to t_i. Obviously, the modified algorithm traverses each edge at most $2d + 2$ times. Thus, the problem is to find the missing edges and corresponding paths.

Our algorithm tries to find the missing edges by maintaining d edge-disjoint chains, such that the endnode of chain i is s_i and the startnode of chain i is our current *guess* of t_i. As the algorithm progresses, paths can be appended at the start of each chain. At termination, the startnode of chain i is indeed t_i. To mark chain i all edges on chain i are colored with color i.

The algorithm consists of two phases.

Phase 1. Run the algorithm of [12] for Eulerian graphs. Since G is not Eulerian, the robot will get stuck at a sink s. At this point stop the Eulerian graph algorithm and goto Phase 2. The part of the graph explored so far contains a cycle C_0 containing s [12]. We assume that at the end of Phase 1 all visited nodes and edges not belonging to C_0 are marked again as unvisited.

Phase 2. Phase 2 consists of *subphases*. During each subphase the robot visits a *current* node x of a *current* chain C and makes progress towards finishing the nodes of C. The current node of the first subphase is s, its current chain is C_0. The current node and current chain of subphase j depend on the outcome of subphase $j - 1$.

A chain can be in one of three states: *fresh*, *in progress*, or *finished*. A chain C is *finished* when all its nodes are finished; C is *in progress* in subphase j if C was a current chain in a subphase $j' \leq j$ and C is not yet finished; C is *fresh* if it is not finished and not yet in progress.

Up to $d+1$ chains in progress and up to d fresh chains can exist at the same time. The invariant that there are always at most $d+1$ chains in progress is convenient but not essential in the analysis of the algorithm. The invariant that there exist always at most d fresh chains is crucial. Every startnode of a fresh chain has more visited outgoing than visited incoming edges and, thus, the robot can get stuck there. In the analysis we require that there always exist at most d such nodes.

The algorithm marks the current guess for t_i with a *token* τ_i, for $1 \leq i \leq d$. In fact, every startnode of a fresh chain represents the current guess for some t_i, $1 \leq i \leq d$, and thus has a token τ_i. To simplify the description of the relocation process, each token is also assigned an *owner* which is a chain that contains the node on which the token is placed. More specifically, the owner of τ_i is the chain that was the current chain when the path from the current guess of t_i to s_i was extended last. Note that the owner is not the chain from the current guess of t_i to s_i. A node can be the current guess for more than one node t_i and, thus, have more than one token.

From a high-level point of view, at any time, the subgraph explored so far can be partitioned into chains, namely C_0 and the chains generated in Phase 2. During the actual exploration in the subphases, the robot travels between chains. While doing so, it generates or extends fresh chains, which will be taken into progress later, and finishes the chains currently in progress.

We give the details of a subphase. First, the algorithm tests if x has an unvisited outgoing edge.

1. If x does not have an unvisited outgoing edge and x is not the endnode of C, then the next node of C becomes the current node and a new subphase is started.
2. If x has no unvisited outgoing edge and x is the endnode of C, procedure *Relocate* is called to decide which chain becomes the current chain and to

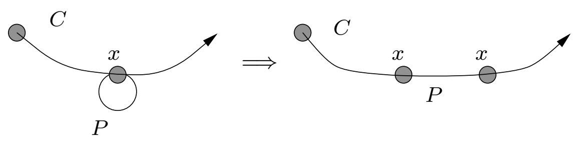


FIG. 5. *Case* 1.

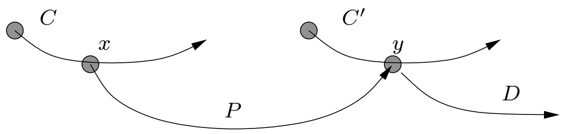


FIG. 6. *Case* 2.

move the robot to the startnode z of this chain. Node z becomes the current node.

3. If x has unvisited outgoing edges, the robot repeatedly explores unvisited edges until it gets stuck at a node y. Let P be the path traversed.
   We distinguish four cases.

Case 1. $y = x$. Cut C at x and add P to C (see Figure 5). The robot returns to x and the next phase has the same current node and current chain.

Case 2. $y \neq x$, y has a token τ_i and is the startnode of a fresh chain D (see Figure 6). Append P at D to create a longer fresh chain, and move the token from y to x. The current chain C becomes the *owner* of the token, the previous owner becomes the current chain, and y becomes the current node.

Case 3. $y \neq x$, y has a token τ_i but is not the startnode of a fresh chain. This is the same as Case 2 except that no fresh chain starts at y. The algorithm creates a new fresh chain of color i consisting of P. It moves the token from y to x and C becomes the owner of the token. The previous owner of the token becomes the current chain and y becomes the current node.

Case 4. $y \neq x$ and y does not own a token. In this case $bal(y) < 0$. If $bal(y) = -k$, then this case occurs k times for y. Let i be the number of existing tokens. The algorithm puts a new token τ_{i+1} on x with owner C, creates a fresh chain of color $i+1$ consisting of P (the first chain with color $i+1$), and moves the robot back to s. The initial chain C_0 becomes the current chain and s becomes the current node.

This leads to the algorithm given in Figure 7. We use x to denote the current node, C to denote the current chain, k the number of tokens used, and j the highest index of a chain. Lines 4–17 of the code correspond to item 3 above. Lines 6 and 7 correspond to Case 1, lines 8–13 correspond to Cases 2 and 3, and lines 14–16 correspond to Case 4. Lines 18 and 19 implement items 1 and 2, respectively. In line 13, C' is the chain that was the previous owner of τ_i and becomes the new current chain.

Additionally, the algorithm maintains a tree T, such that each chain C corresponds to a node $v(C)$ of T and $v(C')$ is a child of $v(C)$ if the last subpath appended to C' was explored while C was the current chain. Conversely, we use $C(v)$ to denote the chain represented by node v. For each chain there is exactly one node in the tree. Note that the tree changes dynamically. If in line 10 of the algorithm a path P is appended at a chain D, then the node representing the resulting chain becomes a child of $v(C)$, i.e., a child of the node representing the current chain C. The node $v(D)$ is removed. Since only fresh chains are reassigned, each added or removed node

Algorithm Balance

```
 1.  $j := 0$, $k := 0$, $x := s$, $C := C_0$.
 2.  repeat
 3.      while $C$ is unfinished do
 4.          while $\exists$ new outgoing edge at $x$ do
 5.              Traverse new edges starting at $x$ until stuck at a node $y$.
                 Call this path $P$.
 6.              if $y = x$ then
 7.                  Insert $P$ into $C$;
 8.              else if $y$ has a token $\tau_i$ then
 9.                      if $\exists$ chain $D$ of color $i$ starting in $y$ and $D$ is fresh then
10.                          Concatenate $P$ with $D$;
11.                      else
12.                          $j := j + 1$; $C_j :=$ chain that consists of $P$;
13.                      $C' := owner(\tau_i)$; Place $\tau_i$ on $x$; $owner(\tau_i) := C$; $x := y$;
                         $C := C'$;
14.                  else ($*$ $y \neq x$ and $y$ has no token $*$)
15.                      $j := j + 1$; $C_j :=$ chain that consists of $P$;
16.                      $k := k + 1$; Place token $\tau_k$ on $x$; $owner(\tau_k) := C$; $x := s$;
                         $C := C_0$;
17.                      Move robot to $x$;
18.          Move robot to first unfinished node $z$ that appears on $C$ after its
             startnode;  $x := z$;
19.      $C := Relocate(C)$;  $x :=$ startnode of $C$;
20.  until $C =$ empty_chain.
```

FIG. 7. *The Balance algorithm.*

is a leaf. This process ensures that the structure of nodes is indeed a tree.

We use T_v to denote the subtree of T rooted at v and say C is *contained* in T_v if $v(C)$ lies in T_v. We also say a token τ or an edge e is *contained* in T_v if $owner(\tau)$, respectively, the chain of e is contained in T_v. If all chains in T_v are finished, we say that T_v is *finished*. To represent T, the algorithm assigns a *parent* to each chain.

To relocate, the robot needs to be able to move on explored edges from the endpoint of a chain C to its startnode. This is always possible, since at the beginning of each subphase the explored edges form a strongly connected graph. To avoid an edge being traversed often for this purpose, we define for each chain C a path $closure(C)$ connecting the endnode of C with the startnode of C such that an edge belongs to $closure(C)$ for at most $d^{O(\log d)}$ chains C. Finally, we will show that $closure(C)$ is traversed at most $O(d^2)$ times.

A path Q is called a *C-completion* if it connects the endnode of a chain C with the startnode of C. A path Q in the graph is called *i-uniform* if it is a concatenation of chains of color i. Let u be a node of T. A path Q in the graph is *T_u-homogeneous* if any maximal subpath R of Q that does not belong to T_u is (a) i-uniform for some color i; (b) the edge of Q preceding R is the last edge of a chain of color i; and (c) the edge of Q after R is the first edge of a chain of color i. Intuitively, if a maximal subpath R of Q that does not belong to T_u is preceded by an edge of color i, then R is just the path of color i that leads to the previous chain of color i in T_u. In Figure 8 solid, dashed, and dotted lines denote different colors. In the corresponding tree,

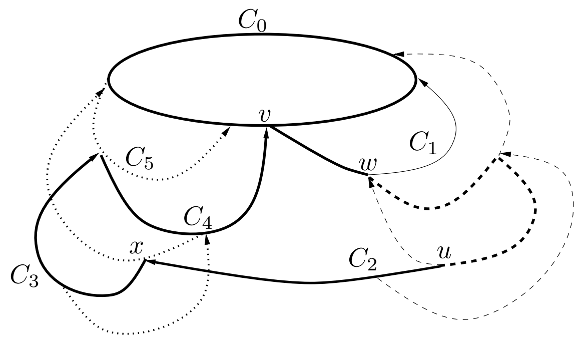


FIG. 8. *The path from x to u via v and w is $T_{v(C_1)}$-homogenous.*

the root $v(C_0)$ has two children, namely $v(C_1)$ and $v(C_5)$. Consider the path Q that starts at x, follows the solid chains to v and w, and then follows the dashed edges to u. (Path Q is shown in bold.) Path Q is a C_2-completion. It is also $T_{v(C_1)}$-homogenous because the two chains C_3 and C_4 not belonging to $T_{v(C_1)}$ have the same color as C_1 and C_2.

We try to choose *closure*(C) to be "as local to C" as possible: Let $S(C)$ be the set of explored edges when C becomes the current chain for the first time. Given $S(C)$, $a(C)$ is the lowest ancestor of $v(C)$ in T such that a $T_{a(C)}$-homogeneous completion of C exists in $S(C)$. Note that $a(C)$ is well defined since each chain has a $T_{v(C_0)}$-homogeneous completion. The path *closure*(C) is an arbitrary $T_{a(C)}$-homogeneous completion of C using only edges of $S(C)$. The algorithm can compute *closure*(C) whenever C becomes the current chain for the first time without moving the robot.

We describe the *Relocation* procedure; see Figure 9. In the relocation step, the robot repeatedly moves from the current chain to its parent until it reaches a chain C such that $T_{v(C)}$ is unfinished. To move from a chain X to its parent X', the robot proceeds along X to the endnode of X and traverses *closure*(X) to the startnode of X, which belongs to X'. When reaching C, the robot repeatedly moves from the startnode of the current chain X to the startnode of one of its children until it reaches the startnode of an unfinished chain. It chooses the child X' of X such that among all subtrees rooted at children of X and containing unfinished chains, $T_{v(X')}$ has the minimum number of tokens.

3.2. The analysis of the algorithm.

3.2.1. Correctness. Since the graph is strongly connected, all nodes of the graph must be visited during the execution of the algorithm. When the algorithm terminates, all visited nodes are finished. Thus, all edges must be explored. We show next that each operation and each move of the robot are well defined. Proposition 1 shows that if a chain of color i is fresh, then τ_i lies at the startnode of the chain. Thus, in line 10, token τ_i lies on y. By assumption, there exists a path from any finished node to s. Thus, the move in line 17 is well defined. In line 18, the robot moves to the next unfinished node of the current chain C. It would be possible to walk along *closure(C)*, but Propositon 1, part 4, shows later that *closure(C)* is not needed.

3.2.2. Fundamental properties of the algorithm.

LEMMA 1. *At most d tokens are introduced during the execution of the Balance algorithm.*

```
Procedure Relocate(C)
 1.  if all chains are finished then return(empty_chain).
 2.  else Move robot to the startnode of $C$ along $closure(C)$;
 3.      while $C \neq C_0$ and $T_{v(C)}$ is finished do
 4.          Move robot to the startnode of $parent(C)$ along $closure(parent(C))$;
 5.          $C := parent(C)$;
 6.      while $C$ is finished do
 7.          Let $C_1, C_2, \ldots, C_l$ be the chains with $parent(C_k) = C$, $1 \leq k \leq l$.
             Let $C_k$ be the chain such that $T_{v(C_k)}$ contains the smallest number
             of tokens among all $T_{v(C_1)}, \ldots, T_{v(C_l)}$ having unfinished chains;
 8.          $C := C_k$; $x :=$ startnode of $C$;
 9.          Move robot to $x$;
10.      if $C$ is not in progress then
11.          Compute $closure(C)$;
12.      return($C$)
```

FIG. 9. *The Relocation procedure.*

Proof. We say that the algorithm first introduces the token τ_k at y in line 16.

Let $in_v(v)$ and $out_v(v)$ denote the number of visited incoming and visited outgoing edges of v, respectively. Let $t(v)$ be the total number of tokens introduced on node v in line 16. We show inductively that $\max\{in_v(v) - out_v(v), 0\} = t(v)$. Since at termination $in_v(v) = in(v)$ and $out_v(v) = out(v)$, it follows that $-bal(v) \geq t(v)$ if $bal(v) < 0$ and $t(v) = 0$, otherwise. Thus, $d = -\sum_{v,with\ bal(v)<0} bal(v) \geq \sum_v t(v)$.

The claim $\max\{in_v(v) - out_v(v), 0\} = t(v)$ holds initially. Let P be the newly explored path when the first token is introduced on v, i.e., when the algorithm for the first time gets stuck at v and there is no token at v. Before P enters v, $in_v(v) = out_v(v)$. Traversing P increments $in_v(v)$ by 1 and sets $in_v(v) - out_v(v) = 1$. Thus, the claim holds. Let P be the newly explored path when the ith new token is introduced on v. It follows inductively that $in_v(v) - out_v(v) = i - 1$ before P enters v and traversing P increments the value by 1 as before. ∎

We prove next some invariants.

PROPOSITION 1.

1. *For every chain C that is in progress or that was in progress and is finished, parent(C) is finished.*
2. *Let C be a chain of color i, $1 \leq i \leq d$. (a) If C is fresh, C does not own a token, τ_i is located at the startnode of C, and parent(C) = owner(τ_i). (b) If C is in progress and not the current chain, then C is the owner of some token τ.*
3. *Every chain C is the parent of at most d chains.*
4. *If the Balance algorithm gets stuck at a node y of a chain C and y holds a token with C being the owner, then the startnode of C and all nodes of C lying between the startnode and y are finished.*

Proof (*Part* 1). Procedure *Relocate* ensures that *parent(C)* is finished before C is taken into progress.

Part 2a. When C is first created in line 12 or 15 of *Balance*, τ_i is placed on the startnode of C. Whenever the robot gets stuck at the current startnode of C and removes τ_i, chain C is extended by a path P because C is not in progress. Token τ_i is placed on the new startnode of C. Lines 13 and 16 ensure that the parent of C is always the owner of τ_i.

Part 2b. We show that whenever C is the current chain and *Balance* leaves C to continue work on another chain, C becomes the owner of a token. This suffices to prove part 2b because the children of a chain, and thus the corresponding tokens, can only be taken over by the current chain; see lines 13 and 16 of the algorithm.

Chain C is unfinished. Thus, if C is the current chain, *Balance* can only leave C to continue work on another chain during lines 5–17 of the algorithm. In this situation, *Balance* places a token on a node of C and C becomes the owner of that token.

Part 3. Chain C can become the parent of other chains while C is in progress and unfinished. During this time, every chain C' with $\mathit{parent}(C') = C$ is not in progress, see Part 1. By Part 2a, the startnode of such a chain C' holds a token and C is the owner of that token. Since there are only d tokens, the proposition follows.

Part 4. Since y holds a token, with C being the owner, y must have been the current node in a subphase when C was current chain. The node selection rule in line 18 of *Balance* ensures that the startnode of C and every node on C between the startnode and y are finished since, otherwise, the robot would have moved to an unfinished node z before y. ∎

The next lemma shows that our algorithm always balances the number of tokens contained in neighboring subtrees of T. For a subtree T_v of T, let the *weight* $w(T_v)$ be the number of tokens contained in T_v. Let $\mathit{active}(T_v) = 1$ if the current chain is in T_v; otherwise let $\mathit{active}(T_v) = 0$.

LEMMA 2. *Let $u, v \in T$ be siblings in T such that T_u and T_v contain unfinished chains. Then $|w(T_u) + \mathit{active}(T_u) - w(T_v) - \mathit{active}(T_v)| \leq 1$.*

Proof. Let $\mathit{active}(C) = 1$ iff C is the current chain, and let $\mathit{active}(C) = 0$ otherwise. Let $\mathit{token}(C)$ be the number of tokens owned by C and let $g(C) = \mathit{token}(C) + \mathit{active}(C)$. Finally, let $g(v) = \sum_{C, v(C) \in T_v} g(C) = w(T_v) + \mathit{active}(T_v)$. We show by induction on the steps of the algorithm that $|g(u) - g(v)| \leq 1$.

The claim holds initially. For a subtree T_v of T, the values $w(T_v)$ and $\mathit{active}(T_v)$ only change in lines 13, 16, and 19 of *Balance* and in lines 4 and 9 of procedure *Relocate*. Additionally, T changes in lines 10, 12, and 15.

Note first that changes in T do not affect the invariant: Whenever T changes, $v(C)$ receives a new child and C is not yet finished (or the algorithm has not yet determined that C is finished). Thus, the children of C are not yet in progress, i.e., they do not own any tokens by Proposition 1. Thus, the claim holds for any pair of children of $v(C)$.

We consider next all changes to $w(T_v)$ and $\mathit{active}(T_v)$.

Line 13: Let C be the current chain before the execution of line 13. Note that $\mathit{token}(C)$ increases by 1, $\mathit{active}(C)$ becomes 0, $\mathit{token}(C')$ decreases by 1, and $\mathit{active}(C')$ becomes 1. Thus, $g(C)$ and $g(C')$, and, hence, $g(v)$ is unchanged for every node $v \in T$.

Line 16: Note that (i) $g(C)$ is unchanged by the same argument as for line 13; (ii) $g(C')$ is unchanged, since $\mathit{token}(C')$ and $\mathit{active}(C')$ are unchanged; and (iii) $g(C_0)$ is increased by 1. Since C_0 only contributes to $g(v(C_0))$ and $v(C_0)$ is the root of T, the claim holds.

Line 19 *of Balance/line* 4 *and* 9 *of Relocate:* Let $\bar{C}$ be the current chain before the execution of line 3 or 7 and let C be the current chain afterwards. In line 3, the claim does not apply to $T_{v(C)}$, since $T_{v(C)}$ is finished. Thus, we are left with line 7. Note that $active(\bar{C})$ drops to 0 and $active(C)$ increases to 1. Thus, for every node v such that T_v contains either both the parent and its child or neither the parent nor its child, $g(v)$ is unchanged. The only remaining subtree is $T_{v(C)}$. Before the execution of line 7, for any sibling C' of C, $w(T_{v(C)}) \le w(T_{v(C')}) \le w(T_{v(C)}) + 1$. Since $active(C') = 0$, $|w(T_{v(C)}) - w(T_{v(C')}) + active(C) - active(C')| \le 1$. ☐

LEMMA 3. *Let C be a chain of color i, $1 \le i \le d$, and, at the time when C is taken in progress, let $u \in T$ be the closest ancestor of $v(C)$ that satisfies the following condition. The path from u to $v(C)$ in T contains d nodes $u_1, u_2, \ldots, u_d$ such that each u_j with $1 \le j \le d$ has a child v_j,* (a) *T_{v_j} contains a node of color i; and* (b) *$v(C) \notin T_{v_j}$. If there is no such ancestor u, then let u be $v(C_0)$. Then there exists a T_u-homogeneous C-completion.*

Proof. By assumption, the graph of explored edges is strongly connected, which implies that there exists a $T_{v(C_0)}$-homogeneous C-completion. Suppose that there are d nodes $u_1, \ldots, u_d$ satisfying (a) and (b). For $j = 1, \ldots, d$, let C_{u_j} be the chain corresponding to u_j. If one of the nodes $u_1, \ldots, u_d$, say u_k, is of color i, then there is the following T_{u_k}-homogeneous C-completion: Follow edges of color i until you reach the startnode of C_{u_k}, then walk "down" in T_{u_k} along ancestors of C to the startnode of C.

Thus, we are left with the case that none of the nodes $u_1, \ldots, u_d$ has color i. For $j = 1, \ldots, d$, let $C_{j,1} \in T_{v_j}$ be a chain of color i such that no ancestor of $C_{j,1}$ contained in T_{v_j} has color i. Let $C_{j,2}, \ldots, C_{j,l(j)}$ be the ancestors of $C_{j,1}$ in T_{u_j}. More precisely, for $k = 1, \ldots, l(j) - 1$, $C_{j,k+1} = parent(C_{j,k})$ and $C_{j,l(j)} = C_{u_j}$ is the chain corresponding to u_j.

Following the edges of color i gives a T_u-homogeneous path from C to every chain $C_{j,1}$ for $1 \le j \le d$. We want to show that there exists a T_u-homogenous path to a chain $C_{j,l(j)}$. We consider the following game on a $d \times \max_j l(j)$ grid, where for $1 \le j \le d$, square (j,k) has the color of $C_{j,k}$ for $1 \le k \le l(j)$ and no color for $k > l(j)$. Thus, all squares $(j,1)$ have color i and no other squares have color i. Initially all squares $(j,1)$ are checked; all other squares are unchecked. A square is checked if the robot can move to the startnode of the corresponding chain on a T_u-homogeneous path. The rules of the game are as follows (note that the startnode of $C_{j',k'-1}$ belongs to $C_{j',k'}$):

- *A square (j,k) of color i' gets checked whenever there exists a square (j',k') of color i' such that square $(j',k'-1)$ is checked and there exists a path of color-i' edges from the endnode of $C_{j',k'}$ to the startnode of $C_{j,k}$.*
- *The game terminates when one of the squares $(j,l(j))$ is checked or when no more squares can be checked.*

We will show that one of the squares $(j,l(j))$ can be checked. This shows that there is a T_u-homogeneous path from C to $C_{j,l(j)}$. Since u_j is an ancestor of $v(C)$, the same argument as above shows that there exists a T_u-homogeneous C-completion.

We employ the pigeonhole principle: Initially, there are d checked squares $(j,1)$ for $1 \le j \le d$ and each square $(j,2)$ has a color $i' \ne i$. Since there are at most $d-1$ other colors, there must be two squares $(s,2)$ and $(t,2)$ with the same color i'. Since the edges of color i' form a chain, there is either a path from $C_{s,2}$ to $C_{t,2}$ or vice versa. Thus, one of the two squares can be checked. Inductively, there are d checked squares $(j,k(j))$ such that $(j,k(j)+1)$ is unchecked. None of the squares $(j,k(j)+1)$ has color i and thus, there must be two squares $(j,k(j)+1)$ with the same color, which leads to checking one of the two squares. The game continues until one of the squares

$(j, l(j))$ has been checked. □

3.2.3. Counting the number of edge traversals.

LEMMA 4. *Each edge is traversed at most d times during executions of line* 17 *and at most $d+1$ times during executions of line* 18 *of the Balance algorithm.*

Proof. Let e be an arbitrary edge and let C be the chain e belongs to. Every time e is traversed during an execution of line 17, a new token is placed on the graph. Since a total of d tokens are placed, the first statement of the lemma follows.

Next we analyze executions of line 18. Let x and y be the tail and the head of e, i.e., $e = (x, y)$. Let C^1 be the portion of C that consists of the path from the startnode of C to x. Similarly, let C^2 be the path from y to the endnode of C.

By Proposition 1, part 4, e is traversed in line 18 when all nodes on C^1 are finished and the robot moves to the next unfinished node on C^2. Thus, e is traversed (a) if the robot gets stuck at a node on C^1 and moves to the next unfinished node of C, or (b) if the robot traverses C from its startnode, since procedure *Relocate* returned chain C. Every time case (a) occurs, a token is removed from C^1, and this token cannot be placed again on C^1. Whenever the robot interrupts the work on C^2, another token is placed on some node of C^2. Every time case (b) occurs, $\mathit{token}(C) + \mathit{active}(C)$ increases by 1, while no other step of the algorithm can decrease this value as long as C is unfinished. Note that a token is placed on a node of C^2. Since there are only d tokens, cases (a) and (b) occur a total of at most $d+1$ times. □

Thus, it only remains to bound how often an edge is traversed in *Relocate*. A chain C' is *dependent* on a chain C, $C \neq C'$, if $C' \in T_{v(C)}$ and $\mathit{closure}(C')$ is not T_u-homogeneous for any true descendant u of $v(C)$.

LEMMA 5. *For every chain C, there exist at most $d^{2\log d+1}$ chains $C' \in T_{v(C)}$ that are dependent on C.*

Proof. Let $n_i(C)$ be the total number of chains of color i dependent on C. For a color i, $1 \le i \le d$, and an integer δ, $1 \le \delta \le d$, let

$$N_i(\delta) = \max_C \{n_i(C); T_{v(C)} \text{ contains at most } \delta \text{ of the d tokens whenever } \mathit{active}\ (T_{v(C)}) = 1\}.$$

We will show that for any δ, $1 \le \delta \le d$, and any color i, (a) $N_i(\delta) \le d^2 N_i(\lfloor \delta/2 \rfloor)$ and (b) $N_i(1) = 1$. This implies $N_i(d) \le d^{2\log d}$. Since $\sum_{i=1}^{d} N_i(d) \le d \cdot d^{2\log d}$, the lemma follows.

To prove (1), fix a color i and an integer δ. Consider a subtree $T_{v(C)}$ that contains at most δ tokens when $\mathit{active}(T_{v(C)}) = 1$. Out of all chains of color i dependent on C, let C' be the chain whose closure is computed last. We show that when the algorithm computes $\mathit{closure}(C')$, then the number of chains of color i that are already dependent on C is at most $d(d-1)N_i(\lfloor \delta/2 \rfloor)$. Thus, $n_i(C) \le d(d-1)N_i(\lfloor \delta/2 \rfloor) + 1 \le d^2 N_i(\lfloor \delta/2 \rfloor)$.

Let $u_1, u_2, \ldots, u_l$ be the sequence of nodes (from lowest to highest) on the path from $v(C')$ to $v(C)$ such that every node u_j, $j = 1, 2, \ldots, l$, has a child v_j with (a) T_{v_j} containing a node of color i, and (b) $v(C') \notin T_{v_j}$. By Lemma 3, $l \le d$. Suppose that node u_j, $1 \le j \le l$, has $c(j)$ children, $v_{j,1}, v_{j,2}, \ldots, v_{j,c(j)}$ with $v \in T_{v_{j,1}}$. By condition (b), $2 \le c(j) \le d$.

For fixed j and $k \ge 2$, we have to show that up to the time when $\mathit{closure}(C')$ is computed, whenever $\mathit{active}(T_{v_{j,k}}) = 1$, then $w(T_{v_{j,k}}) \le \lfloor \delta/2 \rfloor$. Consider the point in time when $\mathit{closure}(C')$ is computed. Since $T_{v_{j,1}}$ contains C', $T_{v_{j,1}}$ is unfinished. By Lemma 2, *Balance* distributes the tokens contained in T_{u_j} evenly among the subtrees $T_{v_{j,1}}, T_{v_{j,2}}, \ldots, T_{v_{j,c(j)}}$ that contain unfinished chains. Thus, for each *unfinished* $T_{v_{j,k}}$

with $k \geq 2$, $w(T_{v_{j,k}})$ was up to now at most $\lfloor \delta/2 \rfloor$ whenever *active*$(T_{v_{j,k}}) = 1$. For each *finished* $T_{v_{j,k}}$, consider the last point of time when an unfinished chain of $T_{v_{j,k}}$ becomes the current chain. Since $v_{j,1}$ exists, $T_{v_{j,1}}$ is unfinished and, by Lemma 2, $w(T_{v_{j,k}})$ is up to this point in time at most $\lfloor \delta/2 \rfloor$ whenever *active*$(T_{v_{j,k}}) = 1$. We conclude that up to the time when *closure*(C') is computed, $T_{v_{j,k}}$ contains at most $N_i(\lfloor \delta/2 \rfloor)$ chains of color i that can be dependent on the chain corresponding to $v_{j,k}$, and, thus, can be dependent on C. Summing up, we obtain that $T_{v(C)}$ contains at most

$$\sum_{j=1}^{d} \sum_{k=2}^{c(j)} N_i(\lfloor \delta/2 \rfloor) \leq d(d-1)N_i(\lfloor \delta/2 \rfloor)$$

chains of color i that can be dependent on C.

Finally we show that $N_i(1) = 1$. If a subtree $T_{v(C)}$ contains at most one token whenever *active*$(T_{v(C)}) = 1$, then each node in $T_{v(C)}$ has only one child, by Proposition 1. Since $T_{v(C)}$ never branches, it can contain at most one chain of color i that is dependent on C. ∎

Lemma 6. *For every chain C, there exist at most $d^{2\log d+1}$ chains $C' \in T_{v(C)}$ such that closure(C') uses edges of C.*

Proof. Let C be an arbitrary chain and let $v \in T$ be the node corresponding to C. We show that if a chain $C' \in T_{v(C)}$ is not dependent on C, then *closure*(C') does not use edges of C. Lemma 6 follows immediately from Lemma 5.

If a chain $C' \in T_{v(C)}$ is not dependent on C, then the path *closure*(C') is T_u-homogeneous for a descendant u of v. Suppose that a T_u-homogeneous path P would use edges of C. Let i be the color of C. Chain C does not belong to T_u. Thus, after P has visited C, it may only traverse chains of color i until it reaches again a chain of color i that belongs to T_u. Note that all chains of color i that are reachable from C via edges of color i must have been generated earlier than C. However, all chains in T_u were generated later than C. We conclude that a T_u-homogeneous path cannot use edges of C. ∎

Lemma 7. *For every chain C, there exist at most $(d+2)d^{2\log d+2}$ chains $C' \notin T_{v(C)}$ such that closure(C') uses edges of C.*

Proof. A chain C' *needs* a chain C if *closure*(C') uses edges of C and C' is *u-hard* if *closure*(C') is T_u-homogeneous, but not T_v-homogeneous for any child v of u. For each chain C' there exists a unique node u of T such that C' is u-hard. If C' is dependent on chain C, then C' is $v(C)$-hard of u-hard for a true ancestor u of $v(C)$. If C' is u-hard and v is a descendant of u and an ancestor of $v(C')$, then C' is dependent on $C(v)$. To prove the lemma, it suffices to show the following two claims.

Claim 1. *There are at most $d^{2\log d+2}$ chains $C' \notin T_{v(C)}$ such that C' needs C and C' is u-hard for some ancestor u of $v(C)$.*

Claim 2. *There are at most $(d+1)d^{2\log d+2}$ chains $C' \notin T_{v(C)}$ such that C' needs C and C' is u-hard for some node u that is not an ancestor of $v(C)$.*

Proof of Claim 1. If C' needs C, then C' either does not yet exist or is unfinished when C is taken into progress. Consider the point in time when C is taken into progress. Let $u_1, u_2, \ldots, u_l$ be the ancestors of $v(C)$ in T that fulfill the following conditions: Each node u_j has a child v_j such that (a) T_{v_j} contains unfinished chains, and (b) $v(C) \notin T_{v_j}$. Thus, every chain that needs C lies in one of the subtrees T_{v_j}. Note that $l \leq d$, since by Proposition 1, every subtree that contains an unfinished chain not equal to the current chain must own a token. Assume C' belongs to T_{v_j}. Since u_j is the least common ancestor of $v(C)$ and $v(C')$, and C' is u-hard for an ancestor

u of $v(C)$, C' is dependent on $C(u_j)$. Since by Lemma 5 there are at most $d^{2\log d+1}$ chains that are dependent on $C(u_j)$, there can be at most $l \cdot d^{2\log d+1} \leq d^{2\log d+2}$ chains $C' \notin T_{v(C)}$ that need C and are u-hard for an ancestor of $v(C)$. ☐

Proof of Claim 2. Let i be the color of C. Let us denote the concatenation of all chains of color i as the *path of color i*. Note that the path of color i introduces a linear order on the chains of color i. We say a chain C *lies between* two other chains on the path of color i if C is not equal to one of the chains and lies between them in the linear order. We define first the nearest predecessor of a chain. Then we show (1) that for each chain $C' \notin T_{v(C)}$ that needs C and is u-hard for some node u that is not an ancestor of $v(C)$, there exists a chain C_1 of color i such that

- C lies on the path of color i between C_1 and its nearest predecessor, and
- C_1 fulfills the conditions of Claim 1, i.e., C' needs C_1 and u is an ancestor of $v(C_1)$.

We show next (2) that there exist at most d chains C_1 of color i for which C lies on the path of color i between C_1 and its nearest predecessor. By Claim 1 and Lemma 6, for each C_1 there exist at most $(d+1)d^{2\log d+1}$ closures that are hard for an ancestor of $v(C_1)$. It follows that there are at most $d(d+1) \cdot d^{2\log d+1}$ chains C' that need C and are u-hard for some node u that is not an ancestor of $v(C)$.

Consider the point in time when C is taken into progress. Let $a(C)$ be the closest ancestor of $v(C)$ such that $T_{a(C)}$ contains a node of color i that is not equal to $v(C)$. The *nearest predecessor* of C is the chain $C' \neq C$ of color i that was taken into progress most recently in $T_{a(C)}$.

(1) The closure of C' introduces an order on the chains belonging to it. Let C_1 be the last chain of T_u before C on *closure*(C') and let C_2 be the first chain of T_u after C on *closure*(C'), i.e., C lies on the path of color i edges between C_1 and C_2. We show below that the path of color i edges between C_1 and C_2 is contained in the path of color i edges between C_1 and its nearest predecessor. This implies that C lies on the path of color i edges between C_1 and its nearest predecessor and completes the proof of (1).

Since T_u is a subtree that contains C_1 and C_2, i.e., C_1 and another chain of color i that was taken into progress before C_1, T_u also must contain the nearest predecessor of C_1. Following the path of color i edges from C_1, C_2 is the first chain of T_u that is encountered. Thus, the color i path between C_1 and C_2 is contained in the color i path between C_1 and its nearest predecessor.

(2) We want to bound the number of color i chains C_1 such that C lies on the path of color i between C_1 and its nearest predecessor. Obviously, C_1 was created after C was taken in progress (otherwise, C_1 would have been appended to C). Consider the point in time when C is taken into progress. Let $\overline{C}_1, \ldots, \overline{C}_l$ be the chains that are parents of fresh chains. All chains created afterwards must belong to $T_{v(C)}$ or to $T_{v(\overline{C}_1)}, \ldots, T_{v(\overline{C}_l)}$. Note (a) that for no color i chain in $T_{v(C)}$, C can lie on the color i path between the chain and its nearest predecessor. Note (b) that for $k = 1, \ldots, l$, only for the color i chain $C^{(k)}$ in $T_{v(\overline{C}_k)}$ created first after C was taken into progress, C can lie between $C^{(k)}$ and its nearest predecessor. The nearest predecessor of every color i chain D created later belongs to $T_{v(\overline{C}_k)}$ and was created after C. Thus, C does not lie on the color i path between D and its predecessor. Thus, at most l chains exist such that C lies on the color i path between the chain and its predecessor. By Proposition 1, $l \leq d$. ☐

THEOREM 3. *Using the Balance algorithm and assuming that when a new sink is discovered the subgraph of explored edges is strongly connected, the robot explores* *an unknown graph with deficiency d and traverses each edge at most $(d+1)^5 d^{2\log d}$ times.*

Proof. Let e be an arbitrary edge of chain C. Edge e is traversed for the first time when it is explored during an execution of line 5 of the *Balance* algorithm. By Lemma 4, it can be traversed $2d+1$ times during executions of lines 17 and 18. By Lemmas 6 and 7, e belongs to at most $d^{2\log d+1} + (d+2)d^{2\log d+2}$ paths *closure*(C'). We show that each path *closure*(C') is traversed at most $d(d+1)$ times. The path *closure*(C') is used at most d times during an execution of line 2 of *Relocate* because each time a token is removed from the finished chain C'. The path *closure*(C') can also be used at most d^2 times in line 4 of *Relocate* because each time a token is removed from the finished subtree $T_{v(C'')}$ of a child C'' of C'.

Finally, the edge e might be traversed $d(d+1)$ times in line 9 of *Relocate*. When e is traversed in line 9, then (i) either the robot had moved to C_0 after the introduction of a new token (line 16) or (ii) there exists an ancestor u of $v(C)$ with a child x such that the robot was stuck at a node in T_x and T_x is finished. Thus, by going "up" the tree T in lines 3–5, the robot reached u. Case (i) occurs at most d times. When C becomes the current chain for the first time, let $u_1, \ldots, u_l$ be the ancestors of $v(C)$ such that each u_j has a child v_j with (a) T_{v_j} containing unfinished chains, and (b) $v \notin T_{v_j}$. By Proposition 1, the nodes $u_1, \ldots, u_l$ can have a total of d children satisfying (a) and (b). Since each subtree rooted at one of these children can contain at most d tokens, case (ii) occurs at most d^2 times.

Thus, edge e is traversed at most

$$(1)\quad 1 + 2d + 1 + d(d+1)(d^{2\log d+1} + (d+2)d^{2\log d+2}) + d(d+1) \le (d+1)^5 d^{2\log d}$$

times. □

3.3. The *Complete* algorithm. In subsections 3.1 and 3.2 we assumed that the subgraph of *explored* edges is strongly connected. We used this assumption only in line 16 of algorithm *Balance*. However, all that is needed in line 16 is that the algorithm "knows" a path from y to s, i.e., the robot can reach s from y. To achieve this we define a parametrized algorithm *P-Balance*$(\mathcal{P},\ s,\ C_0)$ as follows: in addition to s and C_0, it receives as input a set $\mathcal{P}$ of paths between various nodes in the graph. It executes algorithm *Balance* as before except when the robot gets stuck at y in line 16 and there is no path of explored edges from y to s. If there exists a path X from y to s consisting of (i) a (possibly empty) subpath of explored edges, followed by (ii) a path in $\mathcal{P}$, followed by (iii) another (possibly empty) subpath of explored edges, then a *fake* edge from y to s is added to the graph and traversed to reach s. Since the fake edge does not exist in the orginial graph the robot "simulates" traversing the fake edge by traversing X. The fake edge continues to exist (and might be traversed) in the graph until the end of algorithm *P-Balance*. We show below that at most $d-1$ fake edges are added during algorithm *P-Balance*.

We execute algorithm *P-Balance* repeatedly to construct an algorithm *Complete* that assumes only that the original graph is strongly connected and makes *no* assumption about the subgraph of explored edges. We call the edges traversed during execution $i \le k$ of algorithm *P-Balance*$(\mathcal{P},\ s,\ C_0)$ *k-visited.*

We describe algorithm *Complete* in detail: initially $\mathcal{P}$ is empty and Phase 1 (see subsection 3.1) is executed to determine s and C_0. Algorithm *Complete* then repeatedly executes algorithm *P-Balance*$(\mathcal{P},\ s,\ C_0)$ on the graph until *P-Balance* terminates or until while traversing path P the robot gets stuck at a node y in line 16 and cannot reach s. In the former case algorithm *Complete* terminates, in the latter case it adds to $\mathcal{P}$ a path of k-visited edges to y from each node in the subgraph traversed during the current or an earlier execution of algorithm *P-Balance*. Next all fake edges are discarded, all edges are marked as unvisited and unexplored, and all nodes are marked as unexplored and unfinished. Then s is set to y, the cycle C_0 is set to be the path between the first and the last occurrence of y on P, and algorithm *P-Balance*($\mathcal{P}$, s, C_0) is called.

Consider execution k of algorithm *P-Balance*. A *k-path* is a concatenation of three paths A_1, A_2, and A_3 such that A_1 and A_3 are possibly empty subpaths of edges explored during execution k and A_2 is a path of $\mathcal{P}$. Note that the concatenation of a k-path with edges explored during execution k (either at the beginning or at the end of the k-path) results again in a k-path. Note further that each k-path consists of k-visited edges.

Lemma 8 shows that if *P-Balance* gets stuck at a node y in line 16 and cannot reach s, then there exists a path of k-visited edges to y from each node in the subgraph traversed during the current or an earlier execution of algorithm *P-Balance* and that y appears at least twice on P. This proves that algorithm *Complete* is well defined.

LEMMA 8. *If while traversing path P during an execution of P-Balance($\mathcal{P}$, s, C_0) the robot gets stuck in line* 16 *at a node y and cannot reach s then*

1. *each node in the subgraph traversed during an earlier execution of algorithm P-Balance($\mathcal{P}$, s, C_0) can reach y on a path of k-visited edges;*
2. *each node in the subgraph traversed during the current execution of algorithm P-Balance($\mathcal{P}$, s, C_0) can reach y on a k-path;*
3. *y is a newly discovered sink;*
4. *y appears at least twice on P.*

Proof. Parts 1, 2, and 3: We use induction on the number k of calls to algorithm *P-Balance* to show the claim. Obviously the claim holds for $k = 0$. Consider next $k > 0$. Let s_k be the sink newly discovered by execution k of algorithm *P-Balance*. We show first that each node in the subgraph traversed during an earlier execution of algorithm *P-Balance* can reach y on a path of k-visited edges. There exists a path of k-visited edges from s_{k-1} to y, since execution k started at s_{k-1}. Inductively each node in the subgraph traversed during an earlier execution can reach s_{k-1} on a path of $(k-1)$-visited edges. Thus, by transitivity of the reachability relation and since all $(k-1)$-visited edges are also k-visited, each node in the subgraph traversed during an earlier execution of algorithm *P-Balance* can reach y on a path of k-visited edges.

We show next that each node in the subgraph traversed during the current execution of algorithm *P-Balance*($\mathcal{P}$, s, C_0) can reach y on a k-path. Since y is the last node on chain P every node on P can reach y following P. Each other node in the subgraph explored during algorithm *P-Balance*($\mathcal{P}$, s, C_0) belongs to a chain $Q \neq P$. We show by induction on the number of such chains Q created during the current execution that all nodes on such a chain Q can reach s by a k-path. Since execution k started at s, s can reach y on edges explored during execution k. It follows that each node in the subgraph traversed during algorithm *P-Balance*($\mathcal{P}$, s, C_0) can reach y on a k-path.

It remains to be shown that all nodes on a chain $Q \neq P$ created during the current execution can reach s by a k-path. This holds trivially before any chain is created. Consider a path P' that is part of Q. Then the endpoint y' of P' either belongs to an already existing chain or not. If y' belongs to a chain created earlier, then inductively y' and, thus, all nodes on P' can reach s by a k-path. If y' does not belong to a chain created earlier, then there exists a path in $\mathcal{P}$ from y' to s since $P' \neq P$. Thus there is a k-path from y' to s. It follows that every node on P' can reach s by a k-path.

We are left with showing that $y = s_k$, i.e., that y is a newly discovered sink. By the above proof, (a) if y was visited by an earlier execution of algorithm *P-Balance*, then there would exist a path from y to s in $\mathcal{P}$, and (b) if y belonged to a chain $Q \neq P$ in the current execution of algorithm *P-Balance*, then there would exist a k-path from y to s. Thus, algorithm *P-Balance*($\mathcal{P}$, s, C_0) would have been able to reach s from y. It follows that y was not visited before, i.e., that y is a newly discovered sink.

Part 4. Each node has outdegree at least 1. By the proof of part 1, y does not belong to a chain $Q \neq P$. Thus all of y's outedges must belong to P, i.e., y appeared at least twice on P. $\square$

Since there are only d sinks in the graph, part 3 of the above lemma shows that at most d executions of *P-Balance*($\mathcal{P}$, s, C_0) are made. Thus it follows that algorithm *Complete* terminates.

Now let us analyze the number of edge traversals. Algorithm *P-Balance* traverses the same path that algorithm *Balance* would have traversed on the graph consisting of the original graph and all fake edges. Since each fake edge connects two sinks, it does not change the deficiency of the graph. Thus, the previous analysis shows that each edge, including each fake edge, is traversed at most $(d+1)^5 d^{2\log d}$ times. The traversal of a fake edge corresponds to at most one traversal of every nonfake edge. We show below that there are at most $d-1$ fake edges. Thus the total number of traversals per edge is at most $(d-1)(d+1)^5 d^{2\log d}$ for each execution of algorithm *P-Balance*. Since there are at most d such executions, each edge is traversed at most $(d-1)d(d+1)^5 d^{2\log d}$ times during algorithm *Complete*.

It remains to show that there are at most $d-1$ fake edges. Each fake edge in execution k increases the number $in_v(s_i)$ of visited incoming edges for a sink s_i with $i < k$ without increasing the number $out_v(s_i)$ of visited outgoing edges. Since over all sinks s_i, $i < k$, there are at most $d-1$ more incoming than outgoing edges into these sinks, there are at most $d-1$ fake edges created during execution k.

We summarize our main result in the following theorem.

THEOREM 4. *Using the Complete algorithm, the robot explores an unknown graph with deficiency d and traverses each edge at most $(d+1)^7 d^{2\log d}$ times.*

The total number of edge traversals used by our algorithm is also $O(\min\{mn, dn^2 + m\})$, where n is the number of nodes in the graph. It is not hard to show that an upper bound of $O(\min\{mn, dn^2+m\})$ is achieved by any exploration algorithm satisfying the following two properties: (1) When the robot gets stuck, it moves on a cycle-free path to some arbitrary node with new outgoing edges. (2) When the robot is not relocating, it always traverses new edges whenever possible.

We show that any exploration algorithm satisfying (1) and (2) gets stuck at most $\min\{m, dn\}$ times. The bound follows because, by property (1), at most n edges are traversed during each relocation. Obviously, a robot gets stuck at most m times. For the proof of the second bound, let $in_u(v)$ and $out_u(v)$ be the number of unvisited incoming and unvisited outgoing edges of v, respectively. Let $def(v) = \max\{0, in_u(v) - out_u(v)\}$. We show inductively that $\sum_{v\in G} def(v) \leq d$. This implies that, for every node v, whenever the robot explores the last unvisited edge out of v, there are at most d unvisited incoming edges at v. Thus the robot gets stuck at most d times at any node v. Summing over all nodes in G gives the desired bound of dn.

The inequality $\sum_{v\in G} def(v) \leq d$ holds intitially. The invariant is maintained whenever the robot relocates from a node y, where it got stuck, to some node z with

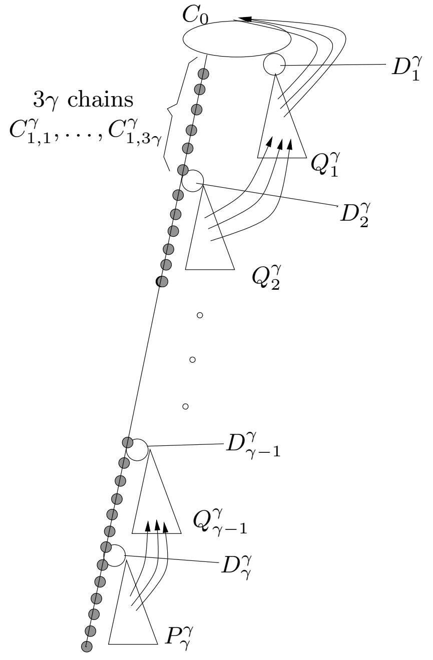


FIG. 10. *The graph G.*

new outgoing edges, because only visited edges are traversed. Whenever the robot starts a new exploration at a node z, visits a sequence of new edges, and gets stuck at a node x, *def*(z) increases by at most 1, $def(x)$ decreases by 1 while at no other node, and the *def*-value changes.

4. A tight lower bound for the *Balance* algorithm and modifications.

In this section we give first a lower bound for the *Balance* algorithm and we give afterwards lower bounds for modifications of *Balance*.

THEOREM 5. *For every $d \geq 1$, there exists a graph G of deficiency d that is explored by Balance using $d^{\Omega(\log d)} m$ edge traversals.*

Proof. We show that there exists a graph $G = (V, E)$ and an edge $e \in E$ that is traversed $d^{\Omega(\log n)}$ times while *Balance* explores G. The theorem follows by replacing e by a path of $\Theta(m)$ edges. We show the bound for d being a power of 5. The bound for all values of d follows by "rounding" down to the largest power of 5 smaller than d.

The graph is a union of chains C, each of which consists of three edges, a startnode, an endnode, and two *interior* nodes $v^1(C)$ and $v^2(C)$. The interior nodes belong to exactly one chain and have up to one additional outgoing edge. We describe G; see also Figure 10. Graph G contains (a) a cycle C_0 that starts and ends in a node v (*Balance* is started at v and finds C_0 during Phase 1) and (b) a recursively defined problem P^d attached to C_0.

In the following let δ, $1 \leq \delta \leq d$, be a power of 5. A *problem* P^δ, for any integer $\delta \geq 5$, is a subgraph that has two *incoming* edges whose startnodes do not belong to P^δ but whose endnodes do, and $\delta + 1$ *outgoing* edges whose startnodes belong to P^δ but whose endnodes do not. A problem P^1 has one incoming and one outgoing edge. In the case of P^d, the two incoming edges start at $v^1(C_0)$ and $v^2(C_0)$, respectively; d

outgoing edges point to v and one outgoing edge points to $v^1(C_0)$.

For the definition of P^δ we also need problems Q^δ. These problems are identical to P^δ except that, for $\delta > 1$, Q^δ has exactly $\delta + 1$ incoming edges.

A problem P^1 consists of a single chain; the first edge of the chain represents an incoming edge and the last edge represents an outgoing edge. The interior nodes have no additional outgoing edges. A problem Q^1 is identical to P^1.

For $\delta \geq 5$, let $\gamma = \delta/5$. Problem P^δ consists of $3\gamma^2$ chains $C^\gamma_{i,k}$, $1 \leq i \leq \gamma$, $1 \leq k \leq 3\gamma$, as well as γ chains D^γ_i and γ recursive subproblems Q^γ_i, $1 \leq i \leq \gamma - 1$, and P^γ_γ.

These components are assembled as follows. One of the incoming edges of P^δ is the first edge of $C^\gamma_{1,1}$. We assume that $v^1(C_0)$ is the startnode of $C^{d/5}_{1,1}$. Node $v^1(C^\gamma_{i,k})$ is the startnode of $C^\gamma_{i,k+1}$, $1 \leq i \leq \gamma$, $1 \leq k \leq 3\gamma - 1$. Node $v^1(C^\gamma_{i,3\gamma})$ is the startnode of $C^\gamma_{i+1,1}$, $1 \leq i \leq \gamma - 1$. The last edge of $C^\gamma_{1,k}$, $1 \leq k \leq 3\gamma$ is an outgoing edge of P^δ. The endnode of $C^\gamma_{i,k}$ is equal to the startnode of $C^\gamma_{i-1,k}$, $2 \leq i \leq \gamma$ and $1 \leq k \leq 3\gamma$. Note that the last edge of $C^\gamma_{2,1}$ is thus an outgoing edge of P^δ. Nodes $v^2(C^\gamma_{i,k})$, $1 \leq i \leq \gamma$, $1 \leq k \leq 3\gamma - 1$ have no additional outgoing edge but nodes $v^2(C^\gamma_{i,3\gamma})$, $1 \leq i \leq \gamma - 1$ do. Chain $C^\gamma_{\gamma,3\gamma}$ has no additional outgoing edges.

The second incoming edge of P^δ is the first edge of a chain D^γ_1 and, for $2 \leq i \leq \gamma$, the edge leaving $v^2(C^\gamma_{i-1,3\gamma})$ is the first edge of D^γ_i. For $1 \leq i \leq \gamma$ the last edge of D^γ_i is an outgoing edge of P^δ. If $\delta = 5$, then the first interior node of the chain $D^\gamma_i = D^\gamma_1$ has an additional outgoing edge pointing into a problem P^1. If $\delta > 5$, then the two interior nodes of D^γ_i, $1 \leq i \leq \gamma$ each have an additional outgoing edge. For $1 \leq i \leq \gamma - 1$, these two edges point into Q^γ_i and, for $i = \gamma$, they point into P^γ_γ.

If $\delta = 5$, then the outgoing edge of the only subproblem P^1 is an outgoing edge of $P^\delta = P^5$. If $\delta > 5$, the problems Q^γ_i, $1 \leq i \leq \gamma - 1$, and P^γ_γ each have $\gamma + 1$ outgoing edges. For Q^γ_1, γ of these edges are also outgoing edges of P^δ and one edge points to the interior node of D^γ_1 that is the startnode of $C^\gamma_{1,1}$. For $2 \leq i \leq \gamma - 1$, exactly $\gamma - 1$ edges leaving Q^γ_i point into Q^γ_{i-1} such that every node that has l more outgoing than incoming edges, for $l > 0$, receives l edges. One outgoing edge points to the interior nodes of D^δ_{i-1} that does not get an edge from Q^γ_{i-1} and the remaining edge points to the interior node of D^γ_i that is the startnode of $C^\gamma_{1,1}$. In the same way, the edges leaving P^γ_γ are connected with $Q^\gamma_{\gamma-1}$, $D^\gamma_{\gamma-1}$ and D^γ_γ.

We identify the sources of P^δ, i.e., the nodes having higher outdegree than indegree. At each source, outdegree and indegree differ by 1. The startnodes of the chains D^γ_i, $2 \leq i \leq \gamma$, and $C^\gamma_{\gamma,k}$, $1 \leq k \leq 3\gamma$, represent a total of $4\gamma - 1$ sources. One interior node of D^γ_γ represents a source. Finally, the subproblem P^γ_γ contains $\gamma - 1$ sources.

A problem Q^δ, $\delta \geq 5$ is the same as P^δ, except that the subproblem P^γ_γ is replaced by a problem Q^γ_γ. As mentioned before, a problem Q^δ receives $\delta - 1$ additional incoming edges. These edges point to the nodes that represent sources in P^δ.

We analyze the number of edge traversals used by *Balance* on G. Consider a problem P^δ, $\delta \geq 5$, and let $\gamma = \delta/5$. When *Balance* generates the strand of chains $C^\gamma_{i,1}, \ldots, C^\gamma_{i,3\gamma}$, for some $1 \leq i \leq \gamma$, this strand contains $3\gamma > \gamma + 1$ tokens. Since D^γ_i and the subproblem attached to it contain γ tokens *Balance* does not explore the unvisited edges out of $C^\gamma_{i,3\gamma}$ before the subproblem attached to D^γ_i is finished. In the same way we can argue for a problem Q^δ.

Let $N(\delta)$ be the number of times the following event happens while *Balance* works on a problem P^δ or Q^δ: *Balance* generates a new chain, gets stuck, and cannot reach a node with new outgoing edges by using only edges in P^δ, respectively, Q^δ. Problem P^δ contains γ subproblems $Q_1^\gamma, \ldots, Q_{\gamma-1}^\gamma$ and P_γ^γ. Every time *Balance* gets stuck in one of these subproblems and has to leave it in order to resume exploration, it also has to leave P^δ. This is because of the following facts: (1) When *Balance* explores Q_i^γ, $1 \leq i \leq \gamma - 1$, or P_γ^γ, the subproblems $Q_1^\gamma, \ldots, Q_{i-1}^\gamma$ respectively $Q_1^\gamma, \ldots, Q_{\gamma-1}^\gamma$ are already finished. (2) The chains $D_1^\gamma, \ldots, D_\gamma^\gamma$ ensure that *Balance* cannot reach any chain $C_{i,k}^\gamma$, $1 \leq i \leq \gamma$, $1 \leq k \leq 3\gamma$, from where the unfinished chains in P^δ can be reached. Again the same holds for a problem Q^δ. Thus, for $\delta \geq 5$, $N(\delta) \geq \gamma N(\gamma) = (\delta/5)N(\delta/5)$. Since $N(\delta) = 1$, for $\delta = 1$, we obtain $N(d) = d^{\Omega(\log d)}$. Finally, consider the edge e on C_0 that leaves v. *Balance* must traverse e at least $N(d) = d^{\Omega(\log d)}$ times. $\square$

We also modified the *Balance* algorithm by relocating to other nodes with new outgoing edges. Replace the choice of C_k in line 7 according to one of the following rules.

Round Robin. Let C_k be the chain among $C_1, \ldots, C_l$ that was selected least often in any execution of line 7.

Cheapest Subtree. Let C_k be the chain among $C_1, \ldots, C_l$, such that $T_{v(C_k)}$ contains the fewest number of dependent chains with respect to the current chain.

THEOREM 6. *For Round Robin and Cheapest Subtree and for all $d \geq 1$, there exist graphs of deficiency d that require $d^{\Omega(\log d)}m$ edge traversals.*

Proof. The proof is identical to that of *Generalized-Greedy* in Theorem 2. $\square$

Acknowledgments. We thank Prabhakar Raghavan for bringing to our attention the literature on the *s–t* connectivity problem. We also thank an anonymous referee for many helpful comments which improved the presentation of the paper.

REFERENCES

[1] B. Awerbuch, M. Betke, R. Rivest, and M. Singh, *Piecemeal graph learning by a mobile robot*, in Proceedings of the 8th Conference on Comput. Learning Theory, Academic Press, New York, San Diego, CA, 1995, pp. 321–328.

[2] E. Bar-Eli, P. Berman, A. Fiat, and R. Yan, *On-line navigation in a room*, J. Algorithms, 17 (1994), pp. 319–341.

[3] G. Barnes, J. F. Buss, W. L. Ruzzo, and B. Schieber, *A sublinear space, polynomial time algorithm for directed s–t connectivity*, in Proceedings of the 7th Annual Conference on Structure in Complexity Theory, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 27–33.

[4] G. Barnes and J. Edmonds, *Time-space lower bounds for directed s–t connectivity on graph automata models*, SIAM J. Comput., 27 (1998), pp. 1190–1202.

[5] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosén, and M. Saks, *Randomized robot navigation algorithms*, in Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, PA, 1996, pp. 74–84.

[6] A. Blum and P. Chalasani, *An on-line algorithm for improving performance in navigation*, in Proceedings of the 34th Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1993, pp. 2–11.

[7] A. Blum, P. Raghavan, and B. Schieber, *Navigating in unfamiliar geometric terrain*, SIAM J. Comput., 26 (1997), pp. 110–137.

[8] M. Betke, R. Rivest, and M. Singh, *Piecemeal learning of an unknown environment*, Machine Learning, 18 (1995), pp. 231–254.

[9] M. Bender and D. Slonim, *The power of team exploration: two robots can learn unlabeled directed graphs*, in Proceedings of the 35th Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alanmitos, CA, 1994, pp. 75–85.

[10] S. A. Cook and C. W. Rackoff, *Space lower bounds for maze threadability on restricted machines*, SIAM J. Comput., 9 (1980), pp. 636–652.

[11] X. Deng, T. Kameda, and C. H. Papadimitriou, *How to learn an unknown environment*, J. ACM, 45 (1998), pp. 215–245.
[12] X. Deng and C. H. Papadimitriou, *Exploring an unknown graph*, in Proceedings of the 31st Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1990, pp. 356–361.
[13] X. Deng and C. H. Papadimitriou, *Exploring an unknown graph*, revised version of [12].
[14] J. Edmonds and C. K. Poon, *A nearly optimal time-space lower bound for directed st-connectivity on the NNJAG model*, in Proceedings of the 27th Symposium on the Theory of Computing, ACM, New York, 1995, pp. 147–156.
[15] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel, *A competitive strategy for learning a polygon*, in Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, PA, 1997, pp. 166–174.
[16] E. Koutsoupias, result reported in [13].
[17] S. Kwek, *On a simple depth-first search strategy for exploring unknown graphs*, in Proceedings of the 5th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 1272, Springer-Verlag, New York, 1997, pp. 345–353.
[18] C. H. Papadimitriou and M. Yannakakis, *Shortest paths without a map*, Theoret. Comput. Sci., 84 (1991), pp. 127–150.
[19] R. Rivest, problem formulation cited in [12].

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1189–1208

REDUCIBILITY AND COMPLETENESS IN PRIVATE COMPUTATIONS[*]

JOE KILIAN[†], EYAL KUSHILEVITZ[‡], SILVIO MICALI[§], AND RAFAIL OSTROVSKY[¶]

Abstract. We define the notions of *reducibility* and *completeness* in (two-party and multiparty) private computations. Let g be an n-argument function. We say that a function f is *reducible* to a function g if n honest-but-curious players can compute the function f n-privately, given a black box for g (for which they secretly give inputs and get the result of operating g on these inputs). We say that g is *complete* (for private computations) if *every* function f is reducible to g.

In this paper, we characterize the complete boolean functions: we show that a boolean function g is complete if and only if g itself cannot be computed n-privately (when there is no black box available). Namely, for n-argument boolean functions, the notions of completeness and n-privacy are *complementary*. This characterization provides a huge collection of complete functions (*any* nonprivate boolean function!) compared to very few examples that were given (implicitly) in previous work. On the other hand, for nonboolean functions, we show that these two notions are *not* complementary.

Key words. private computation, reducibility, completeness, oblivious-transfer

AMS subject classifications. 68Q99, 68R05, 94A60, 94A62

PII. S0097539797321742

1. Introduction.

We consider (two-party and multiparty) private computations. Quite informally, given an arbitrary n-argument function f, a t-private protocol for computing f should allow n players, each possessing an individual secret input, to satisfy simultaneously the following two constraints: (1) (correctness) all players learn the value of f and (2) (privacy) no set of at most t (faulty) players learns more about the initial inputs of other players than is implicitly revealed by f's output. This problem, also known as *secure computation*, has been examined in the literature with two substantially different types of faulty players—malicious (i.e., Byzantine) players and honest-but-curious players. Below we discuss some known results with respect to each of these two types of players.

Secure computation for malicious players. Malicious players may deviate from the prescribed protocol in an arbitrary manner in order to violate the correctness and privacy constraints. The first general protocols for secure computation were given in [Yao-82, Yao-86] for the two-party case and by [GMW-87] for the multiparty case. Other solutions were given in, e.g., [GHY-87, GV-87, BGW-88, CCD-88, BB-89,

[*]Received by the editors May 20, 1997; accepted for publication (in revised form) December 29, 1998; published electronically February 10, 2000. This paper is based on two conference papers: *Completeness theorem for two-party secure computation,* in Proceedings of the 23rd Symposium on the Theory of Computing, New Orleans, LA, 1991, pp. 553–560, and *Reducibility and completeness in multi-party private computations,* in Proceedings of the 35th Symposium on Foundations of Computer Science, Santa Fe, NM, 1994, pp. 478–489.

http://www.siam.org/journals/sicomp/29-4/32174.html

[†]NEC Research Institute, 4 Independence Way, Princeton, NJ 08540 (joe@research.nj.nec.com).

[‡]Department of Computer Science, Technion, Israel (eyalk@cs.technion.ac.il). The research of this author was supported by the E. and J. Bishop Research Fund and by the Fund for the Promotion of Research at the Technion. Part of this research was done while the author was at Aiken Computation Laboratory, Harvard University, and was supported by research contracts ONR-N0001491-J-1981 and NSF-CCR-90-07677.

[§]Laboratory for Computer Science, MIT, Cambridge, MA 02139 (silvio@theory.lcs.mit.edu). The work of this author was supported by NSF grant CCR-9121466.

[¶]Bell Communication Research, MCC-1C365B, 445 South Street, Morristown, NJ 07960-6438 (rafail@bellcore.com).

Table 1.1
The number of faulty players, t, tolerable in the two basic secure-computation models (with n players).

	Honest-but-curious players	Malicious players
Computational model [Yao-82, GMW-87] (assuming trapdoor permutations exist)	$t \leq n$	$t < \frac{n}{2}$
Information-theoretic model [BGW-88, CCD-88]	$t < \frac{n}{2}$	$t < \frac{n}{3}$

RB-89, CKOR-97]; the various solutions differ from each other in the assumptions that are made—varying from intractability assumptions (such as the existence of trapdoor one-way permutations, e.g., in [GMW-87]) to physical assumptions (such as the existence of private (untappable) communication channels between each pair of players, e.g., in [BGW-88, CCD-88]). The above-mentioned solutions give t-private protocols for computing any n-argument function f for $t < \frac{n}{2}$ or $t < \frac{n}{3}$ depending on the assumption made. (See Table 1.1 for a summary of the main results.)

Secure computation for honest-but-curious players. Honest-but-curious players must always follow the protocol precisely but are allowed to "gossip" afterwards. Namely, some of the players may put together the information in their possession at the end of the execution in order to infer additional information about the original individual inputs. It should be realized that, in the case of honest-but-curious players, enforcing the *correctness* constraint is easy, and only enforcing the *privacy* constraint is hard. Honest-but-curious players are not only interesting on their own (e.g., for modeling security against outside listeners or against a passive adversary that wants to remain undetected); their importance also stems from "compiler-type" theorems, such as the one proved by [GMW-87][1] (with further extensions in many subsequent papers, for example, [BGW-88, CCD-88, RB-89]). This type of theorem provides algorithms for transforming any protocol that is t-private with respect to honest-but-curious players into a protocol that is t'-private with respect to malicious players (for some $t' \leq t$). In this paper we concentrate on honest-but-curious players.

Information-theoretic privacy. The first works on secure computation concentrated on the notion of *computational privacy*; roughly speaking, computational t-privacy requires that no coalition of t players can infer *in polynomial time* any additional information on the input of other players (that is not revealed by the output). The information-theoretic model was first examined by [BGW-88, CCD-88]; roughly speaking, information-theoretic t-privacy requires that no coalition of t players can infer any information on the input of other players, even if the players have unlimited computational power. In particular, [BGW-88, CCD-88] prove that if every two players are connected by a private channel, then every function is t-private for $t < n/2$ (with respect to honest-but-curious players; see Table 1.1). The information-theoretic model was then the subject of considerable work (e.g., [CKu-89, BB-89, CGK-90, CGK-92, KR-94, CFGN-96, KOR-96, HM-97, BW-98, KOR-98]). Particularly, [CKu-89] characterized the boolean functions for which n-private protocols exist: an n-argument boolean function f is n-private if and only if it can be represented as $f(x_1, x_2, \ldots, x_n) = f_1(x_1) \oplus f_2(x_2) \oplus \cdots \oplus f_n(x_n)$, where each f_i is boolean. Namely, f is n-private if and only if it is the exclusive-or of n local functions. An immediate

[1]The reader is referred to [G-98] for a fully detailed treatment of the [Yao-86, GMW-87] results.

corollary of this is that *most* boolean functions are *not* n-private (even with respect to honest-but-curious players).

Our contribution. We formally define the notion of *reducibility* among multiparty protocol problems. We say that a function f is reducible to a function g if there is a protocol that allows the n players to compute the value of f n-privately, in the information-theoretic sense, just by repeatedly using a black box (or a trusted party) for computing g. That is, in any round of the protocol, the players *secretly* supply arguments to the black box and then the black box *publicly* announces the result of operating g on these arguments. We stress that the only means of communication among the players is by interacting with the black box (i.e., evaluating g). For example, it is clear that *every* function f is reducible to itself (all players secretly give their private inputs $x_1, \dots, x_n$ to the black box and the black box announces the result $f(x_1, \dots, x_n)$). Naturally, we can also define the notion of *completeness.* A function g is complete if *every* function f is reducible to g. The importance of this notion relies on the following observation:

> If g is complete, and g can be computed t-privately in some "reasonable" setting[2] (such as the settings of [GMW-87, BGW-88], etc.), then any function f can be computed t-privately in the *same* setting. Moreover, from our construction a stronger result follows: if in addition the implementation of g is *efficient*, then so is the implementation of f (see below).

The above observation holds since our definition of reduction requires the highest level of privacy (i.e., n), the strongest notion of privacy (i.e., information-theoretic privacy), a simple use of g (i.e., as a black box), and that it avoid making any (physical or computational) assumptions. Hence the straightforward simulation, in which each invocation of the black box for g is replaced by an invocation of a "t-private" protocol for g, works in any "reasonable" setting (i.e., any setting that is not too weak to prevent simulation) and yields a "t-private" protocol for f in the same setting. Previously, there was no easy way to translate private protocols from one setting (such as the settings of [Yao-82, GMW-87, BGW-88, CCD-88, RB-89, FKN-94]) to other settings.

A simple observation regarding complete functions is that if g is complete, then g itself cannot be information-theoretic n-private. The inverse of this observation is the less obvious part: since the definition of completeness requires that the same function g will be used for computing *all* functions f, and since the definition of reductions seems very restrictive, it may be somewhat surprising that complete functions exist at all. Some examples of complete functions implicitly appear in the literature (without discussing the notions of reducibility and completeness). The first such results were shown in [Yao-82, GMW-87, K-88].

In this work we prove the existence of complete functions for n-party private computations. Moreover, while previous research concentrated on finding a *single* complete function, our main theorem *characterizes* all the *boolean* functions that are complete.

THEOREM 1.1 (main theorem). *For all $n \geq 2$, an n-argument boolean function g is complete if and only if g is* not *information-theoretic n-private.*

Our result thus shows a very strong dichotomy: every boolean function g is either "simple enough" so that it can be computed n-privately (in the information-theoretic

[2]A *setting* consists of defining the type of communication, type of privacy, type of players (e.g., honest-but-curious or malicious), the assumptions made, etc.

model), or it is "sufficiently expressive" so that a black box for it enables reducing every function (not only boolean) to g (i.e., g is complete). We stress that there is no restriction on g, besides being a non-n-private boolean function, and that no relation between the function g and the function f that we wish to compute is assumed. Note that using the characterization of n-private boolean functions by [CKu-89] it is easy to determine whether a given boolean function g is complete. That is, a boolean function g is complete if and only if it cannot be represented as $g(x_1, x_2, \ldots, x_n) = g_1(x_1) \oplus g_2(x_2) \oplus \cdots \oplus g_n(x_n)$, where each g_i is a boolean function.

Some features of our result. To prove the completeness of a function g as above, we present an appropriate construction with the following additional properties:

- We consider the most interesting scenario, where both the reduced function f and the black-box function g are n-argument functions (where n is the number of players). This enables us to organize the reduction in *rounds*, where in each round each player provides a value for a *single* argument of g (and the value of each argument is provided by exactly one player).[3] Thus, no player is "excluded" at any round from the evaluation of g. Our results, however, remain true even if the number of arguments of g is different from the number of arguments of f.
- Our construction evaluates the n-argument function g only on a *constant* number of n-tuples (hence, a partial implementation of g may be sufficient).
- When we talk about privacy, we put no computational restrictions on the power of the players; hence we get information-theoretic privacy. However, when we talk about protocols, we measure their *efficiency* in terms of the computational complexity of f (i.e., the size of the smallest circuit that computes f) and in terms of a confidence parameter k (our protocol allows error probability of $2^{-\Omega(k)}$). Our protocol is efficient (polynomial) in all these measures.[4] We stress, though, that the n-tuples with which we use the function g are chosen nonuniformly (namely, they are encoded in the protocol) for the particular choices of g and n (the size of the network). These n-tuples depend though neither on the size of the *inputs* to the protocol nor on the function f (or the confidence parameter k).

Our main theorem gives a full characterization of the boolean functions g that are complete (i.e., those that are not information-theoretic n-private). When nonboolean functions are considered, it turns out that the above simple characterization is no longer true. That is, we show that there are (nonboolean) functions that are not n-private, yet are *not* complete.

Overview of the proof of the main theorem. Our proof goes along the following lines:

1. We define the notion of embedded-OR for two-argument functions and appropriately generalize this notion to the case of n-argument functions. We then show that if an n-argument function is not private, then it contains an embedded-OR. For the case $n = 2$ this follows immediately from the characterization of n-private boolean functions by [CKu-89]; the case $n > 2$ requires some additional technical work.
2. We show how an embedded-OR can be used to implement an *oblivious transfer*

[3]Which player submits which argument is a permutation specified by the reduction.

[4]Evaluating g on any assignment is assumed to take unit time. All other operations (communication, computation steps, etc.) are measured in the regular way.

(OT) channel/primitive between any pair of players.[5] Finally, it follows from the work of [GMW-87, GHY-87, GV-87, K-88, BG-89, GL-90] that n-private computation of any function f can be implemented given such OT channels. All together, our main theorem follows.

Organization of the paper. In section 2 we specify our model and provide some necessary definitions. In section 3 we prove our main lemma, which shows the existence of an embedded-OR in every non-n-private, boolean function. In section 4 we use the main lemma (i.e., the existence of an embedded-OR) to implement OT channels between players. In section 5 we use the construction of OT channels to prove our main theorem. Finally, section 6 contains a discussion of the results and some open problems. For completeness, we include in the appendix a known protocol for private computations using OT channels (including its formal proof).

2. Model and definitions.

Let f be an n-argument function defined over a finite domain $\mathcal{D}$. Consider a collection of $n \geq 2$ synchronous, computationally unbounded players $P_1, \ldots, P_n$ that communicate using a black box for g, as described below. At the beginning of an execution, each player P_i has an input $x_i \in \mathcal{D}$. In addition, each player can flip unbiased and independent random coins. We denote by r_i the string of random bits flipped by P_i (sometimes we refer to the string r_i as the *random input* of P_i). The players wish to compute the value of a function $f(x_1, x_2, \ldots, x_n)$. To this end, they use a prescribed protocol $\mathcal{F}$. In the ith round of the protocol, every processor P_j secretly sends a message m_j^i to the black box g.[6] The protocol $\mathcal{F}$ specifies which argument to the black-box is provided by which player. The black box then publicly announces the result of evaluating the function g on the input messages.

Formally, with each round i the protocol associates a permutation π_i. The value computed by the black box at round i, denoted s_i, is $s_i = g(m^i_{\pi_i(1)}, m^i_{\pi_i(2)}, \ldots, m^i_{\pi_i(n)})$. Each message m_j^i, sent by P_j to the black box in the ith round, is determined by its input (i.e., x_j), its random input (i.e., r_j), and the output of the black box in previous rounds (i.e., $s_1, \ldots, s_{i-1}$). We say that the protocol $\mathcal{F}$ computes the function f if the last value (or the last sequence of values in the case of nonboolean f) announced by the black box equals the value of $f(x_1, x_2, \ldots, x_n)$ with probability $\geq 1 - 2^{-\Omega(k)}$, where k is a (confidence) parameter and the probability is over the choice of $r_1, \ldots, r_n$.

Let $\mathcal{F}$ be an n-party protocol as described above. The *communication* $\mathtt{S}(\vec{x}, \vec{r})$ is the concatenation of all messages announced by the black box while executing $\mathcal{F}$ on inputs $x_1, \ldots, x_n$ and random inputs $r_1, \ldots, r_n$. We often consider the communication $\mathbf{S}$ while fixing $\vec{x}$ and some of the r_i's; in this case, the communication should be thought of as a random variable, where each of the r_i's that were not fixed is chosen according to the corresponding probability distribution. For example, if T is a set of players, then $\mathbf{S}(\vec{x}, \langle r_i \rangle_{i \in T})$ is a random variable describing the communication when each player P_i holds input x_i, each player in T holds random input r_i, and the random inputs for all players in $\overline{T}$ are chosen randomly. The definition of privacy considers the distribution of such random variables.

[5]OT is a protocol for two players: a sender that holds two bits b_0 and b_1 and a receiver that holds a selection bit s. At the end of the protocol the receiver gets the bit b_s but has no information about the value of the other bit, while the sender has no information about the selection bit s. It should be emphasized that an OT channel in a multiparty setting has the additional requirement that listeners do not get any information; we prove, however, that this property is already implied by the basic properties of two-party OT.

[6]Notice that we do not assume private point-to-point communication among players. On the other hand, we do allow private communication between players and the black box for computing g.

Definition 2.1. *Let $\mathcal{F}$ be an n-party protocol that computes a function f and let $T \subseteq \{1, 2, \ldots, n\}$ be a set of players (coalition). We say that coalition T* does not learn any additional information *from the execution of $\mathcal{F}$ if the following holds: For every two input vectors $\vec{x}$ and $\vec{y}$ that agree on their T entries (i.e., $\forall\ i \in T : x_i = y_i$) and for which $f(\vec{x}) = f(\vec{y})$, for every choice of random inputs for the coalition's parties, $\langle r_i \rangle_{i \in T}$, and for every communication S,*

$$\Pr_{\langle r_i \rangle_{i \in \overline{T}}}(\mathbf{S}(\vec{x}, \langle r_i \rangle_{i \in T}) = S) \quad = \quad \Pr_{\langle r_i \rangle_{i \in \overline{T}}}(\mathbf{S}(\vec{y}, \langle r_i \rangle_{i \in T}) = S).$$

Informally, this definition implies that for all inputs that "look the same" from the coalition's point of view (and for which, in particular, f has the same value), the communication also "looks the same" (i.e., it is identically distributed). Therefore, by executing the protocol $\mathcal{F}$, the coalition T cannot infer any information on the inputs of $\overline{T}$ other than what follows from the inputs of T and the value of the function.

Definition 2.2. *A protocol $\mathcal{F}$ for computing f using a black box g is* t-private *if any coalition T of at most t players does not learn any additional information from the execution of the protocol. A function f is* t-private *(with respect to the black box g) if there exists a t-private protocol that uses the black box g and computes f.*

Definition 2.3. *Let g be an n-argument function. We say that the black box g (alternatively, the function g) is* complete *if every function f is n-private with respect to the black box g.*

OT is a protocol for two players $\mathcal{S}$, the *sender*, and $\mathcal{R}$, the *receiver*. It was first defined by Rabin [R-81] and since then was studied in many works (e.g., [EGL-82, W-83, FMR-85, K-88, IL-89, OVY-91]). The variant of OT protocol that we use here, which is often referred to as $\binom{2}{1}$-OT, was originally defined in [EGL-82]. It was shown equivalent to other notions of OT (see, for example, [R-81, EGL-82, BCR-86, B-86, C-87, K-88, CK-88]). The formalization of OT that we give is in terms of the probability distribution of the communication transcripts between the two players.

Definition 2.4 (OT).

Let k be a (confidence) parameter. The sender $\mathcal{S}$ initially has two bits b_0 and b_1 and the receiver $\mathcal{R}$ has a selection bit c. After the protocol completion the following holds:

- *Correctness: Receiver $\mathcal{R}$ gets the value of b_c with probability greater than $1-2^{-\Omega(k)}$, where the probability is taken over the coin tosses of $\mathcal{S}$ and $\mathcal{R}$. More formally, let $r_\mathcal{S}$, $r_\mathcal{R} \in \{0,1\}^{poly(k)}$ be the random tapes of $\mathcal{S}$ and $\mathcal{R}$ respectively, and denote the communication string by $\mathbf{comm}(\langle b_0, b_1, c\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) \in \{0,1\}^{poly(k)}$. (Again, when one (or both) of $r_\mathcal{S}, r_\mathcal{R}$ is unspecified, then* **comm** *becomes a random variable.) Then, for all k and for all $c, b_0, b_1 \in \{0,1\}$, the following holds:*

$$\Pr_{r_\mathcal{S}, r_\mathcal{R}}\left(\mathcal{R}(c, r_\mathcal{R},\ \mathbf{comm}(\langle b_0, b_1, c\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle)) = b_c\right) \quad \geq \quad 1 - \frac{1}{2^{\Omega(k)}}.$$

  *($\mathcal{R}(c, r_\mathcal{R},$ **comm**$)$ denotes the output of receiver $\mathcal{R}$ when it has a selection bit c and random input $r_\mathcal{R}$ and the communication in the protocol is* **comm***.)*

- *Sender's privacy: Receiver $\mathcal{R}$ does not get any information about b_{1-c}. (In other words, $\mathcal{R}$ has the "same view" in the case where $b_{1-c} = 0$ and in the case where $b_{1-c} = 1$). Formally, for all k, for all $c, b_c \in \{0,1\}$, for all $r_\mathcal{R}$, and for all communication comm,*

$$\begin{aligned}&\Pr_{r_\mathcal{S}}\left(\mathbf{comm}(\langle b_c, b_{1-c} = 0, c\rangle, r_\mathcal{R}) = comm\right)\\ &= \Pr_{r_\mathcal{S}}\left(\mathbf{comm}(\langle b_c, b_{1-c} = 1, c\rangle, r_\mathcal{R}) = comm\right).\end{aligned}$$

- *Receiver's privacy: Sender $\mathcal{S}$ does not get any information about c. (In other words, $\mathcal{S}$ has the "same view" in the case where $c = 0$ and in the case where $c = 1$.) Formally, for all k, for all $b_0, b_1 \in \{0, 1\}$, for all $r_{\mathcal{S}}$, and for all communication comm,*

$$\begin{aligned}&\Pr_{r_{\mathcal{R}}}\left(\mathbf{comm}(\langle b_0, b_1, c=0\rangle, r_{\mathcal{S}}) = \mathit{comm}\right)\\ &= \Pr_{r_{\mathcal{R}}}\left(\mathbf{comm}(\langle b_0, b_1, c=1\rangle, r_{\mathcal{S}}) = \mathit{comm}\right).\end{aligned}$$

Remark. We emphasize that both $\mathcal{S}$ and $\mathcal{R}$ are *honest* (but curious) and assumed to follow the protocol. When OT is defined with respect to *cheating* players, it is usually allowed that with probability $2^{-\Omega(k)}$ information will leak. This, however, is not needed for honest players.

3. A new characterization of n-private boolean functions. In this section we prove our main lemma, which establishes a new combinatorial characterization of the family of n-private boolean functions. First, we define what it means for a two-argument boolean function to contain an "embedded-OR" and use [CKu-89] to claim that any two-argument boolean function that is not 1-private contains an embedded-OR. We then generalize the definition and the claim to multiargument functions in the appropriate way.

DEFINITION 3.1. *We say that a two-argument function h contains an embedded-OR if there exist inputs x_0, x_1, y_0, y_1 ($x_0 \neq x_1$, $y_0 \neq y_1$) and an output value σ such that $h(x_1, y_1) = h(x_1, y_0) = h(x_0, y_1) = \sigma$ but $h(x_0, y_0) \neq \sigma$.*

DEFINITION 3.2. *We say that an n-argument ($n \geq 3$) function f contains an embedded-OR if there exist indices $1 \leq i < j \leq n$ and values a_k for all $k \notin \{i, j\}$ such that the two-argument function*

$$h(y, z) \overset{\triangle}{=} f(a_1, \ldots, a_{i-1}, y, a_{i+1}, \ldots, a_{j-1}, z, a_{j+1}, \ldots, a_n)$$

contains an embedded-OR.

The following facts are proven in [CKu-89].[7] Facts 1 and 2 give a characterization of the n-private boolean functions (these facts are relevant here since, as our main theorem shows, the characterization of the n-private boolean functions is also a characterization of the complete boolean functions). Facts 3 and 4 are used in [CKu-89] as technical lemmas and we use them here to prove Lemma 3.3 below.

1. An n-argument boolean function is $\lceil n/2 \rceil$-private if and only if it can be written as $f(x_1, \ldots, x_n) = f_1(x_1) \oplus \cdots \oplus f_n(x_n)$, where each f_i is boolean.
2. If an n-argument boolean function is $\lceil n/2 \rceil$-private, then it is n-private.
3. A two-argument boolean function f is not 1-private if and only if it contains an embedded-OR.
4. An n-argument boolean function f is $\lceil n/2 \rceil$-private if and only if in every partition of the indices $\{1, \ldots, n\}$ into two sets $S, \bar{S}$, each of size at most $\lceil n/2 \rceil$, the two-argument boolean function f_S defined by

$$f_S(\langle x_i \rangle_{i \in S}, \langle x_i \rangle_{i \in \bar{S}}) \overset{\triangle}{=} f(x_1, \ldots, x_n)$$

is 1-private.

[7]More specifically, the "only-if" part of fact 1 is Theorem 2 of [CKu-89]; the "if" parts of fact 1 and fact 2 is Theorem 3 of [CKu-89]; fact 3 is proved by Lemma 1 and Theorem 1 of [CKu-89]; and fact 4 is Lemma 5 of [CKu-89].

Our main lemma extends fact 3 above to the case of multiargument functions.

LEMMA 3.3 (main lemma). *Let $g(x_1, \ldots, x_n)$ be any boolean, n-argument function. The function g is not $\lceil n/2 \rceil$-private if and only if it contains an embedded-OR.*

Proof. Clearly, if g contains an embedded-OR, then there is a partition of the indices into two sets $S, \bar{S}$, each of size at most $\lceil n/2 \rceil$, such that the corresponding two-argument function g_S contains an embedded-OR (e.g., if i, j are the indices guaranteed by Definition 3.2, then include the index i in S, the index j in $\bar{S}$, and partition the other $n-2$ indices arbitrarily into two halves between S and $\bar{S}$). By fact 3, g_S is not 1-private and so, by fact 4, g itself is not $\lceil n/2 \rceil$-private.

For the other direction, since the function g is not $\lceil n/2 \rceil$-private, then, by fact 4, there is a partition $S, \bar{S}$ of the indices $\{1, \ldots, n\}$ such that the corresponding function g_S is not 1-private. For simplicity of notation, we assume that n is even and that $S = \{1, \ldots, n/2\}$. By fact 3, the two-argument function g_S contains an embedded-OR. Hence, by Definition 3.1, there exist inputs u, v, w, z and a value $\sigma \in \{0, 1\}$ that form the following structure:

$g_S(\cdot, \cdot)$	$w = w_{\frac{n}{2}+1}, \ldots, w_n$	$z = z_{\frac{n}{2}+1}, \ldots, z_n$
$u = u_1, \ldots, u_{\frac{n}{2}}$	σ	σ
$v = v_1, \ldots, v_{\frac{n}{2}}$	σ	$\bar{\sigma}$

where $u \neq v$ and $w \neq z$. To complete the proof, we will show below that it is possible to choose these four inputs so that $u_i \neq v_i$ for exactly one coordinate i and $w_j \neq z_j$ for exactly one coordinate j (this will show that g satisfies the condition of Definition 3.2). To this end, we will first show how, based on the inputs above, we can find u' and v', which differ from each other in exactly one coordinate (and such that u', v', w, z still form an embedded-OR, that is, $g(u', w) = g(u', z) = g(v', w) = \sigma$ and $g(v'z) = \bar{\sigma}$). Then, based on the new u', v' and a similar argument, we can find w', z', which differ from each other in exactly one coordinate. Again, this is done so that u', v', w', z' form the embedded-OR structure. Therefore, by using the above values of i, j and fixing all the other arguments in S to $u'_k = v'_k$ and all the other arguments in $\bar{S}$ to $w'_k = z'_k$, we get that g itself contains an embedded-OR.

Let $L \subseteq \{1, \ldots, \frac{n}{2}\}$ be the set of indices on which u and v disagree (i.e., indices k such that $u_k \neq v_k$). For every m $(0 \leq m \leq |L|)$, define T_m as the set of all vectors that can be obtained from the vector u by choosing a subset of m indices from L and replacing for each chosen index the value u_k by the value v_k (note that by the definition of L, we have $v_k \neq u_k$). In particular, $T_0 = \{u\}$, $T_{|L|} = \{v\}$, and $|T_m| = \binom{|L|}{m}$. In addition, we define the following two sets of vectors:

$$X_1 \stackrel{\triangle}{=} \{x = (x_1, \ldots, x_{n/2}) \mid g_S(x, w) = g_S(x, z)\}$$

and

$$X_2 \stackrel{\triangle}{=} \{x = (x_1, \ldots, x_{n/2}) \mid g_S(x, w) \neq g_S(x, z)\},$$

where w and z are the specific vectors we chose above. In particular, we have $u \in X_1$ and $v \in X_2$.

We now claim that there must exist u', v' as required. Namely, the vector u' is in X_1, the vector v' is in X_2, and u', v' differ in exactly one coordinate. Suppose, toward a contradiction, that this is not true, that is, for every pair u', v' that differ in exactly one coordinate, both u', v' are in the same set (X_1 or X_2). We claim that this implies that $T_m \subseteq X_1$ for all $0 \leq m \leq |L|$, contradicting the fact that v, which is in $T_{|L|}$, belongs to X_2. The proof is by induction. The claim is true for $m = 0$ since T_0 contains only the vector u, which is in X_1. Now, suppose the induction hypothesis holds for m. That is, $T_m \subseteq X_1$. For each vector x in T_{m+1}, there is a vector in T_m that differs from x in exactly one coordinate. By our assumption, this immediately implies that x is also in X_1 and hence $T_{m+1} \subseteq X_1$, as needed. Therefore, we reach a contradiction, which implies the existence of u', v' as required. That is, we found u', v' that differ in a single index i (i.e., $u'_i \neq v'_i$) and such that u', v', w, z still form an embedded-OR structure:

$g_S(\cdot,\cdot)$	$w = w_{\frac{n}{2}+1},\ldots,w_n$	$z = z_{\frac{n}{2}+1},\ldots,z_n$
$u' = u'_1,\ldots,u'_{i-1},u'_i,u'_{i+1},\ldots,u'_{\frac{n}{2}}$	σ	σ
$v' = u'_1,\ldots,u'_{i-1},v'_i,u'_{i+1},\ldots,u'_{\frac{n}{2}}$	σ	$\bar{\sigma}$

A similar argument shows the existence of w', z' that differ in a single index j and such that the above vectors u', v' together with the vectors w' and z' form an embedded-OR structure:

$g_S(\cdot,\cdot)$	$w' = w'_{\frac{n}{2}+1},\ldots,w'_{j-1},w'_j,w'_{j+1},\ldots,w'_n$	$z = w'_{\frac{n}{2}+1},\ldots,w'_{j-1},z'_j,w'_{j+1},\ldots,w'_n$
u'	σ	σ
v'	σ	$\bar{\sigma}$

This shows that g contains an embedded-OR (with indices i, j as required by Definition 3.2). ◻

4. Constructing embedded OT. The first, very simple observation is that given a black box for a function g that contains an embedded-OR, we can actually compute the OR of two bits. That is, suppose that the n players wish to compute $\text{OR}(b_k, b_\ell)$, where b_k is a bit held by player P_k and b_ℓ is a bit held by player P_ℓ. Let i, j, x_0, x_1, y_0, y_1 and $a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_{j-1},a_{j+1},\ldots,a_n$ be the indices and inputs as guaranteed by Definitions 3.1 and 3.2. Then, player P_k will provide the black box with the ith argument which is x_{b_k} (i.e., if $b_k = 0$, then the argument provided by P_k is x_0 and if $b_k = 1$, then this argument is x_1) and player P_ℓ will provide the black box with the jth argument, which is y_{b_ℓ}. The other $n-2$ players will provide the $n-2$ fixed values $a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_{j-1},a_{j+1},\ldots,a_n$. The black box will answer with the value

$$g(a_1,\ldots,a_{i-1},x_{b_k},a_{i+1},\ldots,a_{j-1},y_{b_\ell},a_{j+1},\ldots,a_n),$$

which is σ if $\text{OR}(b_k, b_\ell) = 1$ and is different from σ if $\text{OR}(b_k, b_\ell) = 0$. Hence, we have shown how to compute $\text{OR}(b_k, b_\ell)$.

Our main goal in this section is to show how, based on a black box that can compute OR, we can implement an OT protocol. We start with the two-party case ($n = 2$) and then proceed to the general case, which builds upon the two-party case.

4.1. The two-party case. In this section we show how to implement a two-party OT protocol. We start by implementing a variant of OT, called *random OT* (ROT), which is different from the standard OT (i.e., $\binom{2}{1}$-OT). In a ROT protocol the sender $\mathcal{S}$ has a bit s to be sent. At the end of the protocol, the receiver $\mathcal{R}$ gets a bit s' such that with probability $1/2$ the bit s' equals s and with probability $1/2$ the bit s' is random. The receiver knows which of the two cases happened but the sender has no idea which is the case. We start with a formal definition of the ROT primitive.

Definition 4.1 (ROT). *Let k be a (confidence) parameter. The sender $\mathcal{S}$ initially has a single input bit s (and the receiver has no input). A ROT protocol must satisfy the following four properties:*

- *Correctness. With probability greater than* $1 - 2^{-\Omega(k)}$*, receiver* $\mathcal{R}$ *outputs a pair of bits* (I, s')*, where* I *is referred to as the* indicator *(otherwise* $\mathcal{R}$ *outputs* **fail***). (As usual, the probability is taken over the coin tosses of* $\mathcal{S}$ *and* $\mathcal{R}$*, i.e.,* $r_{\mathcal{S}}$*,* $r_{\mathcal{R}} \in \{0,1\}^{poly(k)}$*.)*
- *Indicator's correctness. Whenever* $\mathcal{R}$ *outputs a pair of bits* (I, s') *such that the indicator* I *equals* 1 *(i.e.,* $\mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) = (1, s')$*), then* s' *is the correct input bit of the sender, i.e.,* $s' = s$*.*
- *Sender's privacy. If the protocol does not fail, then the probability that the receiver gets the sender's bit is exactly* $1/2$*; otherwise, the receiver gets a random bit. That is, the probability that* $\mathcal{R}$ *outputs a pair* (I, s') *such that* $I = 1$ *is exactly* $1/2$*. Formally,*

$$\begin{aligned}&\Pr_{r_{\mathcal{S}}, r_{\mathcal{R}}}(\mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) = (1, s') \mid \mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) \neq \mathbf{fail})\\ &= \Pr_{r_{\mathcal{S}}, r_{\mathcal{R}}}(\mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) = (0, s') \mid \mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) \neq \mathbf{fail})\\ &= \frac{1}{2}.\end{aligned}$$

  Moreover, if $\mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) = (0, s')$ *(i.e.,* $I = 0$*), then* s' *is random, namely,*

$$\Pr_{r_{\mathcal{S}}, r_{\mathcal{R}}}(s' = 1 | \mathcal{R}(r_{\mathcal{R}}, \mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}})) = (0, s')) \quad = \quad \frac{1}{2}.$$

- *Receiver's privacy. Sender* $\mathcal{S}$ *does not get any information about* I*, i.e., the sender does not know whether* $\mathcal{R}$ *received the bit* s *or a random bit. (In other words,* $\mathcal{S}$ *has the "same view" in the case where* $I = 0$ *and in the case where* $I = 1$*.) Formally, for all* k*, for all* $s \in \{0, 1\}$*, for all* $r_{\mathcal{S}}$*, and for all communication comm,*

$$\begin{aligned}&\Pr_{r_{\mathcal{R}}}(\mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}}) = comm \mid I = 0)\\ &= \Pr_{r_{\mathcal{R}}}(\mathbf{comm}(s, r_{\mathcal{S}}, r_{\mathcal{R}}) = comm \mid I = 1).\end{aligned}$$

Transformations of ROT protocols to $\binom{2}{1}$-OT protocols are well known [C-87].[8] Our ROT protocol is implemented, using a black box for OR, as follows:

a. The sender, $\mathcal{S}$, and the receiver, $\mathcal{R}$, repeat the following at most $m = \Theta(k)$ times:
   - $\mathcal{S}$ chooses a pair (a_1, a_2) out of the two pairs $\{(1, 0), (0, 1)\}$, each with probability $1/2$.
     $\mathcal{R}$ chooses a pair (b_1, b_2) out of the three pairs $\{(1, 0), (0, 1), (1, 1)\}$, each with probability $1/3$.
     $\mathcal{S}$ and $\mathcal{R}$ compute (using the black box) $c_1 = \mathrm{OR}(a_1, b_1)$ and $c_2 = \mathrm{OR}(a_2, b_2)$.

[8]Assume that the sender, $\mathcal{S}$, has two bits b_0, b_1 and the receiver, $\mathcal{R}$, has a selection bit c. The players $\mathcal{S}$ and $\mathcal{R}$ repeat the following for at most $m = \Theta(k)$ times: at each time, $\mathcal{S}$ tries to send to $\mathcal{R}$ a pair of random bits (s_1, s_2) using two invocations of ROT. If in both trials the receiver gets the actual bit or in both trials it gets a random bit, then they try again. If the receiver got exactly one of s_1 and s_2, it sends the sender a permutation of the indices π (i.e., either $(1, 2)$ or $(2, 1)$) such that $s_{\pi(c)}$ is known to it. The sender replies with $b_1 \oplus s_{\pi(1)}, b_2 \oplus s_{\pi(2)}$. The receiver can now retrieve the bit b_c and knows nothing about the other bit. The sender, by observing π, learns nothing about c (since it does not know from the invocation of the ROT protocols in which invocation the receiver got the actual bit and in which it got a random bit). Thus, we get a $\binom{2}{1}$-OT protocol based on the ROT protocol.

TABLE 4.1
Analyzing the six cases of (a single round of) the ROT protocol.

	$(b_1,b_2)=(1,0)$	$(b_1,b_2)=(0,1)$	$(b_1,b_2)=(1,1)$
$(a_1,a_2)=(1,0)$	$(c_1,c_2)=(1,0)$	$(c_1,c_2)=(1,1)$ $I=1$ $s'=s$	$(c_1,c_2)=(1,1)$ $I=0$ $s'=s$
$(a_1,a_2)=(0,1)$	$(c_1,c_2)=(1,1)$ $I=1$ $s'=s$	$(c_1,c_2)=(0,1)$	$(c_1,c_2)=(1,1)$ $I=0$ $s'=s\oplus 1$

- If $c_1=c_2=1$, then
  $\mathcal{S}$ sends $w=s\oplus a_1$ to $\mathcal{R}$.
  $\mathcal{R}$ outputs a pair (I,s') such that $s'=w\oplus b_2$, and $I=0$ if $(b_1,b_2)=(1,1)$ and $I=1$ otherwise.
  The protocol halts.

b. In the case that the protocol has not halted so far, $\mathcal{R}$ outputs **fail** and the protocol halts (this step is reached only if in all m times no choices (a_1,a_2) and (b_1,b_2) are such that $c_1=c_2=1$).

To analyze the protocol, we observe the following properties of it (the reader is referred to Table 4.1 for a summary of the protocol behavior in each of the six possible choices of (a_1,a_2), (b_1,b_2)):

1. If $(b_1,b_2)=(a_1,a_2)$, then one of c_1,c_2 is 0. This happens in two of the six choices of (a_1,a_2) and (b_1,b_2). In each of the other four choices we get $c_1=c_2=1$, and so at each round the protocol halts with probability 4/6. Therefore, the probability of failure in $m=\Theta(k)$ trials is exponentially small (in k), which implies the correctness property of the protocol.
2. Given that (a_1,a_2) and (b_1,b_2) are such that $c_1=c_2=1$ (which, as argued above, happens in four of the six cases), we have $(b_1,b_2)=(a_2,a_1)$ with probability 1/2 (two out of the four cases) and $(b_1,b_2)=(1,1)$ with probability 1/2. By the protocol, in the former case $\mathcal{R}$ outputs $I=1$ and in the latter $I=0$. Moreover, in the cases where $\mathcal{R}$ outputs $I=0$ (i.e., when $(b_1,b_2)=(1,1)$), each of the two choices of (a_1,a_2) is equally likely. Therefore, a_1 and hence also w and s' are random (that is, each has the value 0 with probability 1/2 and the value 1 with probability 1/2). This implies the sender's privacy property.
3. In the cases where $\mathcal{R}$ outputs $I=1$ (i.e., when $(b_1,b_2)=(a_2,a_1)$), then we have in particular $b_2=a_1$ and so $s'=w\oplus b_2=(s\oplus a_1)\oplus b_2=s$. This gives the indicator's correctness property.
4. As argued above, if the protocol does not fail, then $\mathcal{R}$ knows the "correct" value of I (since it knows the values of b_1,b_2,c_1, and c_2). The sender, on the other hand, based on (a_1,a_2), cannot know which of the two equally probable events, $(b_1,b_2)=(a_2,a_1)$ or $(b_1,b_2)=(1,1)$, happened and therefore it sees the same view whether $I=1$ or $I=0$. This gives the receiver's privacy property.

To conclude, the above four properties of the ROT protocol give the four properties required in Definition 4.1. Hence, combining the above construction (including the transformation of the ROT protocol to a $\binom{2}{1}$-OT protocol) with Lemma 3.3, we get the following lemma.

LEMMA 4.2. *An OT channel between two players is realizable given a black box g for any non-2-private function g.*

4.2. The multiparty case ($n > 2$). We have shown in our main lemma (Lemma 3.3) that any non-n-private function g contains an embedded-OR. Thus, as explained above, we can use the black box for g to compute the OR of two bits held by two players P_k and P_ℓ (where the other $n-2$ players assist by specifying the fixed arguments given by our main lemma). Then, based on the ability to compute OR, we showed in section 4.1 above how any two players can implement an OT channel between them in a way that satisfies the properties of OT (in particular, the privacy of the sender and the receiver with respect to each other). However, there is a subtle difficulty in implementing a private OT channel in a multiplayer system that we must address: besides the usual properties of an OT channel (as specified by Definition 2.4), we should guarantee that the information transmitted between the two owners of the channel will not be revealed to potential *listeners* (i.e., the other $n-2$ players). If the OT channel is implemented "physically," then clearly no information is revealed to the listeners. However, since we implement the OT protocol using a black box to some function g, which *publicly* announces each of its outcomes, we must also prove that this reveals no information to the listeners. That is, the communication seen during the execution of the OT protocol should be distributed in the same way for all values of b_1, b_2, and c.

The following lemma shows that the security of the OT protocol with respect to listeners is, in fact, already guaranteed by the basic properties of the OT protocol, namely, the security of the protocol with respect to both the receiver and the sender.

LEMMA 4.3. *Consider any (two-player) OT protocol. For every possible communication comm, the probability* $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0, b_1, c\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$ *is the same for all values* b_0 *and* b_1 *for the sender and* c *for the receiver. (In other words, a listener sees the same probability distribution of communications no matter what are the inputs held by the sender and the receiver in the OT protocol.)*

Proof. Consider the following eight probabilities corresponding to all possible values of the bits b_0, b_1, and c:

1. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 0, b_1 = 0, c = 0\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
2. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 0, b_1 = 0, c = 1\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
3. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 0, b_1 = 1, c = 0\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
4. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 0, b_1 = 1, c = 1\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
5. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 1, b_1 = 0, c = 0\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
6. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 1, b_1 = 0, c = 1\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
7. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 1, b_1 = 1, c = 0\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$,
8. $\Pr_{r_\mathcal{S}, r_\mathcal{R}}(\mathbf{comm}(\langle b_0 = 1, b_1 = 1, c = 1\rangle, \langle r_\mathcal{S}, r_\mathcal{R}\rangle) = comm)$.

The receiver's privacy property implies that the terms (1) and (2) are equal, (3) and (4) are equal, (5) and (6) are equal, and (7) and (8) are equal (since each of these four pairs of events differs only in the value of c). The sender's privacy property implies that the terms (1) and (3) are equal, (5) and (7) are equal, (2) and (6) are equal, and (4) and (8) are equal (since each of these four pairs of events differs only in the value of b_{1-c}). All together, we get that the eight probabilities are equal, as desired. □

5. A completeness theorem for multiparty boolean black-box reductions. In this section we state the main theorem and provide its proof. It is based on a protocol that can tolerate $n-1$ honest-but-curious players, assuming the existence of an OT channel between each pair of players. Such protocols appear in [GMW-87, GHY-87, GV-87, K-88, BG-89, GL-90] (these works deal also with malicious players). That is, by these works we get the following lemma. (For self-containment, both a protocol and its proof of security appear in the appendix.)

LEMMA 5.1. *Given OT channels between each pair of players, any n-argument function f can be computed n-privately (in time polynomial in the size of a boolean circuit for f).*

We are now ready to state our main theorem.

THEOREM 5.2 (main). *Let $n \geq 2$ and let g be an n-argument boolean function. The function g is complete if and only if it is* not *n-private.*

Proof.
($\Longrightarrow$) First, we show that any complete g cannot be n-private. Toward the contradiction let us assume that there exists such a function g, which is n-private and complete. This implies that all functions are n-private (as instead of using the black box g the players can evaluate g by using the n-private protocol for g). This, however, contradicts the results of [BGW-88, CKu-89] that show the existence of functions that are *not* n-private.
($\Longleftarrow$) Next (and this is where the bulk of the work is) we show how to compute any function n-privately, given a black box for any g that is not n-private. Recall that there exists a protocol that can tolerate $n-1$ honest-but-curious players, assuming the existence of OT channels (Lemma 5.1). Also, we have shown how a black box, computing *any* nonprivate function, can be used to simulate OT channels (Lemmas 4.2 and 4.3). Combining these we get the result. ∎

The theorem implies that "most" boolean functions are complete. That is, any boolean function that is not of the XOR-form of [CKu-89] is complete (see facts 1 and 2 in section 3).

6. Conclusions and further extensions.

6.1. Nonboolean functions. We have shown that *any* non-n-private boolean function g is complete. Namely, a black box for such a function g can be used for computing any function f n-privately. Let us now briefly turn our attention to nonboolean functions. First, we emphasize that if a function g contains an embedded-OR, then it is still complete even if it is nonboolean (all the arguments go through as they are; in particular note that Definitions 3.1 and 3.2 of embedded-OR apply for the nonboolean case as well). However, in the nonboolean case, not all nonprivate functions contain an embedded-OR and so their completeness does not necessarily follow.

PROPOSITION 6.1. *For every $n \geq 6$, there exists a (nonboolean) n-argument function g that is not n-private, yet such that g is* not *complete.*

Proof. In [CGK-92] it is shown that nonboolean functions form a "privacy hierarchy." That is, for every n and for every t such that $\lceil n/2 \rceil \leq t \leq n-2$, there exists a (nonboolean) n-argument function that is t-private but not $(t+1)$-private. Fix some t such that $\lceil n/2 \rceil < t \leq n-2$ (since $n \geq 6$, such t exists). By the above-mentioned result of [CGK-92], there exists an n-argument function g that is t-private and is not $(t+1)$-private (in particular, g is not n-private). However, such a function g cannot be complete since this would imply that every n-argument function is t-private which, again by [CGK-92], is not the case. ∎

The above proposition can be strengthened so that it applies to all $n \geq 2$ as follows: It is known that there are nonprivate two-argument functions that do not contain an embedded-OR. Examples of such functions were shown in [Ku-89] (see Table 6.1). We claim that with no embedded-OR one cannot compute the OR function. Assume, toward a contradiction, that there is some two-argument function f that does not have an embedded-OR, yet it could be used to compute the OR function. Since f can be used to compute the OR function, we can use it to implement OT (Lemma 4.2). Hence, there exists an implementation of OT based on some f that

Table 6.1
A nonprivate function which does not contain an embedded-OR.

	y_1	y_2	y_3
x_1	0	0	1
x_2	2	4	1
x_3	2	3	3

does not have an embedded-OR. However, [K-91] has shown that for two-argument functions, only the ones that contain an embedded-OR can be used to implement OT, deriving a contradiction. For n-argument functions ($n > 2$), notice that if we take a two-argument function g that is not 1-private and is also not complete, and we define an n-argument function $\hat{g}$ that depends only on its first two arguments by $\hat{g}(x_1, x_2, \ldots, x_n) = g(x_1, x_2)$, the resulting function $\hat{g}$ is not n-private yet it is not complete.

To conclude, we have shown that for the boolean case, the notions of completeness and privacy are *exactly complementary*, while for the nonboolean case they are *not*.

6.2. Additional remarks. In this section, we briefly discuss some possible extensions and easy generalizations of our results.

The first issue that we address is the need for the protocol to specify the permutation π_i that is used in each round i (for mapping the players to the arguments of the black box g). Note that in our construction, we use the black box only for computing the OR function on two arguments. For this, we need to map some two players P_k and P_ℓ, holding these two arguments to the special coordinates i, j guaranteed by the definition of embedded-OR. Therefore, without loss of generality, the sequence of permutations can be made oblivious (i.e., independent of the function f computed) at a price of $O(n^2)$ multiplicative factor to the rounds (and time). Moreover, at a price of $O(n^4)$ the sequence of permutations can even be made *independent* of the non-n-private function g. Finally, note that if g is a symmetric function (which is often the "interesting" case), then there is no need to permute the inputs to g.

Next, we recall the assumption that the number of arguments of g is the same as the number of arguments of f (i.e., n). Again, it follows from our constructions that this is not essential to any of our results: all that is needed is the ability for the two players P_k, P_ℓ that wish to compute the OR function in a certain step to do so by providing the two distinguished arguments i, j for g, and all the other (fixed) arguments can be provided by arbitrary players (e.g., all of them by P_1).

Finally, we note that the negative result of [CKu-89] allows a probability of error; hence, even a weaker notion of reduction that allows for errors in computing f does not change the family of complete functions. This impossibility result (i.e., first direction of the main theorem) still holds even if we allow the players to communicate not only using the black box but also using other types of communication such as point-to-point communication channels.

6.3. Open questions. The above results can be easily extended to show that any boolean g that is complete can also be used for a private computation of any *multioutput* function f (i.e., a function whose output is an n-tuple $(y_1, \ldots, y_n)$, where y_i is the output that should be given to P_i). This is so because Lemma 5.1 still holds. On the other hand, it is an interesting question to characterize the multioutput functions g that are complete (even in the boolean case where each output of g is in $\{0, 1\}$).

It is not clear how to extend the model and the results to the case of malicious players in its full generality. Notice, however, that under the appropriate definition of the model, if we are given as a black box the two-argument OR function, we can still implement private channels (see [KMO-94] for details), and hence by [BGW-88, CCD-88] can implement any f $n/3$-privately with respect to malicious players.

Suppose that we relax the notion of privacy to computational privacy (as in [Yao-82, GMW-87]). In such a case, any computationally n-private implementation of an (information-theoretically) non-n-private (equivalently, complete) boolean function g implies the existence of a one-way function. This is so since we have shown that such an implementation of g implies an implementation of OT, which in turn implies the existence of a one-way function by [IL-89]. However, the best-known implementation of such protocols for a function g as above requires *trapdoor* one-way permutations [GMW-87]. It is an important question whether there exists an implementation based on a one-way function (or permutation) for functions without a trapdoor. This question has only some partial answers. In particular, when one of the players has super-polynomial power, this is possible [OVY-91]. However, if we focus on polynomial-time players and protocols, then the result of our paper together with the work of [IR-89] implies that for all *complete* functions, if we use only black-box reductions, this is as difficult as separating $\mathcal{P}$ from $\mathcal{NP}$. Thus, using black-box reductions, *complete* functions seem to be hard to implement (with computational privacy) without a trapdoor property. Notice, however, that for nonboolean functions we have shown that there are functions that are not n-private and not complete. It is not known even if these functions can be implemented without using a trapdoor (the impossibility results of [IR-89] do not apply to this case).

Appendix. n-private protocols using embedded-OT channels. In this appendix, we present an n-private protocol that uses OT channels to compute an arbitrary n-argument function f. Our starting point is the protocol presented in [GMW-87, GHY-87, GV-87, GL-90, BG-89], which also deals with malicious players. Here we assume that players are honest. This enables us to use a simplified version of the protocol and prove Lemma 5.1.

Proof. The protocol goes as follows: given a circuit with addition and multiplication mod 2 gates that computes the function f, the players do the following. (All arithmetic operations in this protocol are modulo 2.)

1. *Sharing the inputs*: Each player P_i shares each bit $x_{i,k}$ of its input x_i by choosing, uniformly, at random a vector $(a_1^{i,k}, \ldots, a_n^{i,k})$ such that $\sum_{j=1}^{n} a_j^{i,k} = x_{i,k}$. Each such $a_j^{i,k}$ is called a *share* of the secret $x_{i,k}$. The player P_i sends the share $a_j^{i,k}$ to P_j (over their common private channel[9]).
2. *Evaluating the function*: The evaluation of the function is done in a bottom-up fashion. Each gate $c = a \circ b$ is evaluated using the shares corresponding to the inputs a and b of the gate. The evaluation of each gate ends with each player P_i holding a share c_i of the gate's output c, where the vector of shares $(c_1, \ldots, c_n)$ is uniformly distributed among the vectors whose sum is c. We distinguish between two cases according to the operation in the gate:
   - $c = a + b$: P_i computes its share of c by summing its shares of a and b, i.e., $c_i = a_i + b_i$. (No interaction is needed.)

[9]Note that, given an OT channel (as constructed in section 4.2), a private channel is easy to implement: for $\mathcal{S}$ to send a bit b to $\mathcal{R}$ it duplicates its bit twice, $\mathcal{R}$ chooses a selection bit arbitrarily, and they execute their OT protocol.

- $c = ab$: Note that $a \cdot b = (\sum_{i=1}^n a_i) \cdot (\sum_{j=1}^n b_j) = \sum_{1 \le i,j \le n} a_i \cdot b_j$. Each player P_i can compute (locally) $a_i b_i$. (However, if player P_i knew $a_i \cdot b_j$ (for $j \neq i$), it might be able to compute b_j, violating the privacy requirement.) Instead, we let P_i and P_j interact in a two-party protocol, so that at the end P_i will know $v_{i,j} \overset{\triangle}{=} (a_i \cdot b_j) - r_{i,j}$ and P_j will know $r_{i,j}$, where $r_{i,j}$ is a random bit (and so $v_{i,j} + r_{i,j}$ equals $a_i \cdot b_j$). This is done by letting P_j choose $r_{i,j}$ at random. Then, P_i receives from P_j, via their common OT channel, the a_ith element of the pair of values $((0 \cdot b_j) - r_{i,j}, (1 \cdot b_j) - r_{i,j})$ (this pair can be easily computed by P_j). Clearly, this element is exactly $(a_i \cdot b_j) - r_{i,j}$, as desired. As they use the OT channel, P_j has no idea which value P_i selected. We repeat this two-party protocol for each pair P_i, P_j. Each player computes $c_i = a_i \cdot b_i + \sum_{j \neq i} v_{i,j} + \sum_{j \neq i} r_{j,i}$. It can be verified that $c = \sum_{i=1}^n c_i$.

3. *Revealing* $f(x_1, \ldots, x_n)$. Each player P_i broadcasts its share of the output gate of the circuit. The sum of these shares is the desired value.

In the claims below, we verify (inductively) that during the computation, each vector of shares has the required sum and that the distribution in any proper subset of the shares is uniform. In addition, the interaction gives no information about previously computed shares. These properties give the correctness and privacy of the protocol.

Let $\mathbf{View}_f(T, \langle x_i \rangle, \langle R_i \rangle_{i \in T})$ denote the view that the set of players (coalition) T has on the communication in the above protocol for computing f, given that each player P_i ($1 \le i \le n$) has input x_i and that each player P_i in T has random string R_i. This is a random variable that is determined by the choice of random strings R_i for all players P_i not in T. We include in this view only messages that go from players in $\bar{T}$ to players in T. (Note that these messages together with the inputs and random strings of players in T completely define the messages sent among players in T and also messages sent from players in T to players in $\bar{T}$.) Similarly, $\mathbf{View}_C(T, \langle a_i, b_i \rangle, \langle R_i \rangle_{i \in T})$ denotes the view of T in a subprotocol evaluating a (multiplication) gate C (whose inputs a and b are shared by $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$, respectively).

In the above protocol there is no communication for addition gates. Hence the view consists only of messages received during the sharing stage, during the evaluation of multiplication gates, and during the revealing stage. The first claim says that in a single evaluation of a multiplication gate no information is revealed.

Claim A.1. *Consider the subprotocol evaluating a multiplication gate C that computes a value $c = ab$. For every coalition T, for all sets of shares $(a_1, \ldots, a_n)$ $(b_1, \ldots, b_n)$ that are the input for this subprotocol (i.e., $\{a_i, b_i\}$ is the input for player P_i), for all choices of random strings for players in T, $\langle R_i \rangle_{i \in T}$, and for all communication $comm \in \{0,1\}^s$, where $s = |T|(n - |T|)$, we have*

$$\Pr[\mathbf{View}_C(T, \langle a_i, b_i \rangle, \langle R_i \rangle_{i \in T}) = comm] = 2^{-s},$$

where the probability goes over all choices of R_i for $i \in \bar{T}$.

Proof. The communication that goes from $\bar{T}$ to T during the evaluation of a multiplication gate C is as follows: for every $i \in T$ and $j \in \bar{T}$ the players P_i, P_j jointly "compute" $a_i b_j$ and $a_j b_i$. In computing $a_j b_i$ the player P_i does not get any message (its role is to pick a random $r_{j,i}$ and to send a message over their common OT channel). In computing $a_i b_j$ player P_i receives a one-bit message ($v_{i,j}$). Hence, the view of coalition T must be of size s. Moreover, as $r_{i,j}$ is chosen (by P_j) uniformly at random, then $v_{i,j}$ is also uniformly distributed in $\{0,1\}$ (independently of what a_i and b_j are). As all $r_{i,j}$'s are independent, the claim follows. $\square$

The next claim shows that at each stage of the computation the vector of shares is uniformly distributed. This is particularly important in the revealing stage, when we need to be sure that only the output is revealed.

CLAIM A.2. *Let $x_1, \ldots, x_n$ be an input to the protocol (i.e., x_i is the input for player P_i). Let C be a gate in the circuit and let c be the value of this gate when the input for the circuit is $x_1, \ldots, x_n$. Let $\vec{C}$ be a vector of shares that represents c in the above protocol. Then,*

- *$\sum_{i=1}^{n} C_i = c$ (correctness); and*
- *$\vec{C}$ is uniformly distributed among the vectors whose sum is c (privacy), i.e., let $c_1, \ldots, c_n$ satisfy $\sum c_i = c$ (there are 2^{n-1} such vectors). Then $\Pr[\vec{C} = (c_1, \ldots, c_n)|\vec{x}] = 1/2^{n-1}$.*

Proof. The first part easily follows from the description of the protocol, by induction. The second part is also proved by induction. The claim is certainly true after the sharing stage (as this is the way the shares are chosen). Now suppose we evaluate a gate. If the gate is an addition gate, computing $C = A + B$, then

$$\begin{aligned}\Pr[\vec{C} &= (c_1, \ldots, c_n)|\vec{x}] \\ &= \sum_{a_1, \ldots, a_n; \sum a_i = A} \Pr[\vec{A} = (a_1, \ldots, a_n)|\vec{x}] \cdot \Pr[\vec{B} = (c_1 - a_1, \ldots, c_n - a_n)|\vec{x}] \\ &= 2^{n-1} \frac{1}{2^{n-1}} \frac{1}{2^{n-1}} = \frac{1}{2^{n-1}}.\end{aligned}$$

If the gate computes $C = A \cdot B$, then we can fix $\vec{A} = (a_1, \ldots, a_n)$ and $\vec{B} = (b_1, \ldots, b_n)$ and now show that for any such fixed choice $\vec{C}$ still satisfies the requirement. In particular, it suffices to show (by induction on i) that the probability that $C_1 = c_1, \ldots, C_i = c_i$ for $i \leq n-1$ is $1/2^i$. To do so, we consider the bits $r_{i,j}$ $(j \neq i)$ and $r_{j,i}$ $(j \neq i)$ and assign random values to each of them (that were not assigned values so far). At least one of those random bits (e.g., $r_{n,i}$) is still "free." This implies that c_i will be uniformly distributed (as $r_{n,i}$ is one of the summands that construct c_i). Clearly, when we consider C_n, all the random bits already have values and hence the value of C_n is already determined. $\square$

We now turn to the proof of the privacy of the whole protocol.

CLAIM A.3. *For every coalition T (of size $1 \leq |T| \leq n-1$), for all input $x_1, \ldots, x_n$, for all choices of random strings for players in T, $\langle R_i \rangle_{i \in T}$, and for all possible communication comm,*[10]

$$\Pr[\mathbf{View}_f(T, \langle x_i \rangle, \langle R_i \rangle_{i \in T}) = comm] \quad = \quad 2^{-d}$$

for $d = |T| \cdot n_{\bar{T}} + m \cdot |T| \cdot (n - |T|) + (n - |T| - 1)$, where m is the number of multiplication gates in the circuit for f and $n_{\bar{T}}$ is the number of inputs for the circuit held by players in $\bar{T}$. (Again, the probability goes over all choices of R_i for $i \in \bar{T}$.)

Proof. In the sharing stage, each player in T receives a share (a bit) from each input to the circuit held by a player in $\bar{T}$ (by definition there are $n_{\bar{T}}$ such bits). The properties of the secret sharing guarantee that each of these bits is 0 with probability 1/2 and they are all independent. The evaluation of addition gates does not involve any communication. Claim A.1 guarantees that in the evaluation of any multiplication gate, no matter what are the shares that the players start with, the view of the

[10] A communication is *possible* for $x_1, \ldots, x_n$ if it is consistent with $f(x_1, \ldots, x_n)$.

players in T consists of a random string of length $|T|(n - |T|)$. Also, note that each of these evaluations makes use of new (independent) random bits. Finally, if $f_1, \ldots, f_n$ are the shares representing the outcome of the circuit, then by Claim A.2 this vector is uniformly distributed among the vectors whose sum equals $f(x_1, \ldots, x_n)$. Therefore, the players in T get in the revealing stage $n - |T|$ bits, which form $2^{n-|T|-1}$ combinations each with equal probability. Note that if $|T| = n - 1$, then in the revealing stage the players in T get only one bit, which is uniquely determined by $\vec{x}$. However, if $|T| < n - 1$, then the independence of the communication seen in the revealing stage and the communication seen in previous stages is guaranteed by the random bits $r_{i,j}$ for $i, j \in \bar{T}$. Combining all these we get the desired claim. ∎

COROLLARY A.4. *For every coalition T (of size $1 \leq |T| \leq n-1$), for all inputs $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ such that $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ and such that $x_i = y_i$ for all $i \in T$, for all choices of random strings for players in T, $\langle R_i \rangle_{i \in T}$, and for all communication comm,*

$$\Pr[\mathbf{View}_f(T, \langle x_i \rangle, \langle R_i \rangle_{i \in T}) = comm] \quad = \quad \Pr[\mathbf{View}_f(T, \langle y_i \rangle, \langle R_i \rangle_{i \in T}) = comm],$$

where the probability goes over all choices of R_i for $i \in \bar{T}$.

This completes the proof of Lemma 5.1. ∎

Acknowledgments. We wish to thank Oded Goldreich for helpful discussions and very useful comments. We thank Mihir Bellare for pointing out to us in 1991 that the works of Chor, Kushilevitz, and Kilian are complementary and thus imply a special case of our general result. Finally, we thank Amos Beimel and the anonymous referees for very helpful comments.

REFERENCES

[BB-89] J. Bar-Ilan and D. Beaver, *Non-cryptographic fault-tolerant computing in a constant number of rounds*, in Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, 1989, pp. 201–209.

[BGW-88] M. Ben-or, S. Goldwasser, and A. Wigderson, *Completeness theorems for non-cryptographic fault-tolerant distributed computation*, in Proceedings of the 20th Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 1–10.

[B-86] M. Blum, *Applications of Oblivious Transfer*, manuscript.

[BCC-88] G. Brassard, D. Chaum, and C. Crépeau, *Minimum disclosure proofs of knowledge*, J. Comput. System Sci., 37 (1988), pp. 156–189.

[BCR-86] G. Brassard, C. Crépeau, and J.-M. Robert, *Information theoretic reductions among disclosure problems*, in Proceedings of the 27th Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, 1986 pp. 168–173.

[BG-89] D. Beaver and S. Goldwasser, *Multiparty computation with faulty majority*, in Proceedings of the 30th Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 468–473.

[BW-98] D. Beaver and A. Wool, *Quorum-based secure multi-party computation*, EuroCrypt (1998), in Lecture Notes in Comput. Sci. 1403, Springer-Verlag, New York, pp. 375–390.

[CFGN-96] R. Canetti, U. Feige, O. Goldreich, and M. Naor, *Adaptively secure multi-party computation*, in Proceedings of the 28th Symposium on the Theory of Computing, Philadelphia, PA, 1996, pp. 639–648.

[CKOR-97] R. Canetti, E. Kushilevitz, R. Ostrovsky, and A. Rosén, *Randomness vs. fault-tolerance*, in Proceedings of the 16th Symposium on Principles of Distributed Computing, Santa Barbara, CA, 1997, pp. 35–44.

[CCD-88] D. Chaum, C. Crepeau, and I. Damgard, *Multiparty unconditionally secure protocols*, in Proceedings of the 20th Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 11–19.

[CKu-89] B. Chor and E. Kushilevitz *A Zero-one law for Boolean privacy*, SIAM J. Discrete Math., 4 (1991), pp. 36–47.

[CGK-90] B. Chor, M. Geréb-Graus, and E. Kushilevitz, *Private computations over the integers*, SIAM J. Comput., 24 (1995), pp. 376–386.

[CGK-92] B. Chor, M. Geréb-Graus, and E. Kushilevitz, *On the structure of the privacy hierarchy*, J. Cryptology, 7 (1994), pp. 53–60.

[C-87] C. Crépeau, *Equivalence between two flavors of oblivious transfer*, Crypto (1987), in Lecture Notes. in Comput. Sci. 293, Springer-Verlag, New York, pp. 350–354.

[CK-88] C. Crépeau and J. Kilian *Achieving oblivious transfer using weakened security assumptions*, in Proceedings of the 29th Symposium on Foundations of Computer Science, White Plains, NY, 1988, pp. 42–52.

[EGL-82] S. Even, O. Goldreich, and A. Lempel, *A randomized protocol for signing contracts*, Comm. ACM, 28 (1985), pp. 637–647.

[FKN-94] U. Feige, J. Kilian, and M. Naor, *A minimal model for secure computation*, in Proceedings of the 26th Symposium on the Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 554–563.

[FMR-85] M. Fischer, S. Micali, and C. Rackoff, *An Oblivious Transfer Protocol Equivalent to Factoring*, manuscript.

[GHY-87] Z. Galil, S, Haber, and M. Yung, *Cryptographic computation: Secure fault-tolerant protocols and the public-key model*, Crypto (1987), in Lecture Notes in Comput. Sci. 293, Springer-Verlag, New York, pp. 135–155.

[G-98] O. Goldreich, *Secure Multi-Party Computation*, manuscript, 1998. Available from ftp://theory.lcs.mit.edu/pub/people/oded/prot.ps.

[GMW-87] O. Goldreich, S. Micali, and A. Wigderson, *How to play any mental game*, in Proceedings of the 19th Symposium on the Theory of Computing, New York, 1987, pp. 218–229.

[GV-87] O. Goldreich and R. Vainish, *How to solve any protocol problem—An efficiency improvement*, Crypto (1987), in Lecture Notes in Comput. Sci. 293, Springer-Verlag, New York, pp. 73–86.

[GL-90] S. Goldwasser and L. Levin, *Fair computation of general functions in presence of immoral majority*, Crypto (1990), in Lecture Notes in Comput. Sci. 537, Springer-Verlag, New York, pp. 77–93.

[GMR-85] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof-systems*, in Proceedings of the 17th Symposium on the Theory of Computing, 1985, pp. 291–304.

[HM-97] M. Hirt and U. Maurer, *Complete characterization of adversaries tolerable in secure multi-party computation*, in Proceedings of the 16th ACM Symposium on Principles of Distributed Computing, Santa Barbara, CA, 1997.

[IL-89] R. Impagliazzo and M. Luby, *One-way functions are essential for complexity-based cryptography*, in Proceedings of the 30th Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 230–235.

[IR-89] R. Impagliazzo and S. Rudich, *On the limitations of certain one-way permutations*, in Proceedings of the 21st Symposium on the Theory of Computing, 1989, pp. 44–61.

[K-88] J. Kilian, *Basing cryptography on oblivious transfer*, in Proceedings of the 20th Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 20–31.

[K-91] J. Kilian, *Completeness theorem for two-party secure computation*, in Proceedings of the 23rd Symposium on the Theory of Computing, New Orleans, LA, 1991, pp. 553–560.

[Ku-89] E. Kushilevitz, *Privacy and communication complexity*, SIAM J. Disc. Math., 5 (1992), pp. 273–284.

[KOR-96] E. Kushilevitz, R. Ostrovsky, and A. Rosén, *Characterizing linear size circuits in terms of privacy*, J. Comput. System Sci., 58 (1999), pp. 129–136.

[KOR-98] E. Kushilevitz, R. Ostrovsky, and A. Rosén, *Amortizing randomness in private multiparty computations*, in Proceedings of the 17th ACM Symposium on Principles of Distributed Computing, Puerto Vallarta, Mexico, 1998, pp. 81–90.

[KMO-94] E. Kushilevitz, S. Micali and R. Ostrovsky, *Reducibility and completeness in multi-party private computations*, in Proceedings of the 35th Symposium on Foundations of Computer Science, Santa Fe, NM, 1994, pp. 478–489.

[KR-94] E. Kushilevitz and A. Rosén, *A randomness-rounds tradeoff in private computation*, SIAM J. Discrete Math., 11 (1998), pp. 61–80.

[OVY-91] R. Ostrovsky, R. Venkatesan, and M. Yung. *Fair games against an all-powerful adversary*, extended abstract, in Proceedings of Sequences '91, Positano, Italy. See also DIMACS Ser. Discrete Math. Theoret. Comput. Sci., 13 (1993), pp. 155–169.

[RB-89] T. Rabin and M. Ben-Or, *Verifiable secret sharing and multiparty protocols with honest* *majority*, in Proceedings of the 21st Symposium on the Theory of Computing, Seattle, WA, 1989, pp. 73–85.

[R-81] M. Rabin, *How to Exchange Secrets by Oblivious Transfer*, Tech. Report TR-81, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1981.

[W-83] S. Weisner, *Conjugate coding*, SIGACT News, 15 (1983), pp. 78–88.

[Yao-82] A.C. Yao, *Protocols for secure computations*, in Proceedings of the 23th Symposium on Foundations of Computer Science, Chicago, IL, 1982, pp. 160–164.

[Yao-86] A.C. Yao *How to generate and exchange secrets*, in Proceedings of the 27th Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, 1986, pp. 162–167.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1209–1222

LOWER BOUNDS FOR ($\mathrm{MOD}_p - \mathrm{MOD}_m$) CIRCUITS[*]

VINCE GROLMUSZ[†] AND GÁBOR TARDOS[‡]

Abstract. Modular gates are known to be immune for the random restriction techniques of Ajtai (1983), Furst, Saxe, and Sipser (1984), Yao (1985), and Håstad (1986). We demonstrate here a random clustering technique which overcomes this difficulty and is capable of proving generalizations of several known modular circuit lower bounds of Barrington, Straubing, and Thérien (1990), Krause and Pudlák (1994), and others, characterizing symmetric functions computable by small ($\mathrm{MOD}_p, \mathrm{AND}_t, \mathrm{MOD}_m$) circuits.

Applying a degree-decreasing technique together with random restriction methods for the AND gates at the bottom level, we also prove a hard special case of the constant degree hypothesis of Barrington, Straubing, and Thérien (1990) and other related lower bounds for certain ($\mathrm{MOD}_p, \mathrm{MOD}_m, \mathrm{AND}$) circuits.

Most of the previous lower bounds on circuits with modular gates used special definitions of the modular gates (i.e., the gate outputs one if the sum of its inputs is divisible by m or is *not* divisible by m) and were not valid for more general MOD_m gates. Our methods are applicable, and our lower bounds are valid for the most general modular gates as well.

Key words. lower bounds, modular gates, composite modulus

AMS subject classifications. 68Q05, 68Q15, 68Q22

PII. S0097539798340850

1. Introduction.

Boolean circuits are perhaps the most widely examined models of computation. They gain application in diverse areas as VLSI design, complexity theory, and the theory of parallel computation.

A majority of the strongest and deepest lower bound results for computational complexity were proved using the Boolean circuit model of computation (for example, [12], [16], [8], [13], [14], or see [4] for a survey).

Unfortunately, lots of questions, even for very restricted circuit classes, have been unsolved for a long time.

Bounded depth and polynomial size is a natural restriction. Ajtai [1] and Furst, Saxe, and Sipser [6] proved that no polynomial sized, constant-depth circuit can compute the PARITY function. Yao [16] and Håstad [8] generalized this result for sublogarithmic depths. Their technique involved a sophisticated use of *random restriction techniques*, in which randomly assigned 0-1 values to the input variables fixed the output of large fan-in AND and OR Boolean gates.

Since the modular gates are very simple to define, and they are immune to the random restriction techniques in lower bound proofs for the PARITY function, the following natural question was asked by Barrington, Smolensky and others: How powerful will the Boolean circuits be if, beside the standard AND, OR, and NOT gates, MOD_m gates are also allowed in the circuit? Here, a MOD_m^A gate outputs 1 if the sum of its inputs is in a set $A \subset \{0, 1, 2, \ldots, m-1\}$ modulo m.

[*]Received by the editors June 24, 1998; accepted for publication (in revised form) May 18, 1999; published electronically February 10, 2000. This work was supported in part by grants OTKA F014919, FKFP 0835, and AKP 97-56 2,1.

http://www.siam.org/journals/sicomp/29-4/34085.html

[†]Department of Computer Science, Eötvös University, Rákóczi út 5, H-1088 Budapest, Hungary (grolmusz@cs.elte.hu).

[‡]Rényi Institute of the Hungarian Academy of Science, Reáltanoda u. 13-15, H-1055 Budapest, Hungary (tardos@cs.elte.hu).

Razborov [13] showed that for computing MAJORITY with AND, OR, NOT, and MOD_2 gates, exponential size is needed with constant depth. This result was generalized by Smolensky [14] for MOD_p gates instead of MOD_2 gates, where p denotes a prime.

We know very little, however, if both MOD_p and MOD_q gates are allowed in the circuit for different primes p, q, or if the modulus is a nonprime power composite, e.g., 6. For example, it is consistent with our present knowledge that depth-3, linear-sized circuits with MOD_6 gates *only* recognize an NP-complete language (see [2]).

It is not difficult to see that constant-depth circuits with MOD_p gates only (p prime) cannot compute even very simple functions—the n-fan-in OR or AND functions—since they can compute only constant degree polynomials of the input variables over GF_p (see [14]).

But depth-2 circuits with MOD_2 and MOD_3 gates or MOD_6 gates can compute the n-fan-in OR and AND functions [9], [2]. Consequently, these circuits are more powerful than circuits with MOD_p gates only. The sketch of the construction is as follows: we take a MOD_3 gate at the top of the circuit and 2^n MOD_2 gates on the next level, where each subset of the n input variables is connected to exactly one MOD_2 gate, and then this circuit computes the n-fan-in OR, since if at least one of the inputs is 1, then exactly half of the MOD_2 gates evaluate to 1.

Barrington, Straubing, and Thérien [2] conjectured that any $(\mathrm{MOD}_p^B, \mathrm{MOD}_m^A, \mathrm{AND}_d)$ circuit needs exponential size to compute the n-fan-in AND function, where the prime p and the positive integers m and d are fixed and AND_d denotes the fan-in d AND function. They called it the *constant degree hypothesis* (CDH) and proved the $d = 1$ case, with highly nontrivial algebraic techniques. Their proof also works for depth-$(\ell + 1)$

$$(\overbrace{\mathrm{MOD}_{p^k}^B, \mathrm{MOD}_{p^k}^B, \ldots, \mathrm{MOD}_{p^k}^B}^{\ell}, \mathrm{MOD}_m^A) \tag{1.1}$$

circuits, computing the AND function.

Yan and Parberry [15], using Fourier-analysis, also proved the $d = 1$ case for $(\mathrm{MOD}_p^{\{1,2,\ldots,p-1\}}, \mathrm{MOD}_2^{\{1\}})$ circuits, but their method also works for the special case of the CDH where the sum of the degrees of the monomials g_i on the input-level satisfies

$$\sum_{\deg(g_i)\geq 1} (\deg(g_i) - 1) \leq \frac{n}{2(p-1)} - O(1).$$

Krause and Waack [11] applied communication-complexity techniques to show that any $(\mathrm{MOD}_m^{\{1,2,\ldots,m-1\}}, \mathrm{SYMMETRIC})$ circuit, computing the ID function

$$\mathrm{ID}(x, y) = \begin{cases} 1 \text{ if } x = y, \\ 0 \text{ otherwise} \end{cases}$$

for $x, y \in \{0, 1\}^n$, should have size at least $2^n / \log m$, where SYMMETRIC is a gate, computing an arbitrary symmetric Boolean function. Since (non-weighted) MOD_m gates are also SYMMETRIC gates, this lower bound is valid for $(\mathrm{MOD}_m^{\{1,2,\ldots,m-1\}}, \mathrm{MOD}_m^A)$ circuits. When mod m coefficients (or multiple wires) are allowed on the input-level, then the MOD_m gates are *not* SYMMETRIC gates, but the same proof techniques remain applicable. Caussinus [5] proved that the result of

[2] also implies a similar lower bound for the AND function. Unfortunately, results [11], [5] do not generalize for the more general MOD_m^A gates at the top.

Krause and Pudlák [10] proved that any $(\mathrm{MOD}_{p^k}^{\{0\}}, \mathrm{MOD}_q^{\{0\}})$ circuit which computes the $\mathrm{MOD}_r^{\{0\}}$ function has size at least 2^{cn}, for some $c > 0$, where p and r are different primes and q is not divisible by either of them.

Our main result is a characterization of those symmetric Boolean functions which are computable by quasi-polynomial-size

$$(\overbrace{\mathrm{MOD}_{p^k}^B, \mathrm{MOD}_{p^k}^B, \ldots, \mathrm{MOD}_{p^k}^B}^{\ell}, \mathrm{MOD}_m^A)$$

circuits. We prove (Theorem 2.5) that the *only* symmetric functions that are computable by such circuits are the MOD_{mp^j} functions with small j. Consequently, the nontrivial threshold functions (and thus also AND and OR) and the $\mathrm{MOD}_r^{\{0\}}$ functions if r does not divide $p^j m$ need exponential size on that circuits. Even MOD_4 requires exponential size $(\mathrm{MOD}_{3^r}, \mathrm{AND}_t, \mathrm{MOD}_2)$ circuits for constant t and r. Note the asymmetry: MOD_4 is easy to compute with a polynomial size $(\mathrm{MOD}_2, \mathrm{AND}_3)$ circuit. These results generalize the theorems of Barrington, Straubing, and Thérien [2] and Krause and Pudlák [10] and give a characterization of the computable symmetric functions, instead of singular lower bounds.

Grolmusz [7] generalized the results of [2], [15], [11], [10] for $(\mathrm{MOD}_q, \mathrm{MOD}_p, \mathrm{AND}_{cn})$ circuits, where the input-polynomials of each MOD_p gate are constructible from linear terms using at most $cn - 1$ multiplications (or, equivalently, can be computed by an arithmetic circuit of an arbitrary number of mod p additions and at most $cn-1$ fan-in 2 multiplications). In particular, one can allow the sum of *an arbitrary function* of cn variables and a linear polynomial of the n variables as inputs for each MOD_p gate. We generalize this result, too (Lemma 3.12). The main tool of the proof of [7] is a degree decreasing lemma, which we also generalize here for nonprime moduli (Lemma 3.9), and we use it both for lower and upper bound proofs.

Here we generalize the results of [7]: we prove a lower bound on the size of the $(\mathrm{MOD}_p, \mathrm{MOD}_m, \mathrm{AND})$ circuits computing AND_n if m is a positive integer, p is a prime, and each MOD_m gate has not-too-many AND gates as inputs and those AND gates have low fan-in. For the exact statement see Theorem 2.6. This is an important special case of the CDH of [2]. The lower bound also applies to circuits computing some other functions besides AND.

2. Our results.

2.1. Ideas. MOD_m gates are immune to random restriction techniques, since these gates remain MOD_m gates on the remaining variables after an arbitrary restriction, and thus (unless less than m variables remain unrestricted) the complexity does not decrease.

We overcome this difficulty by a *random clustering* technique, which forces some randomly chosen variables to be equal. Each equivalence class (or cluster) will make a new variable of the MOD_m gate, and each new variable will be invisible (i.e., its coefficient will be a multiple of m) for the gate with a constant probability (Lemma 3.4).

We use this for $(\mathrm{MOD}_p, \mathrm{AND}_t, \mathrm{MOD}_m)$ circuits, computing symmetric functions. Suppose that the equivalence classes are of size m; then the resulting function of the new, clustered variables is a unique symmetric function.

Almost all symmetric functions (except the $\mathrm{MOD}_{p^k m}$ functions) have large restrictions, whose unique factor resulting from the clustering above cannot be expressed as a modulo p sum of functions, none of which depends on all variables. An exponential lower bound follows for the number of AND gates on level 2 (Theorem 2.4).

If we have $o(n^2/\log n)$ constant-degree monomials as inputs for each MOD_m gates on level 2, then by random restrictions, one can essentially decrease their number, and a small number of low-degree monomials can be converted to linear polynomials with the help of the degree-decreasing lemma (Lemma 3.9), and we can apply Theorem 2.4 to get lower bounds. (Theorem 2.6)

2.2. Preliminaries.

Definition 2.1. *A fan-in n gate is an n-variable Boolean function. Let $G_1, G_2, \ldots, G_\ell$ be gates of unbounded fan-in. Then a $(G_1, G_2, \ldots, G_\ell)$-circuit denotes a depth-ℓ circuit with a G_1-gate on the top, G_2 gates on the second level, G_3 gates on the third level from the top,..., and G_ℓ gates on the last level.* AND_t *denotes the fan-in t* AND *gate. The size of a circuit is defined to be the total number of the gates in the circuit.*

All of our modular gates are of unbounded fan-in, and we allow for connecting inputs to gates or gates to gates with multiple wires. Note that, by this definition, our modular gates are not symmetric gates in general.

In the literature MOD_m gates are sometimes defined to be 1 iff the sum of their inputs is divisible by m, and sometimes they are defined to be 1 iff the sum of their inputs is not divisible by m. The following, more general definition covers both cases.

Definition 2.2. *We say that gate G is a* MOD_m-gate *if there exists $A \subset \{0, 1, \ldots, m-1\}$ such that*

$$G(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \textit{if } \sum_{i=1}^n x_i \bmod m \in A, \\ 0 & \textit{otherwise.} \end{cases}$$

A is called the 1*-set of G.* MOD_m *gates with* 1*-set A are denoted by* MOD_m^A.

Notation 2.3. *Let* $\Sigma_p(x_1, x_2, \ldots, x_s) = \sum_{i=1}^s x_i \bmod p$.

In general, Σ_p is not a Boolean gate, since its value is from $\{0, 1, \ldots, p-1\}$. However, in all of our statements, its value will be guaranteed to be 0 or 1.

2.3. Theorems. Here we list the three main results of this paper. To be concise we use $((\mathrm{MOD}_{p^k}^B)^\ell, \mathrm{MOD}_m^A)$ to denote circuits of type (1.1). Note that standard techniques (see Lemma 3.2) show that these circuits are equivalent to $(\sum_p, \mathrm{AND}_t, \mathrm{MOD}_m^A)$ circuits, and we could have stated Theorems 2.4 and 2.5 for those circuits instead.

Theorem 2.4. *Suppose that a circuit of type $((\mathrm{MOD}_{p^k}^B)^\ell, \mathrm{MOD}_m^A)$ with p prime computes a symmetric Boolean function f on n variables, such that $f \neq \mathrm{MOD}_{p^j m}^A$ for any A. Then its size S is exponential in p^j; i.e., there exists a number $c > 1$ depending on p, m, k, and ℓ such that $S > c^{p^j}$.*

As a special case we get that the size S of an n-variable circuit of type $((\mathrm{MOD}_{p^k}^B)^\ell, \mathrm{MOD}_m^A)$ with p prime computing any of the nontrivial threshold functions (including AND *and* OR*) or the $\mathrm{MOD}_r^{\{0\}}$ function (where r does not divide mp^j for any j) is exponential in n. We have $S > c^n$ for a number $c > 1$ depending only on p, m, k, and ℓ.*

Theorem 2.5. *Let the prime p and the positive integers m, k, and ℓ be fixed with m not a power of p. The symmetric functions computed by a type $((\mathrm{MOD}_{p^k}^B)^\ell, \mathrm{MOD}_m^A)$*

circuit of quasi-polynomial size are exactly the functions $\mathrm{MOD}^C_{mp^j}$ *with* $j = O(\log\log n)$ *and* $C \subset \{0, 1, \ldots, mp^j - 1\}$.

On the other hand, all the functions $\mathrm{MOD}^C_{mp^j}$ *with* $j = O(\log\log n)$ *can be computed by quasi-polynomial size* $(\sum_p, \mathrm{AND}_2, \mathrm{MOD}_m)$ *circuits.*

Our final result proves a special case of the CDH.

THEOREM 2.6. *Let* p *be prime and* m *a fixed positive integer. Suppose that a* $(\mathrm{MOD}^B_p, \mathrm{MOD}^A_m, \mathrm{AND})$ *circuit computes* AND_n. *If each* MOD_m *gate has fan-in* $o(n^2/\log n)$ *and each* AND *gate has constant fan-in, then the size of the circuit is super-polynomial.*

We remark that this result is a consequence of the tradeoff between the size of $(\Sigma_p, \mathrm{MOD}_m, \mathrm{AND})$ circuits computing AND and a new measure introduced here, the number of pairs of input variables the MOD_m gates relate (see Theorems 3.13 and 3.14). Note that similar bounds can be proved for circuits computing many other natural functions, like threshold or MOD_r functions.

3. The proofs.

3.1. Eliminating the top gate.

The top-gate elimination is widely used in the literature (cf. [10, Lemma 5.2] or [3]). It replaces the top MOD_{p^r} gate with constant fan-in AND gates and a simple summation modulo p with a polynomial increase in the size.

LEMMA 3.1. *Let* p *be a prime,* k *a positive integer, and* $A \subset \{0, 1, \ldots, p^k - 1\}$. *There is a modulo* p *polynomial of degree* $p^k - 1$ *computing the* $\mathrm{MOD}^A_{p^k}$ *function.* □

One can repeatedly use this lemma to eliminate a constant-depth subcircuit of MOD_{p^r} gates from the top of any circuit, as stated by the next lemma.

LEMMA 3.2. *Suppose that* $f : \{0, 1\}^n \to \{0, 1\}$ *is computed by a depth-*$(\ell + 1)$

$$(\overbrace{\mathrm{MOD}^A_{p^k}, \ldots, \mathrm{MOD}^A_{p^k}}^{\ell}, G)$$

circuit, where p *is a prime and on the input level we have arbitrary gates (or subcircuits)* G. *Suppose the number of these gates* G *is* S. *Then* f *can also be computed from the same gates* G *by a* $(\Sigma_p, \mathrm{AND}_t, G)$ *circuit, with* $t < p^{k\ell}$ *and at most* $S^{p^{k\ell}}$ AND_t *gates on the middle level.*

Proof. By Lemma 3.1 all $\mathrm{MOD}^A_{p^k}$ can be replaced by a modulo p polynomial of degree less than p^k; thus f is degree $< p^{k\ell}$ polynomial of the output of the G gates. The bound on the size comes from counting all the possible monomials in such a polynomial. □

Note that the size of the new circuit is still polynomial in S, and the fan-in of the AND gates is constant if the depth ℓ and the modulus p^k are constants. Note also that AND_t gates with $t < p^k$ can be considered as special MOD_{p^k} gates, and thus AND_t gates can be eliminated the same way.

3.2. Random clustering.

DEFINITION 3.3. *Let* $\sim$ *be an equivalence relation on the variables of a function* f. *By the factor* $f/\!\sim$ *of* f *we mean the function obtained from* f *by identifying variables according to* $\sim$. *The variables of* $f/\!\sim$ *correspond to the equivalence classes of* $\sim$. *For an integer* m *we call the* $f/\!\sim$ *an* m*-factor of* f *if each equivalence class in* $\sim$ *consists of* m *variables.*

We say that the Boolean function f is p-simple (p is a positive integer) if it can be expressed as a modulo p sum of functions, none of which depend on all of the variables.

Example. Suppose that f has six variables, and $x_1{\sim}x_2, x_3{\sim}x_4, x_5{\sim}x_6$. Then $f/{\sim}$ is a 2-factor of f, has three variables, and is defined as

$$f/{\sim}(y_1, y_2, y_3) = f(y_1, y_1, y_2, y_2, y_3, y_3).$$

Notice that any factor of the AND function is again an AND function. The m-factor of a symmetric function is unique and it is also a symmetric function. Note that for prime numbers p a function f is p-simple iff it can be expressed as a modulo p polynomial of degree less than the number of its variables.

Implicitly, a random clustering technique was used in the paper of Krause and Pudlák [10]. However, our method gives stronger results more directly.

The following lemma is about a special type of three level circuits. It is stated in a more general way, but the reader may think of polynomial size, $(\sum_p, \mathrm{AND}_t, \mathrm{MOD}_m^A)$ circuits with constant t.

LEMMA 3.4. *Let p, m, and t be positive integers, $1 \geq \varepsilon > 0$, and suppose the Boolean function f on n variables satisfies $f \equiv \sum_{i=1}^{S} f_i \pmod p$, where each f_i is computed in an arbitrary way from t of the functions f_{ij} and from $(1-\varepsilon)n$ of the input variables. Each of the functions f_{ij} is in turn a modulo m linear combination of the input variables. Here the functions f_i output modulo p values while f_{ij} output modulo m values. If n is large enough and divisible by m and $S < c^n$, then there exists a p-simple m-factor of f, where the constant $c > 1$ depends only on m, t, and ε.*

Proof. The idea is to observe that $f_{ij}/{\sim}$ is a modulo m linear combination of its variables, and the coefficient of a variable, corresponding to an equivalence class in a random $\sim$, is equal to zero with a positive constant probability. Thus $f_i/{\sim}$ depends on all of its variables with exponentially small probability. Then, with high probability, all the functions $f_i/{\sim}$ have an invisible variable, and thus $f/{\sim}$ is p-simple.

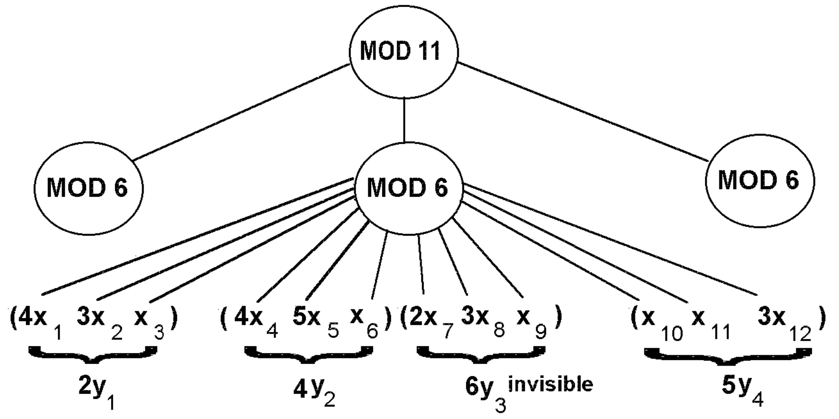


FIG. 3.1. *Random clustering in the simplest case: $t = 1, \varepsilon = 1$, and every f_{i1} is a* MOD_6 *gate.*

Let us choose $\sim$ uniformly at random from all the partitions of the variables into classes of size m. Consider choosing the equivalence classes one by one. Consider a fixed $1 \leq i \leq S$ and one of the first $\lceil \varepsilon n/(2m) \rceil$ classes. When we choose the variables of this class, there are less than $\varepsilon n/2$ variables already in previous classes and at most $(1-\varepsilon)n$ variables directly seen by f_i, so the set H of the remaining variables has at least $\varepsilon n/2$ elements. Each variable in H has a coefficient in each of the combinations f_{ij}. Let $(a_j)_{j=1}^t$ be a list of coefficients that are most popular, and call a variable in H good if its coefficient in f_{ij} is a_j for each value of j. There are at least $|H|/m^t \geq \epsilon n/(2m^t)$ good variables. When choosing the variables for our equivalence class each has a probability of at least $\epsilon/(2m^t)$ to be good. Despite the slight dependence among these events, the probability that each of them are good is still at least $(\epsilon/(4m^t))^m$ for large enough n. If this is the case, $f_i/{\sim}$ does not depend on the variable corresponding to this class, since f_i does not see it directly, and the coefficient of this variable in f_{ij} is $ma_j = 0 \bmod m$. Thus (using $(1-u)^k < e^{-uk}$) with probability at most $e^{-(\varepsilon n/(2m))\cdot(\varepsilon m^{-t}/4)^m}$ does $f_i/{\sim}$ depend on each of its variables. We choose $\ln c = (\varepsilon/4)^{m+1}/m^{tm+1}$. If $S < c^n$, then with positive probability none of the functions $f_i/{\sim}$ depend on all of the variables; consequently, $f/{\sim} \equiv \sum_{i=1}^S f_i/{\sim}$ $(\mathrm{mod}\ p)$ is p-simple. ∎

We remark here that the same proof gives that if S in the lemma is bounded by another exponential function of n, then a random m-factor of f can almost always be expressed as a modulo p sum of functions; none of the functions depends on an m^{-mt} fraction of their variables.

NOTATION 3.5. *Let $w(x)$ denote the weight of a zero-one vector x, i.e., the number of ones in x. Then $f(i)$ denotes the value of the symmetric Boolean function f on inputs of weight i.*

LEMMA 3.6. *Let p be a prime. If f is a symmetric Boolean function on p^k variables with $f(0) \neq f(p^k)$, then f is not p-simple.*

Proof. Notice that

$$\sum_{x\in\{0,1\}^n} (-1)^{w(x)} f(x) \equiv 0 \pmod p$$

for p-simple functions f. The left-hand side is zero for functions not depending on one of the input variables; thus it is divisible by p for a modulo p sum of such functions.

For a symmetric function on $n = p^k$ variables, the left-hand side of the last equation is

$$\sum_{i=0}^{n} (-1)^i \binom{n}{i} f(i) \equiv f(0) - f(n) \pmod p,$$

since p divides $\binom{p^k}{i}$ unless $i = 0$ or $i = p^k$. Thus $f(0) \neq f(n)$ implies that f has full p-degree as claimed. ∎

THEOREM 3.7. *Let p be a prime, m, t, k, and S positive integers, and $1 \geq \varepsilon > 0$. Suppose the symmetric Boolean function f on n variables is the modulo p sum of S of the functions f_i, where each of the f_i is computed in an arbitrary way from t of the functions f_{ij} and from $(1-\varepsilon)n$ of the input variables. Each of the functions f_{ij} is in turn a modulo m linear combination of the input variables. Here the functions f_i output modulo p values while f_{ij} output modulo m values. Suppose f is not equal to any MOD_{mp^k} gate. Then $S > c^{p^k}$ for a constant $c > 1$ depending only on m, t, and ε.*

Proof. Since f is not a MOD_{mp^k} gate, there exist numbers $0 \leq i < i + mp^k = j \leq n$ such that $f(i) \neq f(j)$. Restrict the function f by assigning 0 to $n - j$ of its variables and assigning 1 to i of them. The resulting function f' is a symmetric function of its mp^k variables satisfying $f'(0) \neq f'(mp^k)$. Notice that the restriction does not increase the size of the circuit computing the function. The unique m-factor of f' is a symmetric function f'' on p^k variables satisfying $f''(0) \neq f''(p^k)$. By Lemma 3.6, f'' is not p-simple. Thus Lemma 3.4 gives the claimed bound on S. ∎

We are ready now to prove Theorem 2.4.

Proof of Theorem 2.4. We apply Lemma 3.2 to get rid of the MOD_{p^k} gates and get a $(\sum_p, \mathrm{AND}_t, \mathrm{MOD}_m)$ circuit for our symmetric function. The size of the circuit blows up polynomially, i.e., it is bounded by S^b, where b and t depend on p, m, k, and ℓ. Then Theorem 3.7 bounds S. Notice that we did not use the feature of Theorem 3.7 that the middle gates can directly depend on many input variables.

The statement on the specific functions follows from the observation that every function mentioned there satisfies that it is not of the form $\mathrm{MOD}^A_{mp^j}$ unless $mp^j > n$. ∎

The following lemma nicely complements Theorem 3.7.

LEMMA 3.8. *Consider the Boolean function* $f(x_1, x_2, \ldots, x_n) = \mathrm{MOD}^A_{mp^k}(x_1, x_2, \ldots, x_n)$. *If m is not a power of the prime p, then f can be computed by a* $(\sum_p, \mathrm{AND}_2, \mathrm{MOD}_m)$ *circuit of size at most* $(mn)^{2p^{k'}}$, *where $p^{k'}$ is the largest power of p dividing mp^k.*

Notice that the assumption that m is not a power of p is necessary. Otherwise, if $m = p^\ell$, arbitrary size constant depth circuits of constant fan-in AND and arbitrary MOD_p and MOD_m gates could only compute Boolean functions expressible as constant degree modulo p polynomials, and that constant degree does not depend on k. Consequently, it cannot compute f, which is a degree-$(p^k - 1)$ polynomial.

Proof. Suppose first that all elements of the 1-set A are congruent to a single number a modulo m. There is a degree $p^{k'} - 1$ polynomial on the input computing $\mathrm{MOD}^A_{p^{k'}}$ modulo p (Lemma 3.1). This polynomial can be implemented by a modulo p sum of AND gates of at most $p^{k'} - 1$ variables. The number of AND gates is bounded by $n^{p^{k'}-1}$. Let q be prime factor of m different from p, and place a redundant $\mathrm{MOD}^{\{1\}}_q$ gate above each AND gate. Apply the degree decreasing lemma (Lemma 3.9) to replace each AND gate by a collection of at most $(2q)^{p^{k'}-2}$ MOD_q gates summing to the same value modulo p. First replace each MOD_q gate by a MOD_m gate computing the same function, then replace each MOD_m gate G by the AND of G and the $\mathrm{MOD}^{\{a\}}_m$ gate on all the inputs. The resulting circuit computes the AND of the $\mathrm{MOD}^{\{a\}}_m$ and the $\mathrm{MOD}^A_{p^{k'}}$ functions; thus it computes the $\mathrm{MOD}^A_{mp^k}$ function as desired.

To remove our assumption on A, notice that every set A can be decomposed into m sets A_i satisfying this assumption. The equation $\mathrm{MOD}^A_{mp^k} = \sum_i \mathrm{MOD}^{A_i}_{mp^k}$ proves the lemma. ∎

Consider the smallest $(\sum_p, \mathrm{AND}_t, \mathrm{MOD}_m)$ circuit computing the function $\mathrm{MOD}^{\{0\}}_{mp^j}$, and notice that the lower bound on the circuit size for this function in Theorem 3.7 is c^{p^j}, while the upper bound in Lemma 3.8 is $n^{c'p^j}$. The gap is too wide to characterize polynomial size circuits, but we can characterize quasi-polynomial-size circuits as in Theorem 2.5.

Proof of Theorem 2.5. Apply Lemma 3.2 as in Theorem 2.4 to eliminate the MOD_{p^k} gates. Use Theorem 3.7 and Lemma 3.8 to get the two sides of the characterization. ∎

3.3. The degree-decreasing lemma. Lemma 3.9 exploits a surprising property of (MOD$_s$, MOD$_m$)-circuits, which (MOD$_p$, MOD$_p$) circuits lack, since constant-depth circuits with MOD$_p$ gates and arbitrary size are only capable of computing constant-degree modulo p polynomials of the input. Here we generalize the original version [7] of the degree-decreasing lemma for nonprime moduli.

LEMMA 3.9 (degree-decreasing lemma). *Let p be a prime and $s, m > 1$ be integers, satisfying* $\gcd(s,p) = \gcd(s,m) = 1$. *Let x_1, x_2, x_3 be variables taking values from $\{0,1,\dots,p-1\}$, $x_1' \in \{0,1\}$. Then*

$$\mathrm{MOD}_p^A(x_1x_2 + x_3) \equiv H_0 + H_1 + \cdots + H_{p-1} + \beta \pmod{s}, \tag{3.1}$$

$$\mathrm{MOD}_m^A(x_1'x_2 + x_3) \equiv H_0' + H_1' + \beta' \pmod{s}, \tag{3.2}$$

where H_i abbreviates

$$H_i = \alpha \sum_{j=0}^{p-1} \mathrm{MOD}_p^A(ix_2 + x_3 + j(x_1 + (p-i)))$$

for $i = 0,1,\dots,p-1$; α is the multiplicative inverse of p modulo s: $\alpha p \equiv 1 \pmod{s}$; β is a positive integer satisfying $\beta = -|A|(p-1)\alpha \bmod s$; H_i' abbreviates

$$H_i' = \alpha' \sum_{j=0}^{m-1} \mathrm{MOD}_m^A(ix_2 + x_3 + j(x_1' + (m-i)))$$

for $i = 0,1$; and α' is the multiplicative inverse of m modulo s: $\alpha' m \equiv 1 \pmod{s}$; and β' is a positive integer satisfying $\beta' = -|A|\alpha \bmod s$.

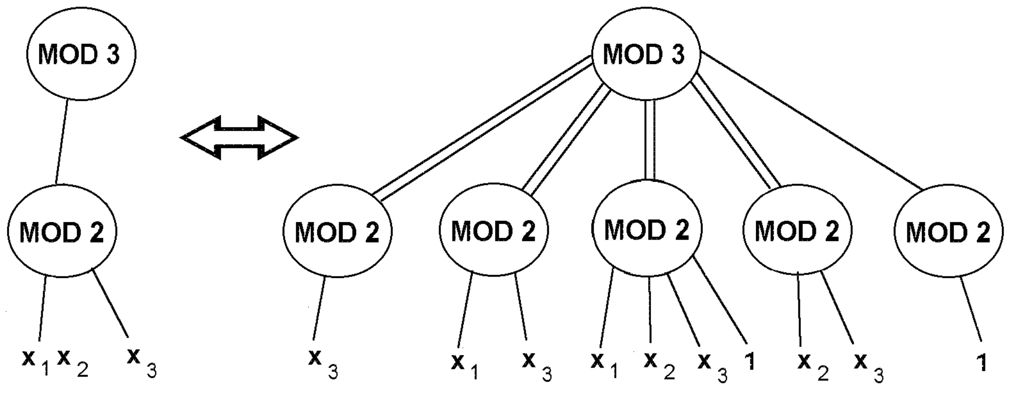


FIG. 3.2. *Degree decreasing in* (MOD$_3$, MOD$_2^{\{1\}}$) *case. On the left the input is a degree-2 polynomial, and on the right the input consists of linear polynomials.*

Proof. Let $x_1 = k$, and let $0 \le i \le p-1$, $k \ne i$. Then

$$H_k = \alpha \sum_{j=0}^{p-1} \mathrm{MOD}_p^A(kx_2 + x_3) = \alpha p \mathrm{MOD}_p^A(kx_2 + x_3) \equiv \mathrm{MOD}_p^A(x_1x_2 + x_3) \pmod{s}$$

and

$$H_i = \alpha \sum_{j=0}^{p-1} \mathrm{MOD}_p^A(ix_2 + x_3 + j(k-i)) = \alpha|A|,$$

since for any fixed x_2, x_3, i, k expression $kx_2+x_3+j(k-i)$ takes on every value exactly once modulo p while $j = 0, 1, \dots, p-1$; so $\mathrm{MOD}_p^A(ix_2+x_3+j(k-i))$ equals 1 exactly $|A|$ times. Consequently,

$$H_0 + H_1 + \cdots + H_{p-1} + \beta \equiv \mathrm{MOD}_p^A(x_1x_2+x_3) + (p-1)\alpha|A| + \beta \equiv \mathrm{MOD}_p^A(x_1x_2+x_3) \pmod{s}.$$

Similarly, let $x_1' = k \in \{0,1\}$, and let $i \in \{0,1\}$, $k \neq i$. Then

$$H_k' = \alpha' \sum_{j=0}^{m-1} \mathrm{MOD}_m^A(kx_2+x_3) = \alpha' m \mathrm{MOD}_m^A(kx_2+x_3) \equiv \mathrm{MOD}_p^A(x_1'x_2+x_3) \pmod{s}$$

and

$$H_i' = \alpha' \sum_{j=0}^{m-1} \mathrm{MOD}_m^A(ix_2 + x_3 + j(k-i)) = \alpha'|A|,$$

since for any fixed x_2, x_3, i, k, for $i \neq k$ $|i-k| = 1$, so expression $kx_2 + x_3 + j(k-i)$ takes on every value exactly once modulo m while $j = 0, 1, \dots, m-1$; so $\mathrm{MOD}_m^A(ix_2 + x_3 + j(k-i))$ equals 1 exactly $|A|$ times. Consequently,

$$H_0' + H_1' + \beta' \equiv \mathrm{MOD}_m^A(x_1'x_2+x_3) + \alpha'|A| + \beta' \equiv \mathrm{MOD}_p^A(x_1'x_2+x_3) \pmod{s}. \qquad \square$$

3.4. Random restriction. The CDH of [2] states that any $(\sum_p, \mathrm{MOD}_m, \mathrm{AND}_d)$ circuit computing AND has superpolynomial size if p is a prime and m and d are constants. We make progress toward this statement by proving Theorem 2.6 stating that AND requires superpolynomial size circuits of this type, if each MOD_m gate has fan-in $o(n^2/\log n)$. A stronger form of this statement (see Theorem 3.13) can be based on the following definition.

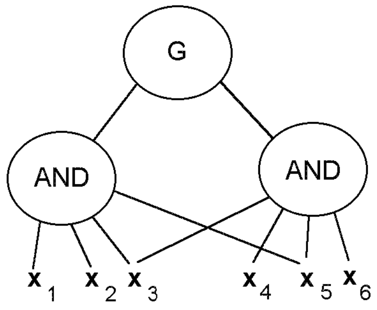


FIG. 3.3. *Gate G relates, e.g., x_1 and x_2 or x_3 and x_6 but does not relate x_1 and x_4.*

DEFINITION 3.10. *Let G be a gate of a circuit on the second level from the inputs computing some function of* AND*'s of variables. We say that G relates two input variables if they appear as inputs in a common* AND *gate below G.*

*We say that a gate G is H-*linear *if H is a subset of the input-variables, such that G does not relate two input variables outside H; i.e., the input of G is linear in the variables outside H with coefficients that are arbitrary functions of the variables in H. We call a gate ε-*linear *if it is H-linear with a set H containing at most an ε-fraction of all variables.*

We start with a simple application of the degree decreasing lemma (Lemma 3.9).

LEMMA 3.11. *Let p and m be relatively prime integers, and consider an n variable Boolean function f computed by a* (MOD_m^B, AND) *circuit, where the top* MOD_m^B *gate is H-linear. Then f can be computed by a* ($\sum_p$, MOD_m) *circuit with* $(2m)^{|H|}$ MOD_m *gates.*

Proof. We use induction on $|H|$. In the $|H| = 0$ case the AND gates have fan-in 1; thus they can be removed.

We can translate the AND gates to multiplications on the 0-1 variables. Consequently, the input of the MOD_m gate is a polynomial P of the input variables with all of its monomials having at most a single variable outside H. We may suppose that P is multilinear. If $x_i \in H$ for some $1 \leq i \leq n$, we can write this input in the form $P = Qx_i + R$, where the polynomials Q and R do not depend on x_i, and all their monomials contain at most a single variable outside H. We apply Lemma 3.9 to replace our MOD_m gate with the modulo p sum of $2m$ MOD_m gates. The inputs of these MOD_m gates are linear combinations of x_i, Q, and R. To finish the proof, we apply the inductive hypothesis with $H \setminus \{x_i\}$ to replace each of these new MOD_m gates with the modulo p sum of $(2m)^{|H|-1}$ MOD_m gates on the input variables. □

LEMMA 3.12. *Let the prime p and the positive integer m be fixed. Then there exist constants $c > 1$ and $\varepsilon > 0$ such that if a circuit* $((\mathrm{MOD}_{p^k}^A)^\ell, \mathrm{MOD}_m^B, \mathrm{AND})$ *computes* AND_n, *and every* MOD_m *gate is an ε-linear gate, then the size of the circuit is $S > c^n$.*

The proof of this lemma is simpler for the case when p is not dividing m. We need Theorem 3.7 in its full generality for the remaining case.

Proof. Suppose first that p does not divide m.

We apply Lemma 3.11 for the MOD_m gates. The resulting circuit computes a modulo p polynomial of degree less than $p^{k\ell}$ of the at most $S(2m)^{\varepsilon n}$ MOD_m gates (Lemma 3.2). The size is therefore at most $(S(2m)^{\varepsilon n})^{p^{k\ell}}$. But Theorem 2.4 claims an exponential lower bound on this size, thus for a small enough ε, size S must be exponential in n.

In the general case where p may divide m, we write $m = p^a m_0$ where p does not divide m_0. First we decompose each MOD_m^B gate into the sum of $\mathrm{MOD}_m^{\{b_i\}}$ gates for $B = \{b_1, b_2, \ldots, b_t\}$. Then $\mathrm{MOD}_m^{\{b_i\}}$ gates are converted to $\mathrm{MOD}_m^{\{0\}}$ gates, connecting bit 1 with multiple wires to the gate. Next, we exchange $\mathrm{MOD}_m^{\{0\}}$ gates to AND of the $\mathrm{MOD}_{m_0}^{\{0\}}$ and $\mathrm{MOD}_{p^a}^{\{0\}}$ gates. (We used a similar decomposition in the proof of Lemma 3.8.) We have increased the size of the circuit by a factor of at most $2m$ so far. We apply Lemma 3.11 to the MOD_{m_0} gates. This increases the size by a factor of at most $(2m)^{\varepsilon n}$. The resulting circuit has MOD_{m_0} and AND gates at the bottom level and MOD_{p^a}, MOD_p, $\sum_p$, and AND_2 gates everywhere else. As the last two types can be replaced with MOD_p gates, we can apply Lemma 3.2. We get a three level circuit computing AND_n with a $\sum_p$ gate on top and AND_t gates in the middle (with a constant t depending on m, p, k, and ℓ). The bottom gates are MOD_m and AND gates. Notice that the number S_2 of the gates in the middle level is at most S_1^t, where S_1 is the number of gates on the bottom level, and $S_1 \leq (2m)^{\varepsilon n+1}S$.

The fan-in of these bottom AND gates is bounded by $\varepsilon n+1$. We choose $\varepsilon < 1/4t$. Merging the bottom AND gates with the middle AND gates, one gets that AND_n is the modulo p sum of AND functions on at most $n/2$ inputs and at most t MOD_m gates. Applying Theorem 3.7, one gets that $S_2 > c^n$ with some $c > 1$ depending on p, m, k, and ℓ. Thus $S^t > c^n/(2m)^{t(\varepsilon n+1)}$ proving an exponential lower bound on S if $c > (2m)^{\varepsilon t}$. □

Now we turn to prove Theorem 2.6. It is a special case of the following result proving an optimal tradeoff between size and the new measure of the maximal number of related pairs.

THEOREM 3.13. *Let p be a prime and m, k, and ℓ positive integers. Suppose that a* $((\mathrm{MOD}^B_{p^k})^\ell, \mathrm{MOD}^A_m, \mathrm{AND})$ *circuit computes* AND_n. *If each* MOD_m *gate in the circuit relates at most $X \geq n$ pairs of input variables then the size of the circuit is at least $c_0^{n^2/X}$, with a constant $c_0 > 1$ depending on p, m, k, and ℓ.*

Proof. We fix the values c and ε claimed in Lemma 3.12. We take a restriction on the circuit by leaving a variable unrestricted with probability $P = \varepsilon n/(22X)$ independently for each of the variables. We assign 1 to the rest of the variables. Clearly, the restricted circuit computes the AND of the remaining variables.

With probability of at least 1/2, the number of the remaining variables is at least $n_0 = \lfloor Pn/2 \rfloor = \lfloor \varepsilon n^2/(44X) \rfloor$.

Exactly those pairs remained related in a MOD_m gate in the restricted circuit, whose both variables remained unrestricted.

The expected number of pairs related by a single gate in the restricted circuit is at most $XP^2 = \varepsilon n_0/11$. Unfortunately, the deviation can be large, it is easy to construct n gates, relating $n-1$ pairs each, such that *any* restriction to n' variables has a gate relating $n'-1$ pairs. Thus, it is important that, when using Lemma 3.12, we need not bound the number of related pairs, only the size of a set, covering each pair.

Lemma 3.12 easily implies the Theorem if there is a restriction leaving n_0 variables unrestricted, such that every MOD_m gate is ε-linear. In other words, for each G, we need the existence of a set H (depending on G) of size at most εn_0, which contains at least one of every pair related by G in the restricted circuit.

Let us bound the probability that this is not the case for a fixed MOD_m gate G. Take a maximal matching on pairs of unrestricted variables that are related by G. The set H of the endpoints of the matching-edges satisfies that no pair is related outside H, since otherwise, adding that pair to the matching would yield a larger matching. Thus it suffices to bound the probability that $|H| \geq \varepsilon n_0$, i.e., that all the variables involved in some $j = \lceil \varepsilon n_0/2 \rceil$ pairs of G-related variables forming a matching remain unrestricted. We bound this probability by the product of the number of choices for the matching and the probability that the variables remain unrestricted for a fixed matching. For a fixed gate G we get that the probability that at least n_0 variables remain unrestricted but G is not ε-linear after the restriction is at most $\binom{X}{j}P^{2j}$. Hence if $S\binom{X}{j}P^{2j} < 1/2$, where S is the size of our circuit, then Lemma 3.12 proves our theorem. The alternative is $S \geq (2\binom{X}{j}P^{2j})^{-1} \geq \left(\frac{j}{eXP^2}\right)^j$, which proves the same bound. □

Next we show that the logarithmic order of magnitude of the bound in Theorem 3.13 is tight.

THEOREM 3.14. *If m is not a power of the prime p, and $X > 0$ is arbitrary, then the n variable* AND *function is computable by a* $(\sum_p, \mathrm{MOD}_m, \mathrm{AND})$ *circuit of size* $(2m)^{n^2/(2X)}$ *such that the total number of pairs of variables related by any* MOD_m *gate in the circuit is at most X.*

Proof. Compute AND of the variables in two levels with AND gates, first computing the AND of $\lceil n^2/(2X) \rceil$ classes of at most $\lceil 2X/n \rceil$ variables each. Then place a $\mathrm{MOD}_m^{\{1\}}$ gate of fan-in 1 onto the top. Apply Lemma 3.11 to replace the top two levels by the modulo p sum of $(2m)^{n^2/(2X)}$ MOD_m gates. The inputs of these new gates are linear combinations of the outputs of the gates computing AND for a single class.

Note that Lemma 3.11 works only if m is not a multiple of p. Otherwise use that MOD_m gates can simulate MOD_q gates if q divides m. $\square$

We remark that the proofs of Lemma 3.12 and Theorem 3.13 use Theorem 3.7 for the lower bound, so they apply to circuits computing OR or MOD_r with r not dividing mp^s, not just for AND. The upper bound in Theorem 3.14 can also be applied to these functions.

Acknowledgment. We are grateful to Zoltán Király for helpful discussions.

REFERENCES

[1] M. AJTAI, $\sum_1^1$ *formulae on finite structures*, Ann. Pure Appl. Logic, 24 (1983), pp. 1–48.

[2] D. A. M. BARRINGTON, H. STRAUBING, AND D. THÉRIEN, *Non-uniform automata over groups*, Inform. and Comput., 89 (1990), pp. 109–132.

[3] R. BEIGEL AND J. TARUI, *On ACC*, in Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1991, pp. 783–792.

[4] R. B. BOPPANA, AND M. SIPSER, *The complexity of finite functions*, Handbook of Theoretical Computer Science Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier, Amsterdam, MIT, Cambridge, MA, 1990.

[5] H. CAUSSINUS, *A note on a theorem of Barrington, Straubing and Thérien*, Inform. Process. Lett., 58 (1996), pp. 31–33.

[6] M. L. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits and the polynomial time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13–27.

[7] V. GROLMUSZ, *A degree-decreasing lemma for (mod q,mod p) circuits*, in Proceedings ICALP'98, Aalborg, Denmark, Lecture Notes in Comput. Sci. 1443, Springer-Verlag, New York, 1998, pp. 215–222.

[8] J. HÅSTAD, *Almost optimal lower bounds for small depth circuits*, in Proceedings of the 18th Annual ACM Symposium on the Theory of Computing, Berkley, CA, 1986, pp. 6–20.

[9] J. KAHN AND R. MESHULAM, *On mod p transversals*, Combinatorica, 10 (1991), pp. 17–22.

[10] M. KRAUSE AND P. PUDLÁK, *On the computational power of depth 2 circuits with threshold and modulo gates*, in Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, Montreal, Canada, 1994, pp. 48–57.

[11] M. KRAUSE AND S. WAACK, *Variation ranks of communication matrices and lower bounds for depth-two circuits having nearly symmetric gates with unbounded fan-in*, Math. Systems Theory, 28 (1995), pp. 553–564.

[12] A. RAZBOROV, *Lower bounds for the monotone complexity of some Boolean functions*, Sov. Math. Dokl., 31 (1985), pp. 354–357.

[13] A. RAZBOROV, *Lower bounds on the size of bounded depth networks over a complete basis with logical addition*, Mat. Zametki, 41 (1987), pp. 598–607 (in Russian).

[14] R. SMOLENSKY, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proceedings of the 19th Annual ACM Symposium on the Theory of Computing, New

York, NY, 1987, pp. 77–82.

[15] P. Yan and I. Parberry, *Exponential size lower bounds for some depth three circuits*, Inform. and Comput., 112 (1994), pp. 117–130.

[16] A. C. Yao, *Separating the polynomial-time hierarchy by oracles*, in Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1985, pp. 1–10.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1223–1246

SHORTEST PATH QUERIES AMONG WEIGHTED OBSTACLES IN THE RECTILINEAR PLANE[*]

DANNY Z. CHEN[†], KEVIN S. KLENK[†], AND HUNG-YI T. TU[‡]

Abstract. We study the problems of processing single-source and two-point shortest path queries among weighted polygonal obstacles in the rectilinear plane. For the single-source case, we construct a data structure in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space, where n is the number of obstacle vertices; this data structure enables us to report the length of a shortest path between the source and any query point in $O(\log n)$ time, and an actual shortest path in $O(\log n + k)$ time, where k is the number of edges on the output path. For the two-point case, we construct a data structure in $O(n^2 \log^2 n)$ time and space; this data structure enables us to report the length of a shortest path between two arbitrary query points in $O(\log^2 n)$ time, and an actual shortest path in $O(\log^2 n + k)$ time. Our work improves and generalizes the previously best-known results on computing rectilinear shortest paths among weighted polygonal obstacles. We also apply our techniques to processing two-point L_1 shortest obstacle-avoiding path queries among arbitrary (i.e., not necessarily rectilinear) polygonal obstacles in the plane. No algorithm for processing two-point shortest path queries among weighted obstacles was previously known.

Key words. computational geometry, path planning, data structures, analysis of algorithms

AMS subject classifications. 68P05, 68Q25

PII. S0097539796307194

1. Introduction.

The problems of computing shortest paths among geometric obstacles are among the most fundamental topics in computational geometry. Given a *rectilinear* plane with disjoint weighted polygonal obstacles of n vertices in total, we study the problems of answering shortest path queries in this setting. A plane is rectilinear if all geometric objects (e.g., lines, obstacles, paths) in it are *rectilinear*, i.e., each line or edge of such an object is parallel to the x- or y-axis, and distances are measured based on the L_1 metric. No two obstacles intersect each other in their interior points. Every obstacle is *weighted*, i.e., it is associated with a nonnegative *weight* factor, so that a path in the plane, if it intersects the interior of an obstacle, is charged extra cost based on the weight of that obstacle in addition to the cost of the L_1 length of the path. A *shortest path* connecting two points in the plane is a path with the minimum total cost. Note that shortest paths of this kind are in fact a generalization of the shortest paths (that completely avoid the interior of obstacles) studied in classical geometric path planning problems; our shortest paths become obstacle-avoiding if the weight of each obstacle is $+\infty$. A shortest path query specifies two points, s and t, in the plane and requests a rectilinear shortest path (or its weighted total cost) connecting s and t. If the point s is always fixed and t is arbitrary, then the query is called a *single-source* query and the fixed point s is

[*]Received by the editors July 22, 1996; accepted for publication (in revised form) November 23, 1998; published electronically February 10, 2000. A preliminary version of this work appeared in Proceedings of the Eleventh Annual ACM Symposium on Computational Geometry, Vancouver, Canada, 1995, pp. 370–379.

http://www.siam.org/journals/sicomp/29-4/30719.html.

[†]Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 (dchen@cse.nd.edu, kklenk@cse.nd.edu). The research of the first author was supported in part by National Science Foundation grant CCR-9623585. The research of the second author was supported in part by the William D. Mensch, Jr., Summer Fellowship.

[‡]Department of Computer Science and Information Management, Providence University, Shalu, Taichung Hsien, 43309, Taiwan, ROC (hytu@simon.pu.edu.tw).

called the *source point*. If both s and t are arbitrary points, then the query is called a *two-point* query. Our objective is to create data structures which enable shortest path queries to be answered efficiently.

Unlike shortest path problems in graph theory, in which both the single-source and two-point (or called all-pairs) versions are well studied, geometric shortest path query problems, especially the two-point query versions, are not yet well understood. Geometric shortest path queries can specify arbitrary points in the plane which consists of uncountable infinitely many points. Furthermore, it is still difficult in many situations to obtain good solutions to the two-point query versions even when efficient algorithms for the corresponding single-source query versions are known. The two-point Euclidean shortest path query problem is such an example, for which only efficient approximation results [6, 2] are known so far.

A number of algorithms have been discovered for computing rectilinear shortest obstacle-avoiding paths [3, 4, 8, 9, 10, 11, 13, 21, 22, 24, 25, 26, 31, 32, 33] and shortest paths among weighted obstacles [15, 24, 28, 32]. When the plane has only a single obstacle (i.e., a simple polygon), shortest path query problems are much easier and in fact are quite well studied (e.g., see [7, 16, 17, 18, 19, 23]). However, shortest path query problems become substantially harder in environments with multiple obstacles. There were only a few algorithms [10, 11, 20, 23, 25, 26, 27, 29] for *single-source obstacle-avoiding* path queries in multiple obstacle environments. Results on *two-point* queries in such environments were even rarer; the only such previous algorithms we know are for the special cases of *rectilinear* shortest obstacle-avoiding paths among *rectangular* obstacles [3, 4, 13] and among *rectilinear polygonal* obstacles [21]. For rectilinear shortest obstacle-avoiding path queries among rectilinear polygonal obstacles, Iwai, Suzuki, and Nishizeki [21] built a data structure in $O(n^2 \log^3 n)$ time and $O(n^2 \log^2 n)$ space, which supports each two-point length query in $O(\log^2 n)$ time. For shortest paths among weighted obstacles, very few algorithms were previously known [15, 24, 28, 32], and none of these results can handle arbitrary shortest path queries. In particular, Mitchell and Papadimitriou [28] solved the problem of determining shortest paths through a weighted planar polygonal subdivision; their algorithm takes a time of $O(n^8)$ times another factor based on the precision of the problem instance. Lee, Yang, and Chen [24] presented two algorithms for computing a rectilinear shortest path between two points among weighted rectilinear polygonal obstacles in the plane; their first algorithm takes $O(n \log^2 n)$ time and $O(n \log n)$ space and the second takes $O(n \log^{3/2} n)$ time and space.

In this paper, we present new techniques for processing single-source and two-point rectilinear shortest path queries among disjoint weighted rectilinear polygonal obstacles. Using these techniques, we are also able to obtain efficient algorithms for answering other path queries, including two-point L_1 shortest obstacle-avoiding path queries among arbitrary (not necessarily rectilinear) polygonal obstacles. Our techniques are based on numerous new geometric observations and the generalized visibility graph of Lee, Yang, and Chen [24]. We introduce the idea of "gateway points" which are used to control effectively the connection between any query points and the generalized visibility graph. We also present methods for computing single-source shortest path trees in the generalized visibility graph that are more efficient in both the time and space bounds than directly applying the best-known graphical shortest path algorithm by Fredman and Tarjan [14]. Our techniques and ideas could be useful in solving other shortest path query problems. Our main results are as follows:

- For single-source queries among weighted rectilinear obstacles in the rectilinear plane, we construct a data structure in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space. This data structure supports each path length query in $O(\log n)$ time. An actual shortest path can be reported in $O(\log n + k)$ time, where k is the number of edges on the output path. An immediate consequence of this result is an improvement over the previously best-known algorithms in [24] for computing a rectilinear shortest path between two points among weighted rectilinear obstacles by a factor of $O(\log^{1/2} n)$ in either the time or space bound.

• For two-point queries among weighted rectilinear obstacles in the rectilinear plane, we construct a data structure in $O(n^2 \log^2 n)$ time and space. This data structure supports each path length query in $O(\log^2 n)$ time. An actual shortest path can be obtained in $O(\log^2 n + k)$ time.

• For two-point L_1 shortest *obstacle-avoiding* path queries among arbitrary (not necessarily rectilinear) polygonal obstacles in the plane, we build a data structure in $O(n^2 \log^2 n)$ time and $O(n^2 \log n)$ space. This data structure supports each path length query in $O(\log^2 n)$ time. An actual shortest path can be obtained in $O(\log^2 n + k)$ time.

Note that in comparison with the two-point result in [21], our algorithms work for more general geometric settings (*weighted* rectilinear obstacles or *arbitrary* polygonal obstacles) yet take less time (a factor of $\log n$) in constructing the data structures.

Unless otherwise specified, all geometric objects (e.g., lines, paths, polygons) are implicitly assumed to be rectilinear; i.e., each of their segments is parallel to one of the coordinate axes.

The rest of the paper is organized as follows. Section 2 reviews briefly the construction of the generalized visibility graph that is a basis for our data structures. Section 3 details our insights and procedures for efficiently processing path queries among weighted rectilinear obstacles. Section 4 presents our algorithms for creating the data structures for both the single-source and two-point queries among weighted rectilinear obstacles. Section 5 extends our work to two-point L_1 shortest obstacle-avoiding path queries among arbitrary polygonal obstacles.

2. Preliminaries.

Let the x- (resp., y-) coordinate of a point s on the plane be denoted by s_x (resp., s_y). For points s and t, the L_1 distance between s and t is $\text{dist}(s,t) = |s_x - t_x| + |s_y - t_y|$. A path P_{st} connecting s and t is a sequence of line segments $\overline{p_0p_1}$, $\overline{p_1p_2}$, …, $\overline{p_{m-1}p_m}$, with $p_0 = s$ and $p_m = t$. Let $d(P_{st})$ denote the (unweighted) length of P_{st}, i.e., the sum of the L_1 lengths of the segments on P_{st}. If P_{st} intersects the interior of some weighted obstacles, the *weighted length* of P_{st} is defined as follows: Partition P_{st} into subpaths $A_1, B_1, A_2, B_2, \ldots, A_k, B_k$, where A_i is a subpath without intersecting the interior of any obstacle, B_i is a subpath completely within the interior of some obstacle R_i, and subpaths A_1 and B_k may be of zero length; the weighted length of P_{st}, denoted by $d_w(P_{st})$, is $d_w(P_{st}) = \sum_{i=1}^{k}(d(A_i) + d(B_i)) + \sum_{i=1}^{k}(d(B_i) \times W(R_i))$, where $W(R_i)$ is the weight factor of R_i.

An obstacle R_i (a polygon, possibly with holes) is specified by a sequence of edges on each of its outer and inner boundaries. For each vertex u of R_i such that the interior angle of R_i at u is $3\pi/2$, we define the *internal projection points* of u as follows: Let $\overline{uv}$ be the horizontal (resp., vertical) edge of R_i containing u; shoot a horizontal (resp., vertical) ray from u along the direction opposite to that of v until the ray hits the first boundary point $p_h(u)$ (resp., $p_v(u)$) of R_i. We call $p_h(u)$ (resp., $p_v(u)$) the *horizontal* (resp., *vertical*) *internal projection point* of u. Internal projection points are useful for controlling shortest paths that penetrate weighted obstacles.

The notion of visibility is generalized to the rectilinear plane with weighted obstacles by Lee, Yang, and Chen [24]. Two points s and t are *visible* to each other if $\overline{st}$ is horizontal or vertical and $\overline{st}$ either does not intersect the interior of any obstacle or is completely contained in a single obstacle. A point p is *visible* from a horizontal or vertical line L if and only if p is visible from a finite point p' on L. A key component of our shortest path data structures is the generalized visibility graph $G = (V, E)$ defined in [24] which captures the necessary information about shortest paths among the n obstacle vertices. We denote the cost of each edge $e \in E$ by $w(e)$.

The vertex set V of the visibility graph G can be partitioned into three subsets: (i) V_O of obstacle vertices, (ii) V_I of internal projection points of the vertices in V_O, and (iii) V_S of *Steiner points*. We need to sketch the recursive procedure in [24] for generating Steiner points for graph G: (1) Draw a vertical (resp., horizontal) line L, which we call a *cut-line*, at the median of the x- (resp., y-) coordinates of all the vertices in $V_O \cup V_I$; (2) project the vertices in $V_O \cup V_I$ that are visible from the cut-line L onto L (the projection points of $V_O \cup V_I$ on L are the Steiner points of V_S on L); (3) use L to partition $V_O \cup V_I$ into two subsets S_1 and S_2, one on each side of L; (4) perform the procedure recursively on the subsets S_1 and S_2, resp., until the size of each such subset becomes 1. Because this procedure has $O(\log n)$ recursion levels, it clearly generates $O(n \log n)$ Steiner points in V_S. Since $|V_O \cup V_I| = O(n)$, there are totally $O(n \log n)$ vertices in $V = V_O \cup V_I \cup V_S$. We associate with each cut-line L a *level number* $LN(L)$, which is the number of the recursion level at which L is used in the above procedure (with the root level being level 1).

The edge set E of G consists of two subsets: (1) E_V of line segments between every Steiner point in V_S and its corresponding vertex in $V_O \cup V_I$, and (2) E_L of line segments connecting consecutive Steiner points on every cut-line. The weighted lengths of the edges in E_V can be easily determined and stored during the generation of the Steiner points. The weighted lengths of the edges in E_L are obtained by using Widmayer's algorithm [24, 30] in $O(n \log n)$ time.

It was shown in [24] that graph G so created captures the necessary information about shortest paths among points in the plane that are vertices of V. The first algorithm in [24] runs Fredman and Tarjan's shortest path algorithm [14] on G to find a shortest path between two points s and t (with both s and t being included in V), in $O(n \log^2 n)$ time and $O(n \log n)$ space.

The second algorithm in [24] improved the time bound of the first algorithm, based on Clarkson, Kapoor, and Vaidya's idea [8] of reducing the number of vertices (precisely, the Steiner points) in graph G by increasing the number of edges. Let the set of additional edges to G be E_S, and denote the graph so resulted by $G' = (V', E')$. G' was obtained from G in [24] as follows: (1) For every vertical (resp., horizontal) cut-line L, partition the plane into horizontal (resp., vertical) strips such that each strip contains $O(\log^{1/2} n)$ Steiner points of $L \cap V_S$; (2) within each strip, for every pair of vertices u and v in $V_O \cup V_I$ such that u and v are both visible from L and are on the opposite sides of L, add an edge (u, v) to E_S, and let the cost of (u, v) be the sum of the costs of the edges $(u, s(u))$, $(s(u), s(v))$, and $(s(v), v)$ in G, where $s(u)$ and $s(v)$ are Steiner points on L corresponding to u and v, resp.; (3) in each strip, retain the Steiner points that have the highest or lowest y- (resp., x-) coordinate among the Steiner points in $L \cap V_S$, and remove from V_S the rest of the Steiner points on L in that strip together with edges adjacent to any of the removed Steiner points. (The retained Steiner points play the role of maintaining communication between consecutive strips.)

Graph $G' = (V', E')$ created by this procedure has $|V'| = O(n \log^{1/2} n)$ and $|E'|$

$= O(n \log^{3/2} n)$. The second algorithm in [24] then runs the shortest path algorithm [14] on G', in $O(n \log^{3/2} n)$ time and space. Thus, there is a time-space trade-off between the first and second algorithm in [24].

Our algorithms for shortest path queries make use of both graphs G and G'. In particular, G is a key component of our shortest path data structures, and G' is used in *constructing* our shortest path data structures. We will also show (in section 4) that by using an *implicit* representation scheme for G', the time-space trade-off incurred by the algorithms in [24] can be eliminated, and hence the best time and space bounds of both these algorithms can be achieved simultaneously.

3. Shortest path queries among weighted rectilinear obstacles. We discuss in this section our algorithms for answering single-source and two-point shortest path queries among weighted rectilinear obstacles, while deferring the complete description of the desired shortest path data structures and their construction until the next section. This allows us to show an array of useful geometric structures of these shortest path query problems before giving the details of various components of our algorithmic solutions. Exploiting these new geometric observations is one of our main contributions in this paper.

3.1. Useful observations. We start with some notation and useful insights. For a query point z, we project z vertically (resp., horizontally) onto the first obstacle edge (if it exists) that is above (resp., below, to the left of, to the right of) z. Let $Q(z)$ be the set of the at most four projection points of z on obstacle boundaries.

A path P is *monotone* with respect to the x- (resp., y-) axis if and only if no vertical (resp., horizontal) line crosses P more than once. A path is called a *staircase* if it is monotone with respect to both the x-axis and the y-axis. Clearly, a staircase from a point p to a point q in an obstacle-free region is a shortest path between p and q since its length equals $\text{dist}(p, q)$.

The following lemmas specify various useful structures of shortest paths between two arbitrary points q and r.

LEMMA 3.1. *Suppose there is a shortest path P_{qr} connecting points q and r whose interior does not intersect any obstacle boundary. Then* (1) *P_{qr} is a staircase, and* (2) *there is a shortest path P'_{qr} between q and r such that P'_{qr} either contains an obstacle vertex or P'_{qr} consists of at most two edges.*

Proof. We prove (1) by contradiction. Assume the interior of a shortest path P_{qr} has no intersection with any obstacle boundary yet P_{qr} is not a staircase. Then P_{qr} must have three consecutive edges that form two right turns or two left turns. But, by moving the middle one of these three edges, one can obtain a shorter path between q and r than P_{qr}, a contradiction.

For proving (2), assume without loss of generality (WLOG) the staircase P_{qr} consists of at least three edges. Consider its first two edges $\overline{qq_1}$ and $\overline{q_1q_2}$ (see Figure 3.1). Since the interior of $\overline{qq_1}$ does not intersect any obstacle boundary, we can move $\overline{qq_1}$, until either the interior of the segment $\overline{q'q'_1}$ encounters some obstacle boundary or $q'_1 = q_2$. The path P'_{qr} obtained by replacing $\overline{qq_1}$ and $\overline{q_1q_2}$ of P_{qr} with $\overline{qq'}$, $\overline{q'q'_1}$, and $\overline{q'_1q_2}$ has the same length as P_{qr}. If the interior of $\overline{q'q'_1}$ encounters some obstacle boundary, then $\overline{q'q'_1}$ must contain a vertex of an obstacle edge e (because e does not intersect $\overline{q_1q_2}$), and hence the lemma is proved. If $q'_1 = q_2$, then the resulting shortest path P'_{qr} has one less edge than P_{qr}. By repeatedly applying this reduction to P'_{qr}, we obtain a shortest path between q and r that either passes through an obstacle vertex or consists of at most two edges. ∎

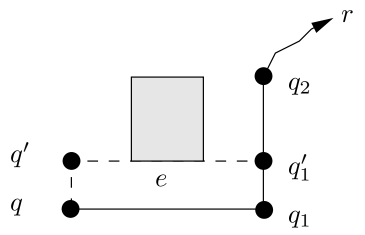


FIG. 3.1. *Moving the segment $\overline{qq_1}$ vertically up.*

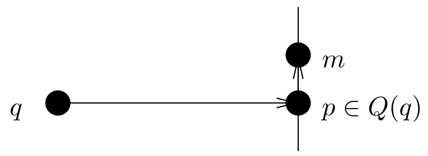


FIG. 3.2. *The two line segments of P_{qm} meeting at a point $p \in Q(q)$.*

LEMMA 3.2. *For two points q and r with q not on any obstacle boundary, if the interior of a shortest q-to-r path P_{qr} intersects the boundary of an obstacle, then there is a shortest q-to-r path that either goes through a point $p \in Q(q)$ (via the segment $\overline{qp}$) or goes through an obstacle vertex.*

Proof. Let m be the obstacle boundary point on the interior of P_{qr} such that the interior of the subpath P_{qm} of P_{qr} does not intersect any obstacle boundary point. By Lemma 3.1, P_{qm} is a staircase that can be made either to pass through an obstacle vertex or to consist of at most two line segments. WLOG, we assume that P_{qm} consists of at most two edges.

Let m be on an obstacle edge $e(m)$. Then either $e(m)$ contains a point $p \in Q(q)$, or $e(m)$ contains no point of $Q(q)$. It is not hard to see the following. If $e(m)$ contains a point $p \in Q(q)$, then the two edges of P_{qm} can be made to join together at $p \in Q(q)$ (e.g., Figure 3.2). If $e(m)$ contains no point of $Q(q)$, then a vertex of $e(m)$ must belong to P_{qm} (e.g., see Figure 3.3). ☐

LEMMA 3.3. *Suppose a point q is on an obstacle boundary. Then, for any point r, there is a shortest q-to-r path P_{qr} that either consists of at most two line segments or goes through an obstacle vertex.*

Proof. The proof is similar to that of Lemma 3.1. ☐

The significance of Lemmas 3.1, 3.2, and 3.3 to processing shortest path queries is as follows. Given two query points s and t, there are only three possibilities: (1) a shortest path P_{st} goes through an obstacle vertex and hence through a vertex of the generalized visibility graph G (by possibly going through a point in $Q(s) \cup Q(t)$); (2) P_{st} goes through a point in $Q(s) \cup Q(t)$ but not through any vertex of G; or (3) P_{st} consists of at most two line segments. In case 2, the path P_{qr}, for some $q \in Q(s)$ and $r \in Q(t)$, is one with at most two line segments.

Note that in the single-source case, a shortest path from any query point in the plane to the source point always goes through some vertices of graph G (e.g., the source vertex), and thus G can be used to effectively control single-source shortest paths for all query points. However, the two-point case may be different, because shortest paths between certain query points in the plane do not pass through any vertex of G. Therefore, our two-point query algorithms must account for both possibilities: (i) a sought path goes through a vertex of G, and (ii) no desired path goes through any

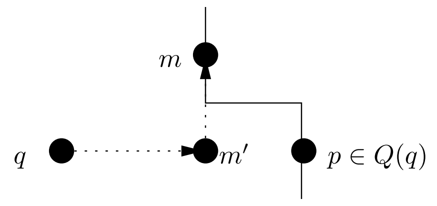


FIG. 3.3. *The segment $\overline{mm'} \subseteq e(m)$ of P_{qm} must contain an obstacle vertex.*

vertex of G. In the next subsection, we discuss our query algorithms for the possibility of a desired shortest path going through a vertex of G (i.e., case 1). The possibility of no desired shortest path going through any vertex of G is handled in section 3.5.2 (i.e., cases 2 and 3).

3.2. Main ideas and difficulties. In this subsection, we focus on two-point queries for which a desired shortest path goes through some vertices of G. Let s and t be two query points. Based on Lemmas 3.2 and 3.3, we need to compute the length of a shortest p-to-q path through a vertex of G for every $p \in Q(s) \cup \{s\}$ and $q \in Q(t) \cup \{t\}$.

Our main idea for computing a shortest p-to-q path that intersects some vertices of G is to identify a subset $V_{\mathrm{g}}(z)$ of vertices in G for each $z \in \{p, q\}$, such that a shortest p-to-q path must go through at least one pair v_q, v_p of vertices, with $v_q \in V_{\mathrm{g}}(q)$ and $v_p \in V_{\mathrm{g}}(p)$. We call the vertices of G in $V_{\mathrm{g}}(q)$ the *gateways* of q. If we know $V_{\mathrm{g}}(q)$ and $V_{\mathrm{g}}(p)$ and the weighted path between q (resp., p) and each vertex in $V_{\mathrm{g}}(q)$ (resp., $V_{\mathrm{g}}(p)$), then we can easily find a shortest p-to-q path by utilizing a data structure that maintains shortest path information for every pair of vertices in G. In this section, we simply assume that our two-point data structure can report the length of a shortest path between any two vertices of G in $O(1)$ time and a corresponding actual shortest path in an additional $O(k)$ time, where k is the number of edges of the output path.

For this idea to work, we must overcome two difficulties: (1) identifying $V_{\mathrm{g}}(q)$, and (2) determining the lengths of the weighted paths between q and the vertices in $V_{\mathrm{g}}(q)$. Note that it is possible for a path between q and a vertex in $V_{\mathrm{g}}(q)$ to penetrate a number of obstacles. For the efficiency of our query algorithms, it is certainly desirable that $|V_{\mathrm{g}}(q)|$ be small. Indeed, we will show that $|V_{\mathrm{g}}(q)| = O(\log n)$ for any point q and $V_{\mathrm{g}}(q)$ can be computed in $O(\log n)$ time. Furthermore, we show that the weighted paths between q and the vertices in $V_{\mathrm{g}}(q)$ and their lengths can also be computed in $O(\log n)$ time.

Our idea for computing the set $V_{\mathrm{g}}(q)$ of gateways for a point q is that of "inserting" q into graph G. We treat q as if it were one of the obstacle vertices and compute the edges in G adjacent to q that would have resulted if the procedure in [24] for constructing G were applied to q. Note that such an "insertion" of a point into G does not actually change the shortest path information contained in G (i.e., q is not a new obstacle); furthermore, the resulting graph (after the "insertions") contains shortest path information among the inserted points and the original vertices of G, if a shortest path between two inserted points does intersect G. Also, note that our "insertion" process does not actually modify graph G because the points are never truly inserted into G.

3.3. Characterization of gateways. To "insert" a point q into graph G, we project q onto the relevant cut-lines based on the graph construction procedure [24]. A fixed set of $O(n)$ cut-lines is used in constructing G, subdividing the plane recursively in [24]. Each cut-line can be associated with a particular recursion level. In consequence, all points in each region of the resulted planar subdivision are projected onto the same subset of cut-lines. If, according to the graph construction procedure [24], q would have been projected onto a cut-line L, then we say L is a *projection cut-line* of q. Note that if a cut-line L is a projection cut-line of q, then q is visible from L. But, not every cut-line visible from q is a projection cut-line of q. It is sufficient for us to discuss geometric observations with vertical cut-lines only (those with horizontal cut-lines are symmetric). In the rest of this section, we assume all the cut-lines are vertical unless otherwise specified.

LEMMA 3.4. *Suppose that the projection cut-lines of a point q are sorted in increasing order by their x-coordinates. Then the projection cut-lines of q that are to the right (resp., left) of q are simultaneously in the decreasing (resp., increasing) order of their level numbers.*

Proof. We prove the lemma only for the projection cut-lines of q that are to the right of q. Let L' and L'' be two projection cut-lines to the right of q with L' to the left of L''. Then the level number of L' must be bigger than that of L'' (otherwise, L' would have prevented the projection of q from reaching L'', a contradiction). Hence, the lemma follows. □

The gateway set $V_{\mathrm{g}}(q)$ of a point q is determined as follows: For each projection cut-line L of q, if z is the vertex of G on L that is immediately above (resp., below) $p_L(q)$, then $z \in V_{\mathrm{g}}(q)$, where $p_L(q)$ is the projection point of q on L. For $z \in V_{\mathrm{g}}(q)$, we denote the projection cut-line of q that contains z by L_z. Observe that if q were "inserted" into G, then the only edges adjacent to q in the graph would be those connecting q with its projection points. Because each projection point $p_L(q)$ of q is adjacent to at most two neighboring vertices of G on its cut-line L, $V_{\mathrm{g}}(q)$ controls every path in G from q to any other vertex of G. Since at each recursion level of the graph construction algorithm, q can be projected onto at most one cut-line, there are $O(\log n)$ projection cut-lines for q. Each such projection cut-line has at most two neighboring vertices of G for $p_L(q)$. Therefore, we have the next two lemmas.

LEMMA 3.5. *For each point q in the plane, $|V_{\mathrm{g}}(q)| = O(\log n)$.*

LEMMA 3.6. *For each point q and each vertex v of G, there is a shortest q-to-v path in the plane that goes through a vertex of $V_{\mathrm{g}}(q)$.*

For every gateway $z \in V_{\mathrm{g}}(q)$, we define an "edge" (q, z) in the graph as follows: Let z be on a projection cut-line L of q; then the edge (q, z) consists of the two segments $\overline{qp(q)}$ and $\overline{p(q)z}$.

By the definition of gateways, $V_{\mathrm{g}}(q)$ can be computed efficiently (e.g., in $O(\log^2 n)$ time by doing a binary search on the vertices of G along each projection cut-line of q, or in $O(\log n)$ time by using a fractional cascading data structure [5]). However, the real difficulty is that each edge connecting q with a vertex in $V_{\mathrm{g}}(q)$ can penetrate many obstacles, and straightforward methods fail to compute the weighted lengths of such edges efficiently. By exploiting a number of geometric observations, we show below how to compute $V_{\mathrm{g}}(q)$ and the weights of all these edges in $O(\log n)$ time.

A *gateway region* $\mathcal{R}(q)$ (Figure 3.4) of a point q is the area enclosed by an interconnection of the gateways in $V_{\mathrm{g}}(q)$. We show only how the set ${V_{\mathrm{g}}}^1(q)$ of gateways in the first quadrant of q are connected: (1) Let a vertical "pseudo" cut-line pass through q. (2) For each vertex v of ${V_{\mathrm{g}}}^1(q)$, project v horizontally to the projection cut-line L of q (on point $p_L(v)$) that is immediately to the left of v (it can be shown that such a projection is always possible among weighted obstacles). Note that L can be the

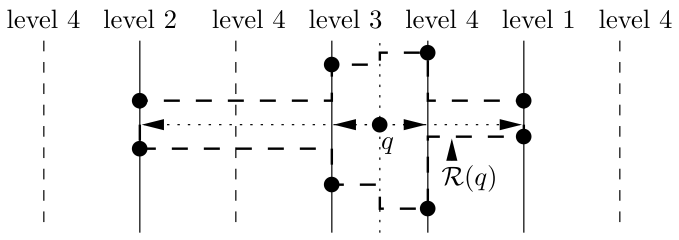


FIG. 3.4. *The gateway region $\mathcal{R}(q)$ for a point q.*

"pseudo" cut-line containing q. For v, let the vertex of $V_{\mathrm{g}}{}^1(q)$ on L be u (if u exists). (3) Connect u and v by line segments $\overline{up_L(v)}$ and $\overline{p_L(v)v}$. (4) If v is the rightmost gateway of $V_{\mathrm{g}}{}^1(q)$, then connect v with the point (v_x, q_y) by a segment. (5) If L is the "pseudo" cut-line containing q (and hence u does not exist on L), then connect v and $p_L(v)$ by a segment. The area in the first quadrant of q enclosed together by the polygonal chain so defined, the x-axis, and y-axis (with q as the origin) is part of $\mathcal{R}(q)$. $\mathcal{R}(q)$ is the union of the areas so defined in all four quadrants of q.

A planar region $\mathcal{R}$ is *rectilinearly convex* if $\mathcal{R}$ is connected and for any horizontal or vertical line L, $L \cap \mathcal{R}$ is also connected. The next two lemmas characterize some crucial structures of $\mathcal{R}(q)$.

LEMMA 3.7. *For any point q, the gateway region $\mathcal{R}(q)$ is rectilinearly convex; furthermore, for any point z in $\mathcal{R}(q)$, the points (z_x, q_y) and (q_x, z_y) are both in $\mathcal{R}(q)$.*

Proof. We know by Lemma 3.4 that the projection cut-lines of q are sorted by their level-numbers. It is sufficient to prove that in each quadrant of q, the gateways in sorted order form a sequence of "restricted" maximal elements. Let S be a point set in the first quadrant of q. We say a point $a \in S$ is a *restricted maximal element* of S if there is no other point $b \in S$ such that $a_x < b_x$ and $a_y < b_y$. The restricted maximal elements in other quadrants of q are defined in a symmetric fashion. WLOG, we prove only the case for the first quadrant of q.

This proof is by contradiction. Suppose the first quadrant of q contains two gateways a, $b \in V_{\mathrm{g}}(q)$ such that $a_x < b_x$ and $a_y < b_y$. Let a and b be on the projection cut-lines L_a and L_b of q, resp. Since $a_x < b_x$, by Lemma 3.4, L_a is at a higher recursion level than L_b (i.e., $LN(L_a) > LN(L_b)$). Let a be a projection point of vertex $v_a \in V_O \cup V_I$. But $a_y < b_y$ implies that v_a is not able to project on L_b. Furthermore, v_a must be to the left of L_b (otherwise, v_a would have been projected on L_b because L_a is to the left of L_b). Hence, the projection of v_a on L_b is blocked. There are two possible cases on blocking the projection of v_a from L_b: (i) blocking by a cut-line L^* such that $LN(L^*) \leq LN(L_b)$, or (ii) blocking by a vertical obstacle edge e that is to the left of L_b.

In (i), cut-line L^* would have also blocked q from being projected on L_b because of the level numbers of L^* and L_b, contradicting the assumption that $b \in V_{\mathrm{g}}(q)$. In (ii), let v be the lower vertex of the vertical edge e. Note that e cannot cross the horizontal line H_q that contains q. This implies that $v_y < b_y$ but v is also not able to project on L_b. Repeatedly applying the same argument on v, we must eventually have some vertical obstacle edge that either crosses the horizontal line H_q or is to the right of L_b since there are only finitely many obstacle edges. But that would contradict with either $b \in V_{\mathrm{g}}(q)$ or that such a vertical edge must be to the left of L_b. □

LEMMA 3.8. *The gateway region $\mathcal{R}(q)$ contains no vertices of $V_O \cup V_I$ in its*

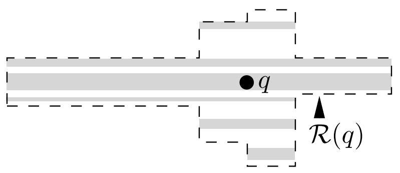


FIG. 3.5. *Horizontal obstacle strips inside the gateway region* $\mathcal{R}(q)$.

interior.

Proof. This can be proved by a contradiction argument similar to the proof of Lemma 3.7. ∎

To further discuss the properties of gateways of a point q, we distinguish three cases: (a) q is an obstacle vertex, (b) q is not an obstacle vertex but is on a vertical obstacle edge, and (c) q is not on any vertical obstacle edge. Case (a) can be readily handled. Hence we only focus on (b) and (c) in the following two subsections. We first discuss (c) and then show how to reduce (b) to (c).

3.3.1. Point q not on any vertical obstacle edge. Throughout this subsection, we assume that point q is not on any vertical obstacle edge.

The lemma below follows from Lemma 3.8 and it further characterizes $\mathcal{R}(q)$.

LEMMA 3.9. *There are three possible relations of obstacles to the gateway region* $\mathcal{R}(q)$*:* (1) *the interior of* $\mathcal{R}(q)$ *does not intersect any obstacle;* (2) $\mathcal{R}(q)$ *is completely contained within an obstacle; or* (3) $\mathcal{R}(q)$ *has horizontal obstacle strips running through it (e.g., see Figure* 3.5*).*

Proof. The interior of $\mathcal{R}(q)$ either intersects some obstacle edges or it does not. If the interior of $\mathcal{R}(q)$ does not intersect any obstacle edge, then clearly either case 1 or 2 holds. If the interior of $\mathcal{R}(q)$ does intersect an obstacle edge e, then by Lemma 3.8, no vertex of e can be in the interior of $\mathcal{R}(q)$. By the convexity of $\mathcal{R}(q)$ (Lemma 3.7), $e \cap \mathcal{R}(q)$ is a single line segment s_e. Segment s_e cannot be vertical because if it were, then it would have intersected the horizontal line containing q at a point inside $\mathcal{R}(q)$ and thus blocked the projection of q beyond s_e, contradicting that s_e is in the interior of $\mathcal{R}(q)$. Hence, s_e must be on the boundary of a horizontal obstacle strip across $\mathcal{R}(q)$. ∎

The following lemmas provide the basis for an efficient computation of $V_{\mathrm{g}}(q)$ and their weighted edges to q.

LEMMA 3.10. *For two gateways* $a, b \in V_{\mathrm{g}}(q)$ *that are both above the horizontal line* H_q *containing* q*, let* $LN(L_a) > LN(L_b)$*, with* L_a *not being the leftmost or rightmost projection cut-line of* q*. Then the set of horizontal obstacle strips penetrated by segment* $\overline{bb'}$ *is a subset of the horizontal obstacle strips penetrated by segment* $\overline{aa'}$*, where* $a' = (a_x, q_y)$ *and* $b' = (b_x, q_y)$ *(see Figure* 3.6*).*

Proof. WLOG, assume both a and b are above line H_q. If a and b are both to the left (resp., right) of q, then the lemma follows immediately from Lemmas 3.4, 3.7, and 3.9. If a and b are on the opposite sides of q, then based on Lemma 3.9, the lemma will follow if we can show $a_y \geq b_y$. We prove $a_y \geq b_y$ by contradiction. Assume $a_y < b_y$. WLOG, let a be to the right of q and b to the left of q (Figure 3.6), and let a be a projection point of a vertex $v_a \in V_O \cup V_I$.

First, we show that v_a is to the right of L_a. By Lemmas 3.8 and 3.9, v_a cannot be between L_a and L_b and cannot be on L_a. Also, v_a cannot be to the left of L_b (otherwise, a would have not been a projection point of v_a on L_a because $LN(L_b)$

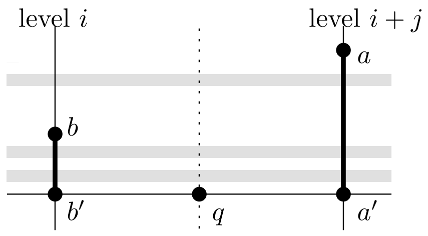


FIG. 3.6. *Illustration of Lemma* 3.10.

$< LN(L_a)$). Next, we show that a cannot be on a vertical obstacle edge. If a were on a vertical obstacle edge e, then, since L_a (which contains e) is neither the leftmost nor rightmost projection cut-line of q, e cannot cross line H_q; hence the lower vertex of e would be in the interior of $\mathcal{R}(q)$, a contradiction. Finally, we show that v_a cannot be blocked from being projected on L_b. By Lemma 3.4, no projection cut-line of q between L_a and L_b has a lower level number than L_b; hence no such projection cut-line can block the projection of v_a on L_b. Since a is not on a vertical obstacle edge, the projection of v_a can reach the left of L_a. The projection of v_a also cannot be blocked by obstacles between L_a and L_b by Lemma 3.9. Therefore, if $a_y < b_y$, then v_a would have been projected onto L_b, contradicting that b is a gateway of q on L_b. ∎

LEMMA 3.11. *For two gateways a, $b \in V_g(q)$ that are both above the horizontal line H_q containing q, let $LN(L_a) > LN(L_b)$. Then $a_y < b_y$ implies that L_a is the leftmost or rightmost projection cut-line of q and that a is on a vertical obstacle edge e such that e crosses H_q.*

Proof. WLOG, assume a and b are both above H_q and $a_y < b_y$. By Lemmas 3.4 and 3.7, a and b cannot be both to the left (resp., right) of q. Hence WLOG, assume a is to the right of q and b to the left of q. By the same argument as in the proof of Lemma 3.10, we can show that assuming a is not on any vertical obstacle edge leads to a contradiction. But if a is on a vertical obstacle edge e, then e must cross H_q (otherwise, the lower vertex of e on L_a would be above H_q, contradicting that $a \in V_g(q)$). This is possible only if e is on the rightmost projection cut-line of q. It is not difficult to see that in this case, a is actually a projection point of a vertex of $V_I \cup V_O$ to the right of e. ∎

COROLLARY 3.12. *For two gateways $a, b \in V_g(q)$ that are both above line H_q, suppose none of L_a or L_b contains a vertical obstacle edge that crosses H_q. Then $LN(L_a) > LN(L_b)$ implies $a_y \geq b_y$.*

Proof. An immediate consequence of Lemma 3.11. ∎

The ordering of the gateways of q by their y-coordinates is a key to our computation of $V_g(q)$ and its weighted edges to q. The next corollary shows that this ordering can be easily obtained.

COROLLARY 3.13. *Let $A(q)$ be the set of gateways of a point q above the horizontal line H_q, such that no gateway in $A(q)$ is on a vertical obstacle edge crossing H_q. If $A(q)$ is in increasing order of the level numbers of the projection cut-lines of q that contain the points of $A(q)$, then $A(q)$ is also in nondecreasing order of the y-coordinates of the points of $A(q)$.*

Proof. An immediate consequence of Corollary 3.12. ∎

The next lemma shows that the special case (i.e., for the at most two gateways on those projection cut-lines of q that contain a vertical obstacle edge crossing H_q)

can be handled easily.

LEMMA 3.14. *Let a be a gateway of a point q on a vertical obstacle edge e that crosses the horizontal line H_q. Then either q is in the obstacle that contains e or q is not contained in any obstacle.*

Proof. The proof follows immediately from Lemma 3.9. $\square$

Hence, computing the weight of the "inserted" edge (q, a), where a is defined in Lemma 3.14, is easy. The vertical segment contained in edge (q, a) is on an obstacle edge and the weighted horizontal segment of (q, a) is determined by Lemma 3.14. The following two lemmas lay the foundation for our iterative procedure to obtain $V_{\mathrm{g}}(q)$ and the weighted edges between q and other gateways in $V_{\mathrm{g}}(q)$.

LEMMA 3.15. *For gateway $a \in V_{\mathrm{g}}(q)$, suppose $LN(L_a)$ is the smallest among all projection cut-lines of q that do not contain a vertical obstacle edge crossing the horizontal line H_q. Let b be the point (a_x, q_y). Then either* (i) *the interior of the vertical segment $\overline{ab}$ does not intersect the interior of any obstacle, or* (ii) *segment $\overline{ab}$ is contained in the same obstacle that contains q.*

Proof. The interior of $\mathcal{R}(q)$ either intersects some horizontal obstacle edges or it does not. If the interior of $\mathcal{R}(q)$ intersects no obstacle edge, then clearly case i or ii holds. So assume the interior of $\mathcal{R}(q)$ does intersect a horizontal obstacle edge. Also, assume a is above line H_q.

Suppose we start at q and go vertically upward until we meet the first point z on any obstacle boundary. By Lemma 3.9, it is sufficient to show $z_y \geq a_y$. There are two cases to consider: (1) no projection cut-line of q contains a vertical obstacle edge that crosses H_q, and (2) a projection cut-line of q contains a vertical obstacle edge e that crosses H_q. WLOG, assume q is to the left of L_a.

To show case 1, we consider two subcases: (1.1) $LN(L_a) = 1$, and (1.2) $LN(L_a) > 1$.

Subcase 1.1. $LN(L_a) = 1$. Then no obstacle blocks q from being projected onto the level 1 cut-line L_a. Let v be the left vertex of the horizontal obstacle edge containing z. If v is projected onto L_a, then $z_y = v_y \geq a_y$ is true. Otherwise, the projection of v can only be blocked from L_a by a vertical obstacle edge e'. Edge e' must be between z and L_a, and it cannot cross H_q. By repeatedly applying this argument to the lower vertex u of e', we can show $z_y \geq u_y \geq a_y$.

Subcase 1.2. $LN(L_a) > 1$. Then a vertical obstacle edge e' must block q from being projected onto the level 1 cut-line. Let v be the upper vertex of edge e'. We have either $v_y \geq z_y$ or $v_y < z_y$. When $v_y < z_y$, if v is projected onto L_a, then the lemma holds; otherwise, v is blocked by a cut-line or a vertical obstacle edge from being projected onto L_a. If v is blocked by a cut-line L, then L must be between L_a and e', and $LN(L) < LN(L_a)$. But then q would have been projected onto L, contradicting that $LN(L_a)$ is the smallest among all the projection cut-lines of q. If the projection of v is blocked from L_a by a vertical obstacle edge e'', then the same argument can be applied to the lower vertex u of e'' to show $z_y \geq u_y \geq a_y$. When $v_y \geq z_y$, the right vertex u' of the horizontal obstacle edge containing z must be to the left of the vertical obstacle edge e', and by Lemma 3.9, u' can be projected onto L_a. Hence, $z_y = u'_y \geq a_y$.

In case 2, a projection cut-line of q contains a vertical obstacle edge e that crosses H_q. The following can be shown: (i) Let u be the upper vertex of e; then $u_y \geq z_y$. (ii) One vertex of the horizontal obstacle edge containing z must be projected onto L_a. The argument is similar to that for case 1. $\square$

LEMMA 3.16. *For two gateways $a, a' \in V_{\mathrm{g}}(q)$ that are both above line H_q, suppose*

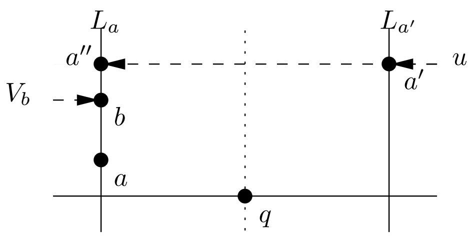


FIG. 3.7. *Illustration of Lemma* 3.16.

none of L_a and $L_{a'}$ contains a vertical obstacle edge crossing H_q, $LN(L_{a'}) > LN(L_a)$, and there is no projection cut-line L of q such that $LN(L_{a'}) > LN(L) > LN(L_a)$. Then the following are true (Figure 3.7*):* (1) *There is a Steiner point a'' on L_a such that $a'_y = a''_y$; furthermore, there is a vertex $u \in V_O \cup V_I$ such that u is projected on both a'' and a'.* (2) *For any Steiner point b on L_a such that $a''_y > b_y > a_y$, let vertex $v_b \in V_O \cup V_I$ be projected on b; then v_b and a' are on the opposite sides of L_a.* (3) *Let $p_1 = (a''_x, q_y)$ and $p_2 = (a'_x, q_y)$ be points on H_q; then segments $\overline{a''p_1}$ and $\overline{a'p_2}$ have the same weighted length.*

Proof. If $a_y = a'_y$, then (1) and (2) are trivially true, and (3) follows from Lemmas 3.9 and 3.10. So WLOG, assume $a_y < a'_y$ (Corollary 3.13) and L_a is to the left of $L_{a'}$. Let a'' be the point (a_x, a'_y).

The following claim is the key to proving (1), (2), and (3).

CLAIM. *Let $\mathcal{R}$ be the region enclosed by the boundary of $\mathcal{R}(q)$ and segments $\overline{a'a''}$ and $\overline{aa''}$, such that the interior of $\mathcal{R}$ does not intersect $\mathcal{R}(q)$. Then $\mathcal{R} - \overline{a'a''}$ does not intersect any vertical obstacle edge and does not contain any vertex in $V_O \cup V_I$.*

Proof of the claim. By contradiction. Suppose $\mathcal{R} - \overline{a'a''}$ contains a vertex $v \in V_O \cup V_I$. Then $a_y \le v_y < a'_y$ and v cannot be projected onto $L_{a'}$ because $a' \in V_{\mathrm{g}}(q)$. Hence the projection of v is blocked from $L_{a'}$ either by a cut-line $L \neq L_a$ with $LN(L) < LN(L_{a'})$, or by a vertical obstacle edge e not intersecting the interior of $\mathcal{R}(q)$ (by Lemma 3.9). In the former case, L must be between L_a and $L_{a'}$, and hence L cannot contain a vertical obstacle edge crossing H_q. If $LN(L) \le LN(L_a)$, then q would have not been projected onto either L_a or $L_{a'}$, a contradiction. If $LN(L) > LN(L_a)$ (but $LN(L) < LN(L_{a'})$), then q would have been projected on L, contradicting that there is no projection cut-line L of q such that $LN(L_{a'}) > LN(L) > LN(L_a)$. Hence, the projection of v cannot be blocked from $L_{a'}$ by a cut-line. If the projection of v is blocked from $L_{a'}$ by a vertical obstacle edge e which does not intersect the interior of $\mathcal{R}(q)$, then the lower vertex u' of e must be in $\mathcal{R} - \overline{a'a''}$. The argument can then be repeated on u', eventually showing that such a vertex u' must be projected on $L_{a'}$, a contradiction to $a' \in V_{\mathrm{g}}(q)$. Therefore, no obstacle vertex of $V_O \cup V_I$ can be in $\mathcal{R} - \overline{a'a''}$. This argument can also be used to show that no vertical obstacle edge intersects $\mathcal{R} - \overline{a'a''}$. ☐

Let $v_{a'} \in V_O \cup V_I$ be projected onto a'. Then $v_{a'}$ cannot be to the left of L_a; otherwise, since a' is to the right of L_a and $LN(L_a) < LN(L_{a'})$, $v_{a'}$ would have not been projected onto $L_{a'}$. So $v_{a'}$ must be to the right of L_a. Then by Lemma 3.9 and the above claim, $v_{a'}$ must be projected onto a'' on L_a as well. Hence, (1) is true (with $u = v_{a'}$).

For (2), if v_b is to the right of L_a (as a'), then v_b would have been projected onto L'_a because the interior of $\mathcal{R} \cup \mathcal{R}(q)$ contains no obstacle vertex and does not intersect any vertical obstacle edge, thus contradicting that $a' \in V_{\mathrm{g}}(q)$. Hence, v_b can only be to the left of L_a.

For (3), because any horizontal obstacle strip intersected by segment $\overline{a'p_2}$ cannot stop within $\mathcal{R}$ and must run through $\mathcal{R}$, segment $\overline{a''p_1}$ also intersects such a horizontal obstacle strip. By the same argument, if $\overline{a''p_1}$ intersects any horizontal obstacle strip, then such a horizontal strip must also be intersected by $\overline{a'p_2}$. Hence, the weighted lengths of both $\overline{a''p_1}$ and $\overline{a'p_2}$ are the same. $\square$

3.3.2. Point q on a vertical obstacle edge. If point q is on a vertical obstacle edge e, then we reduce this case to the case in which q is not on any vertical obstacle edge, as follows: Treat q as if q_x is perturbed to the left (resp., right) of e by some very small value $\epsilon > 0$, and then apply the observations and procedures for the case with q not on any vertical obstacle edge.

In the rest of this section, we assume WLOG that point q is not on any vertical obstacle edge.

3.4. Computing gateways and weighted edges to gateways. First, we need to find all projection cut-lines of point q. These cut-lines can be identified easily by using a data structure based on trapezoidal decomposition and planar point location [12] in $O(\log n)$ time. By Lemma 3.4, the x-coordinates of these cut-lines can be put in sorted order by bucket sort in $O(\log n)$ time. Let $L_{i_1}, L_{i_2}, \ldots, L_{i_g}$ be the sequence of all projection cut-lines of q in increasing order of their level numbers, and let $a_{i_j} \in V_{\mathrm{g}}(q)$ be on L_{i_j} that is above line H_q, with $g = O(\log n)$.

Next, we compute the gateways a_{i_j} and weighted edges (q, a_{i_j}). Let p_{i_j} be the projection point of q on projection cut-line L_{i_j} of q. Since edge (q, a_{i_j}) consists of two segments $\overline{qp_{i_j}}$ and $\overline{p_{i_j}a_{i_j}}$ and since the interior of $\overline{qp_{i_j}}$ is either contained completely in an obstacle or outside any obstacle, it is sufficient to show how to compute a_{i_j} and the weighted length $d_w(\overline{p_{i_j}a_{i_j}})$. We first find the gateways that are either on the projection cut-lines of q that contain a vertical obstacle edge crossing line H_q or on the projection cut-line of q that has the smallest level number among the projection cut-lines that do not contain a vertical obstacle edge crossing H_q. There are at most three such gateways and they can all be obtained by binary search on the vertices of G on the relevant projection cut-lines of q in $O(\log n)$ time. The weights of edges from q to these three gateways can be easily computed by Lemmas 3.14 and 3.15. Then, based on Lemma 3.16, we show that given a_{i_j} and $d_w(\overline{p_{i_j}a_{i_j}})$, $a_{i_{j+1}}$ and $d_w(\overline{p_{i_{j+1}}a_{i_{j+1}}})$ can both be obtained in $O(1)$ time. Our data structure for computing gateways and the weights of their edges to q maintains the following information for every cut-line L.

1. For any two vertices u and v of G on L, $d_w(\overline{uv})$ can be reported in $O(1)$ time. This is done by first using Widmayer's algorithm [30] to compute the weighted length between every two consecutive vertices of G on L and then by performing a prefix sum operation, along the vertices of G on L, to compute the weighted length from every vertex on L to the highest vertex on L. From the prefix sums of u and v, it is easy to find $d_w(\overline{uv})$ in $O(1)$ time.

2. Each vertex u of G on L keeps track of all vertices z in $V_O \cup V_I$ such that z is projected on u. Note that u can be the projections of vertices in $V_O \cup V_I$ on each side of L.

3. For each $z \in V_O \cup V_I$, maintain an array Proj_z of size $O(\log n)$. If z is projected to point $p(z)$ on cut-line L of level i, then $\mathrm{Proj}_z(i)$ has a pointer to $p(z)$ on L; otherwise, $\mathrm{Proj}_z(i)$ is undefined.

4. Each vertex u of G on L keeps track of the lowest vertex v of G on L such that $v_y \geq u_y$ and v is a projection of a vertex of $V_O \cup V_I$ that is to the left (resp., right) of L (v can be u).

Note that the graph construction procedure [24] can be easily modified to compute the information in items 1–4 above in $O(n\log n)$ time and space. Now based on Lemma 3.16, it is an easy matter to compute $a_{i_{j+1}}$ and $d_w(\overline{p_{i_{j+1}}a_{i_{j+1}}})$ from a_{i_j} and $d_w(\overline{p_{i_j}a_{i_j}})$, in $O(1)$ time. WLOG, assume L_{i_j} is to the left of $L_{i_{j+1}}$. We perform the following steps: (1) Find the lowest vertex a'' of G on L_{i_j} such that a'' is on or above a_{i_j} and that a'' is a projection point of a vertex $z \in V_O \cup V_I$ with z being to the right of L_{i_j}; (2) let $a_{i_{j+1}} = \mathrm{Proj}_z(LN(L_{i_{j+1}}))$ and $d_w(\overline{p_{i_{j+1}}a_{i_{j+1}}}) = d_w(\overline{p_{i_j}a_{i_j}}) + d_w(\overline{a_{i_j}a''})$.

Based on the above discussion, for any point q, we compute the gateway set $V_{\mathrm{g}}(q)$ and the weighted edges from all vertices of $V_{\mathrm{g}}(q)$ to q in $O(\log n)$ time.

3.5. Answering shortest path queries.

This subsection gives our procedures for answering single-source and two-point shortest path queries.

3.5.1. Single-source queries.

A single-source query requests a shortest path from the fixed source point s to an arbitrary query point t. Since s is fixed, it can be included as a vertex of graph G. The query procedure compares the lengths of the s-to-q shortest paths through the gateways of $V_{\mathrm{g}}(q)$ and selects the smallest path length for all $q \in Q(t) \cup \{t\}$. Precisely, the length of such a shortest s-to-q path is given by $\min\{d_w(P_{sv})+w(q,v) \mid v \in V_{\mathrm{g}}(q)\}$, where $d_w(P_{sv})$ is the length of a weighted shortest s-to-v path and $w(q,v)$ is the weight of the "inserted" edge connecting q and v. Since $|V_{\mathrm{g}}(q)| = O(\log n)$ and $V_{\mathrm{g}}(q)$ (together with the weighted edges between q and $V_{\mathrm{g}}(q)$) can be obtained in $O(\log n)$ time, a single-source length query is performed in $O(\log n)$ time.

An actual s-to-q shortest path consists of the path (with at most two segments) between q and a vertex $x \in V_{\mathrm{g}}(q)$ and a path in graph G from x to s. We already show how to compute the path between q and x in $O(\log n)$ time. The path between x and s in G can be reported by standard single-source shortest path tree methods in $O(k)$ time, where k is the number of edges on that path.

In the rest of the paper, we will no longer specify the details on reporting actual shortest paths, since these can all be carried out based on standard shortest path techniques.

3.5.2. Two-point queries.

A two-point query requests a shortest path between two arbitrary query points s and t. Our two-point query procedure must account for two cases: There is a shortest s-to-t path going through a vertex in G and there is no shortest s-to-t path going through any vertex of G. We then select the true shortest s-to-t path from the two candidates so obtained.

To obtain a shortest s-to-t path going through a vertex in G, we compute a shortest p-to-q path that goes through $V_{\mathrm{g}}(p)$ and $V_{\mathrm{g}}(q)$, for every pair p and q, $p \in Q(s) \cup \{s\}$ and $q \in Q(t) \cup \{t\}$. A shortest p-to-q path through $V_{\mathrm{g}}(p)$ and $V_{\mathrm{g}}(q)$ is computed as follows. First, compute $V_{\mathrm{g}}(p)$ (resp., $V_{\mathrm{g}}(q)$) for point p (resp., q) and the weighted edges between $V_{\mathrm{g}}(p)$ and p (resp., $V_{\mathrm{g}}(q)$ and q), as shown in section 3.4. Then, create a graph G_{pq} as follows: The vertices of G_{pq} are $\{p,q\} \cup V_{\mathrm{g}}(p) \cup V_{\mathrm{g}}(q)$; the edges of G_{pq} consist of the weighted edges between $V_{\mathrm{g}}(p)$ and p, the weighted edges between $V_{\mathrm{g}}(q)$ and q, and the edges (v_p, v_q), where $v_p \in V_{\mathrm{g}}(p)$, $v_q \in V_{\mathrm{g}}(q)$, and the weight of an edge (v_p, v_q) is the length of a shortest v_p-to-v_q path in G. It is easy to see that G_{pq} has $O(\log n)$ vertices and $O(\log^2 n)$ edges. Finally, find a shortest p-to-q path in G_{pq} by Fredman and Tarjan's shortest path algorithm [14]. This obtains the length of a shortest p-to-q path in $O(\log^2 n)$ time.

Computing a shortest s-to-t path not going through any vertex of G is much easier. By the lemmas in section 3.1, it is clear that such a path can be obtained from the shortest p-to-q path that consists of at most two line segments, for every pair p and q, $p \in Q(s) \cup \{s\}$ and $q \in Q(t) \cup \{t\}$. WLOG, assume p and q are not on the same vertical and horizontal line. Note that there can be only two such two-segment paths between p and q, one through the point (p_x, q_y) and the other through (q_x, p_y). Also, note that each such two-segment path may still penetrate many obstacles. Obviously, the following is a key operation for computing such two-segment paths: Given two points a and b that are on a same vertical or horizontal line, compute the weighted length of segment $\overline{ab}$. We use a grid-partition approach to perform this operation.

We partition the plane by using a set S_E of $O(n)$ lines such that each line in S_E contains a vertical or horizontal obstacle edge. The plane is thus divided into $O(n^2)$ rectangular grid cells with a weight factor associated with each cell. The following is easy to observe: Suppose we sweep a row (resp., column) of grid cells by a horizontal (resp., vertical) line L without letting L touch the horizontal (resp., vertical) boundaries of that row (resp., column); then for any two points a and b on L, the weighted length of segment $\overline{ab}$ remains unchanged during the sweeping.

We maintain the following information with the grid. (We only show the horizontal case.) (1) For every row X of the grid, let L_X be an arbitrary horizontal line strictly between the two horizontal boundaries of X; let S'_E be the set of such horizontal lines L_X for all the $O(n)$ rows of the grid. (2) The horizontal lines in $S_E \cup S'_E$ intersect the vertical lines in S_E at $O(n^2)$ points, which we call *special points.* (3) Along the row of special points on each horizontal line L in $S_E \cup S'_E$, perform a prefix sum computation on their weighted lengths along L to the leftmost special point. This grid-partitioning takes $O(n^2)$ time and space. Using the grid-based data structure, it is easy to compute the weighted length of any horizontal or vertical segment $\overline{ab}$ in $O(\log n)$ time.

In summary, each two-point path length query can be answered in $O(\log^2 n)$ time.

4. Constructing shortest path data structures.

This section focuses on the construction of our shortest path data structures. Our data structures for both single-source and two-point queries make use of graph G which has $O(n\log n)$ vertices and edges. If Fredman and Tarjan's shortest path algorithm [14] were directly applied to G, then computing a shortest path tree in G (for single-source queries) would take $O(n \log^2 n)$ time and $O(n\log n)$ space, and computing $O(n\log n)$ shortest path trees in G (for two-point queries) would take $O(n^2 \log^3 n)$ time and $O(n^2 \log^2 n)$ space. By using an implicit graph representation scheme and divide-and-conquer paradigms, we are able to construct the single-source data structure in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space, and the two-point data structure in $O(n^2 \log^2 n)$ time and space. Note that graph G' (discussed in section 2) plays a crucial role in computing shortest path trees in G.

Since other components of our shortest path data structures have been discussed in section 3, this section only shows how to compute shortest path trees in G in the claimed time and space bounds.

4.1. Distance tables.

Recall that a major difference between graph G' and the generalized visibility graph G is the addition of a set E_S of $O(n \log^{3/2} n)$ edges to G'. The edges in E_S were represented *explicitly* using $O(n \log^{3/2} n)$ space in [24]. We use an *implicit* scheme to store E_S in $O(n\log n)$ space, which still enables us to retrieve information about G' as quickly (within a constant factor) as using the explicit representation. We call such a scheme *distance tables.*

The construction of G' is based on partitioning every cut-line L into strips, with each strip containing $O(\log^{1/2} n)$ Steiner points on L. Let S be the set of Steiner points on L within a strip of L, and let $PV_l(S)$ (resp., $PV_r(S)$) be an array containing the $O(\log^{1/2} n)$ vertices of $V_O \cup V_I$ within the strip that are to the left (resp., right) of L and are visible from L. We associate a distance table T_S with the strip of L for S. Note that for each vertex $u \in PV_l(S)$ and $v \in PV_r(S)$, there is an edge (u, v) in G' consisting of segments $\overline{up_L(u)}$, $\overline{p_L(u)p_L(v)}$, and $\overline{p_L(v)v}$, where $p_L(z)$ denotes the projection of a vertex $z \in PV_l(S) \cup PV_r(S)$ on L. Edges between $PV_l(S)$ and $PV_r(S)$ can be implicitly represented by arrays $PV_l(S)$ and $PV_r(S)$. The weights of these edges can also be represented implicitly, as follows: For every $z \in PV_l(S) \cup PV_r(S)$, store $p_L(z)$ and the weighted length of $\overline{zp_L(z)}$ in table T_S; for any $p_L(u)$ and $p_L(v)$ in S, use the prefix sums of weighted lengths discussed in section 3.4 to compute the weighted length of $\overline{p_L(u)p_L(v)}$, in $O(1)$ time. Hence, the weight of every edge (u, v), for $u \in PV_l(S)$ and $v \in PV_r(S)$, can be computed in $O(1)$ time via T_S. T_S uses $O(\log^{1/2} n)$ space. The distance tables for all $O(n \log^{1/2} n)$ strips of G use altogether $O(n\log n)$ space and can be trivially built in $O(n\log n)$ time. Running the shortest path algorithm of [14] on G' with edge set E_S being represented by distance tables is similar to the explicit representation of G' in [24], yet it takes $O(n \log^{3/2} n)$ (not $O(n \log^2 n)$) time and $O(n\log n)$ (not $O(n \log^{3/2} n)$) space.

4.2. Construction of the single-source data structure. The single-source shortest path tree in graph G rooted at the source vertex s is computed by making use of graph G'. The procedure for computing this shortest path tree in G consists of two steps.

1. Compute the single-source shortest path tree in G' rooted at s, in $O(n \log^{3/2} n)$ time and $O(n\log n)$ space (by using our distance table scheme). To obtain the shortest path tree in G rooted at s, we need to compute, for every cut-line L, the lengths of shortest paths from s to all Steiner points on L in each strip of L. Let S be the set of Steiner points on L within a strip of L, and let $PV(S)$ denote the vertices of $V_O \cup V_I$ within the strip that are visible from L. Observe that for each point $u \in S$, $PV(S)$ controls all paths in G from u to the source s, and that the lengths of shortest paths from the vertices in $PV(S)$ to s are already computed in this step.

2. Let u_m be the median Steiner point in S along L. Let $d_w(P^*_{u_m s}) = \min\{d_w(P_{u_m v}) + d_w(P^*_{vs}) \mid v \in PV(S)\}$, where P^*_{wz} denotes a shortest path between points w and z, and path $P_{u_m v}$ consists of segments $\overline{vp_L(v)}$ and $\overline{p_L(v)u_m}$. It is easy to compute $d_w(P^*_{u_m s})$ in $O(|PV(S)|)$ time. Next, use u_m to partition S into two subsets S_1 and S_2 of balanced sizes. Note that for a point $u' \in S_1$ (resp., S_2), all paths in G from u' to s are controlled by the vertices in $PV(S_1) \cup \{u_m\}$ (resp., $PV(S_2) \cup \{u_m\}$). Therefore, we recursively perform step 2 on S_1 and S_2, resp., until the size of each set becomes one.

The time bound of the divide-and-conquer procedure for step 2 obeys recurrence $T(N) = 2T(N/2) + cN$ for some constant $c > 0$. Because $|PV(S)| = O(\log^{1/2} n)$ for each strip, the time of the procedure taken on each strip is $O(\log^{1/2} n \log\log n)$. The total time of step 2 on all $O(n \log^{1/2} n)$ strips of G is $O(n\log n \log\log n) = o(n \log^{3/2} n)$, and the space is $O(n\log n)$.

4.3. Construction of the two-point data structure. For each of the $O(n\log n)$ vertices in G, we need to compute a single-source shortest path tree in G rooted at that vertex. The $O(n\log n)$ shortest path trees in G altogether take $O(n^2 \log^2 n)$ space. Applying the algorithm in section 4.2 for computing one shortest path tree to each vertex of G would take $O(n^2 \log^2 n \log\log n)$ time. We show here how to achieve an $O(n^2 \log^2 n)$ time and space algorithm for obtaining the $O(n\log n)$ shortest path trees in G. The algorithm consists of two steps.

1. Compute the lengths of shortest paths from all vertices in G' to all vertices in G. There are $O(n \log^{1/2} n) \times O(n\log n)$ path lengths to compute. By applying the algorithm in section 4.2 to each vertex of G', this step takes $O(n \log^{1/2} n) \times O(n \log^{3/2} n) = O(n^2 \log^2 n)$ time and $O(n^2 \log^{3/2} n)$ space.

2. Compute the lengths of shortest paths between the vertices of G that are not vertices of G'. Let S' (resp., S'') be the set of Steiner points on a cut-line L' (resp., L'') within a strip of L' (resp., L''). The $O(\log n)$ path lengths between the points in S' and the points in S'' are computed as follows: (2.1) Let u'_m (resp., u''_m) be the median Steiner point in S' (resp., S''), and compute the lengths of shortest paths from u'_m (resp., u''_m) to all the points in S'' (resp., S'). This can be done in $O(\log^{1/2} n \log\log n)$ time by using a divide-and-conquer procedure similar to the one in section 4.2. (2.2) Use u'_m (resp., u''_m) to partition S' (resp., S'') into two subsets S'_1 and S'_2 (resp., S''_1 and S''_2) of balanced sizes. (2.3) Recursively repeat this procedure on each pair of subsets S'_i and S''_j, with $i, j \in \{1, 2\}$, until the size of a set in the pair becomes one.

The time bound of the divide-and-conquer procedure for step 2 obeys recurrence $T(N) = 4T(N/2) + cN \log N$ for some constant $c > 0$, whose solution is $T(N) = O(N^2)$. Since there are $O(\log^{1/2} n)$ Steiner points in each strip of a cut-line, the time of the procedure for computing the $O(\log n)$ path lengths for each pair of strips is $T(\log^{1/2} n) = O(\log n)$. Hence, the total time of step 2 on all the $O(n^2\log n)$ pairs of strips of G is $O(n^2 \log^2 n)$, and the space is also $O(n^2 \log^2 n)$.

5. Shortest path queries among arbitrary polygonal obstacles.

This section focuses on processing two-point L_1 shortest obstacle-avoiding path queries among arbitrary (i.e., not necessarily rectilinear) polygonal obstacles. We call nonpenetrating obstacles *solid* obstacles. The query processing techniques that we will present have some similarity to those used in the previous sections, but nevertheless significant differences exist in our data structures and algorithms for this case. One notable difference in the solid polygonal obstacle case is that, corresponding to the nonrectilinear shape of obstacles, we will not limit ourselves to using rectilinear paths, although we still use the L_1 metric to measure the lengths of the (possibly nonrectilinear) paths. The line segments of nonrectilinear paths can be viewed as representing staircase subpaths consisting of many sufficiently short line segments (if rectilinear paths are desired).

We will show how to construct a two-point shortest path data structure in $O(n^2 \log^2 n)$ time and $O(n^2 \log n)$ space, which answers each two-point length query in $O(\log^2 n)$ time and reports an actual path in an additional $O(k)$ time, where k is the number of edges on the output path.

5.1. The visibility graph G^*.

In this section, we say a point q is *visible* from a point p iff line segment $\overline{qp}$ does not intersect the interior of any obstacle, and q is *rectilinearly visible* from p iff p and q are visible to each other and $\overline{pq}$ is horizontal or vertical. The visibility graph $G^* = (V^*, E^*)$, which we use for the solid polygonal obstacle case, is similar to graph G defined in section 2 for the weighted rectilinear obstacle case. But there are a few differences which account for the facts that the obstacles are solid and that the obstacle edges need not be rectilinear. This visibility graph was introduced in [8, 9]. The vertex set V^* of G^* includes: (i) the set of obstacle vertices and (ii) two sets of Steiner points.

As in [8, 9], there are two types of Steiner points in G^*: (1) projections of obstacle vertices onto cut-lines, and (2) projections of obstacle vertices onto obstacle boundaries. The first type of Steiner points (called *type*-1 *Steiner points*) are created in [8, 9] similar to the way sketched in section 2, building a graph with $O(n \log n)$ vertices and edges, as well as other information (e.g., cut-lines). The second type of Steiner points (called *type*-2 *Steiner points*) consists of horizontal and vertical projection points of each obstacle vertex p onto rectilinearly visible obstacle edges of p. Include edges in E^* between every obstacle vertex p and each of the (at most four) projection points of p. For each obstacle edge e, there is a linear ordering of the vertices in V^* which lie on e; include in E^* edges between every two consecutive vertices of V^* along e. Clearly, there are $O(n)$ type-2 Steiner points and they introduce $O(n)$ edges to E^*. These type-2 Steiner points and their edges can be created in $O(n \log n)$ time by a standard plane sweeping technique [8, 9].

5.2. Useful observations. We project each query point z vertically (resp., horizontally) onto the first point of the obstacle edge (if it exists) immediately above (resp., below, to the left of, to the right of) z. Let $Q(z)$ be the set of the (at most four) projection points of z on obstacle boundaries so resulted. The lemmas below show some useful structures of shortest paths between two points q and r.

LEMMA 5.1. *For two points q and r that are not visible to each other, there is a q-to-r shortest path whose interior passes through an obstacle vertex.*

Proof. The proof is obvious and omitted. □

Next, we characterize shortest paths between points that are visible to each other. Before doing so, we should point out that by simply performing a ray shooting operation (in any given direction), it is easy to find out whether two points in the plane are visible to each other, and if they are, give the shortest path (i.e., the segment) connecting the two points. Such a ray shooting data structure [1] takes $O(n^2)$ space and $O(n^2 \log n)$ time to construct, in order to support each operation in logarithmic time. In our case, it is possible to avoid such general ray shooting operations and use only special ray shooting (along the horizontal or vertical direction). We believe this simplifies the algorithm and even improves its implementation performance.

Let two points q and r be visible to each other. Then a shortest q-to-r path P_{qr} can be classified into one of three possible cases: (1) P_{qr} goes through an obstacle vertex and hence through a vertex of G^* (by possibly going through a point in $Q(q) \cup Q(r)$); (2) P_{qr} goes through a point in $Q(q) \cup Q(r)$ but not through any obstacle vertex of G^*; or (3) P_{qr} goes through neither a point in $Q(q) \cup Q(r)$ nor an obstacle vertex. Note that, even by using general ray shooting, it may be difficult to decide which case holds for each pair of mutually visible points. Fortunately, it turns out that it is not necessary to decide exactly which case holds. The next lemma shows that, for all practical purposes, case 1 can be differentiated from cases 2 and 3 without much trouble.

LEMMA 5.2. *Suppose two points q and r are visible to each other. By performing $O(1)$ special ray shooting operations (horizontal and vertical), it is possible to decide whether a shortest q-to-r path P_{qr} needs to be forced to go through an obstacle vertex.*

Proof. WLOG, assume r is in the first quadrant of q. In this case, what we need to do is to perform an upward (resp., downward) vertical ray shooting and a rightward (resp., leftward) horizontal ray shooting from q (resp., r). We only show how to analyze the information obtained by the leftward shooting from q and the downward shooting from r (the other case is similar).

There are three cases for the leftward shooting from q and the downward shooting from r: (a) The rays from q and r intersect each other before hitting an obstacle edge,

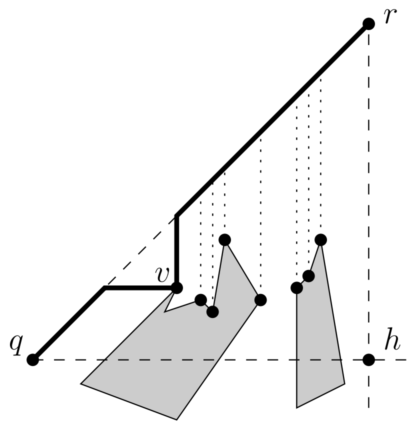


FIG. 5.1. *A shortest path P_{qr} goes through an obstacle vertex.*

(b) the rays hit the same obstacle edge e, and (c) the rays hit different obstacle edges.

In case (a), one can simply use a shortest q-to-r path that consists of segments $\overline{qh}$ and $\overline{hr}$, where h is the intersection point of the two rays. In case (b), a shortest q-to-r path consisting of three segments $\overline{qh_q}$, $\overline{h_qh_r}$, and $\overline{h_rr}$ can be used, where h_q (resp., h_r) is the hit point of the horizontal (resp., vertical) ray from q (resp., r) on the (same) obstacle edge e.

In case (c), a shortest q-to-r path P_{qr} can be forced to go through an obstacle vertex. Since q and r are visible to each other, segment $\overline{qr}$ does not intersect the interior of any obstacle. Let h be the intersection point of the two rays from q and r as if they are extended to infinity (Figure 5.1). Because the two rays do not hit the same obstacle edge before they intersect each other, the triangle $\triangle qrh$ must contain at least one obstacle vertex. Consider the obstacle vertices in $\triangle qrh$ that are vertically visible from a point on segment $\overline{qr}$. Let v be the first of such vertices in the left-to-right order based on their corresponding vertically visible points on $\overline{qr}$ (Figure 5.1). A shortest q-to-r path P_{qr} can then be obtained as in Figure 5.1, which goes through obstacle vertex v. $\square$

The horizontal and vertical ray shooting operations can be performed in $O(\log n)$ time each. This is done by using a planar point location data structure [12] which is based on a horizontal (resp., vertical) trapezoidal decomposition of the plane.

The significance of the above lemmas is as follows. For any two points q and r, regardless of whether they are visible to each other or not, a shortest q-to-r path can be obtained in one of two ways: (i) such a path can be forced to go through an obstacle vertex (Lemma 5.1 and case c in the proof of Lemma 5.2), and (ii) such a path need not go through any obstacle vertex (cases (a) and (b) in the proof of Lemma 5.2). Lemma 5.2 also shows how to handle paths of type ii efficiently. All other paths (i.e., type i) are captured by graph G^*, as shown in Lemma 5.1 and case (c) of the proof of Lemma 5.2. The next subsection further discusses how to handle two-point path queries of type i.

5.3. Characterization of gateways. Let p and q be any two query points. WLOG, we assume in this section a shortest p-to-q path goes through a vertex of G^*. As in the weighted rectilinear obstacle case, we want to compute a subset of vertices in G^* (the gateways) which captures a shortest p-to-q path. But our data structure for maintaining shortest path information for each pair of vertices in G^* is somewhat different from the weighted rectilinear obstacle case. Our two-point query procedure is also different. For the time being, we assume our data structure can report the length of a shortest path between any two vertices of G^* in $O(\log n)$ time. We again use the idea of "inserting" the query points into the visibility graph.

The set $V_{\mathrm{g}}(q)$ of gateways for a point q is defined similarly to the one for the weighted rectilinear obstacle case but differs in that the visibility graph for the solid polygonal obstacle case must account for the type-2 Steiner points. We therefore split $V_{\mathrm{g}}(q)$ into two subsets $V'_{\mathrm{g}}(q)$ of gateways with the type-1 Steiner points, and $V''_{\mathrm{g}}(q)$ of gateways with the type-2 Steiner points.

In the rest of this section, we assume WLOG that all cut-lines are vertical unless otherwise specified.

The set $V'_{\mathrm{g}}(q)$ of gateways for a point q is determined as follows: For each cut-line L of q that is horizontally visible from q, if z is the vertex of G^* on L that is immediately above (resp., below) the horizontal projection point $p_L(q)$ of q on L and if z is visible to $p_L(q)$, then $z \in V'_{\mathrm{g}}(q)$. For $V''_{\mathrm{g}}(q)$, shoot a ray from q onto the obstacle edge immediately above (resp., to the left of, to the right of, below) q; if u is the vertex of G^* that is immediately before (resp., after) the projection point of q on the obstacle edge, then $u \in V''_{\mathrm{g}}(q)$.

For each $z \in V'_{\mathrm{g}}(q)$ on a cut-line L of q, let an "edge" (q, z) in the graph consist of segments $\overline{qp_L(q)}$ and $\overline{p_L(q)z}$. If $z \in V''_{\mathrm{g}}(q)$, then edge (q, z) consists of segments $\overline{qp_e(q)}$ and $\overline{p_e(q)z}$, where $p_e(q)$ is the projection point of q on the obstacle edge e containing z. It is now easy to see that $V_{\mathrm{g}}(q)$ controls every path in G^* from q to any other vertex of G^*, and $|V_{\mathrm{g}}(q)| = O(\log n)$.

The *gateway region* $\mathcal{R}(q)$ of q (Figure 3.4) is defined and constructed in the same manner as the weighted rectilinear obstacle case (section 3.3), with the minor difference that here it is based on the vertices in $V'_{\mathrm{g}}(q)$ instead of $V_{\mathrm{g}}(q)$.

LEMMA 5.3. *The gateway region $\mathcal{R}(q)$ is rectilinearly convex and intersects no interior points of any obstacle. Furthermore, for any point z in $\mathcal{R}(q)$, the points (z_x, q_y) and (q_x, z_y) are both in $\mathcal{R}(q)$.*

Proof. The proof is similar to those of Lemmas 3.7 and 3.9. □

Comparing with the weighted case discussed in previous sections, it is straightforward to compute in this case $V_{\mathrm{g}}(q)$ and the L_1 distances from q to all points in $V_{\mathrm{g}}(q)$ in $O(\log n)$ time, by Lemma 5.3.

5.4. Construction of the data structure. In the weighted rectilinear obstacle case, our processing of queries depends on the ability to report efficiently the shortest paths (or their lengths) between any two vertices in the visibility graph. In the polygonal obstacle case, rather than applying graphical shortest path algorithms to graph G^* explicitly, we maintain needed path information by using the shortest path maps in [25, 26]. In effect, we need the geometric properties of graph G^* for query purposes, but we do not need its graph structures for computing our data structure for shortest paths between vertices of G^*.

Mitchell [25, 26] created an optimal single-source L_1 shortest path map in $O(n \log n)$ time and $O(n)$ space, where n is the number of obstacle vertices. Given any query point t, Mitchell's data structure can report the length of a shortest path from t to the source point s in $O(\log n)$ time (by planar point-location within the map), and an actual t-to-s shortest path in an additional $O(k)$ time, where k is the number of segments on the output path.

We apply Mitchell's algorithm [25, 26] to create $O(n \log n)$ L_1 single-source shortest path maps. In particular, we use each vertex $v \in V^*$ as a source point to construct a single-source shortest path map. Such a map is of size $O(n)$ because its geometric setting consists of n obstacle vertices (note that we need not use the $O(n \log n)$ Steiner points of V^* in building this shortest path map). This map allows us to handle single-source queries between the source $v \in V^*$ and any query point q.

Our two-point shortest path data structure for the solid polygonal obstacle case contains as a major component the $O(n \log n)$ shortest path maps. Using these maps, a length query of the shortest path between any vertex $v \in V^*$ and any query point p in the plane can be answered in $O(\log n)$ time. Our data structure can be created in altogether $O(n \log n) \times O(n \log n) = O(n^2 \log^2 n)$ time and $O(n \log n) \times O(n) = O(n^2 \log n)$ space. This is a savings of a factor of $\log n$ in space complexity over the methods we used in previous sections.

5.5. Answering shortest path queries. To answer a shortest path query for any two points p and q, we need to account for two cases: (i) a shortest p-to-q path goes through a vertex in G^*, and (ii) the shortest p-to-q path need not go through any vertex of G^*. The proof of Lemma 5.2 has shown how to distinguish these two cases. This subsection shows that a length query can be performed in $O(\log^2 n)$ time.

Suppose, by Lemma 5.2, we already know that a shortest p-to-q path goes through a vertex of G^* for two query points p and q. To obtain the length of such a shortest path, we use the shortest path maps constructed in section 5.4, as follows:

1. For one of the two query points, say q, compute $V_{\mathrm{g}}(q)$.
2. For each point $x \in V_{\mathrm{g}}(q)$, use p as a query point to find, in the shortest path map with x as the source point, the length of the shortest x-to-p path in the plane (in $O(\log n)$ time).
3. There are $O(\log n)$ paths, each consisting of the (trivial) shortest path from q to an $x \in V_{\mathrm{g}}(q)$ and the shortest x-to-p path. Find the length of the shortest p-to-q path among these paths.

This computation clearly takes altogether $O(\log^2 n)$ time. The correctness follows from the facts that a shortest p-to-q path goes through a point (say y) in $V_{\mathrm{g}}(q)$ and that the p-to-y-to-q path so obtained consists of a shortest q-to-y path and a shortest y-to-p path.

The computation of a shortest p-to-q path not going through any vertex of G^* is much simpler. From Lemma 5.2 in section 5.2, such a path and its length can be found in $O(\log n)$ time (by performing $O(1)$ horizontal and vertical ray shootings).

REFERENCES

[1] P. K. Agarwal, *Ray shooting and other applications of spanning trees with low stabbing number*, SIAM J. Comput., 21 (1992), pp. 540–570.

[2] S. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. Smid, and C. D. Zaroliagis, *Planar spanners and approximate shortest path queries among obstacles in the plane*, in Algorithms—ESA '96, Barcelona, Spain, Lecture Notes in Comput. Sci. 1136, J. Diaz and M. Serna, eds., Springer-Verlag, New York, 1996, pp. 514–528.

[3] M. J. Atallah and D. Z. Chen, *Parallel rectilinear shortest paths with rectangular obstacles*, Comput. Geom., 1 (1991), pp. 79–113.

[4] M. J. Atallah and D. Z. Chen, *On parallel rectilinear obstacle-avoiding paths*, Comput. Geom., 3 (1993), pp. 307–313.

[5] B. Chazelle and L. J. Guibas, *Fractional cascading:* I. *A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[6] D. Z. Chen, *On the all-pairs Euclidean short path problem*, in Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1995, ACM, New York, pp. 292–301.

[7] Y. Chiang, F. P. Preparata, and R. Tamassia, *A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps*, SIAM J. Comput., 25 (1996), pp. 207–233.

[8] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, *Rectilinear Shortest Paths through Polygonal Obstacles in $O(n \log^{3/2} n)$ Time*, manuscript, 1989.

[9] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, *Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time*, in Proceedings of the Third Annual ACM Symposium on Computational Geometry, ACM, New York, 1987, pp. 251–257.

[10] M. de Berg, M. van Kreveld, and B. J. Nilsson, *Shortest path queries in rectangular worlds of higher dimension*, in Proceedings of the Seventh Annual ACM Symposium on Computational Geometry, North Conway, NH, 1991, pp. 51–59.

[11] P. J. de Rezende, D. T. Lee, and Y. F. Wu, *Rectilinear shortest paths in the presence of rectangular barriers*, Discrete Comput. Geom., 4 (1989), pp. 41–53.

[12] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[13] H. ElGindy and P. Mitra, *Orthogonal shortest route queries among axes parallel rectangular obstacles*, Internat. J. Comput. Geom. Appl., 4 (1994), pp. 3–24.

[14] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[15] L. Gewali, A. Meng, J. S. B. Mitchell, and S. Ntafos, *Path planning in $0/1/\infty$ weighted regions with applications*, in Proceedings of the Fourth Annual ACM Symposium on Computational Geometry, Urbana–Champaign, Illinois, 1988, pp. 266–278.

[16] M. T. Goodrich and R. Tamassia, *Dynamic ray shooting and shortest paths in planar subdivisions via balanced geodesic triangulations*, J. Algorithms, 23 (1997), pp. 51–73.

[17] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209–233.

[18] L. J. Guibas and J. Hershberger, *Optimal shortest path queries in a simple polygon*, J. Comput. Systems Sci., 39 (1989), pp. 126–152.

[19] J. Hershberger, *A new data structure for shortest path queries in a simple polygon*, Inform. Process. Lett., 38 (1991), pp. 231–235.

[20] J. Hershberger and S. Suri, *An optimal algorithm for Euclidean shortest paths in the plane*, SIAM J. Comput., 28 (1999), pp. 2215–2256.

[21] M. Iwai, H. Suzuki, and T. Nishizeki, *Shortest path algorithm in the plane with rectilinear polygonal obstacles*, in Proceedings of SIGAL Workshop, Ryukoku University, Japan, 1994 (in Japanese).

[22] R. C. Larson and V. O. K. Li, *Finding minimum rectilinear distance paths in the presence of barriers*, Networks, 11 (1981), pp. 285–304.

[23] D. T. Lee and F. P. Preparata, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.

[24] D. T. Lee, C. D. Yang, and T. H. Chen, *Shortest rectilinear paths among weighted obstacles*, Internat. J. Comput. Geom. Appl., 1 (1991), pp. 109–124.

[25] J. S. B. Mitchell, *An Optimal Algorithm for Shortest Rectilinear Paths Among Obstacles*, presented at the First Canadian Conference on Computational Geometry, Montreal, Canada, 1989.

[26] J. S. B. Mitchell, *L_1 shortest paths among polygonal obstacles in the plane*, Algorithmica, 8 (1992), pp. 55–88.

[27] J. S. B. Mitchell, *Shortest paths among obstacles in the plane*, in Proceedings of the Ninth Annual ACM Symposium on Computational Geometry, San Diego, CA, 1993, pp. 308–317.

[28] J. S. B. Mitchell and C. H. Papadimitriou, *The weighted region problem: Finding shortest paths through a weighted planar subdivision*, J. Assoc. Comput. Mach., 38 (1991), pp. 18–73.

[29] J. A. Storer and J. H. Reif, *Shortest paths in the plane with polygonal obstacles*, J. Assoc. Comput. Mach., 41 (1994), pp. 982–1012.

[30] P. Widmayer, *On graphs preserving rectilinear shortest paths in the presence of obstacles*, in Topological Network Design, P. L. Hammer, ed., Ann. Oper. Res., 33 (1991), pp. 557–575.

[31] Y.-F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong, *Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles*, IEEE Trans. Comput.,

C-36 (1987), pp. 321–331.

[32] C. D. Yang, T. H. Chen, and D. T. Lee, *Shortest rectilinear paths among weighted rectangles*, J. Inform. Proc., 13 (1990), pp. 456–462.

[33] C.-D. Yang, D. T. Lee, and C. K. Wong, *Rectilinear path problems among rectilinear obstacles revisited*, SIAM J. Comput., 24 (1995), pp. 457–472.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1247–1256

REDUCING RANDOMNESS VIA IRRATIONAL NUMBERS*

ZHI-ZHONG CHEN† AND MING-YANG KAO‡

Abstract. We propose a general methodology for testing whether a given polynomial with integer coefficients is identically zero. The methodology evaluates the polynomial at efficiently computable approximations of suitable irrational points. In contrast to the classical technique of DeMillo, Lipton, Schwartz, and Zippel, this methodology can decrease the error probability by increasing the precision of the approximations instead of using more random bits. Consequently, randomized algorithms that use the classical technique can generally be improved using the new methodology. To demonstrate the methodology, we discuss two nontrivial applications. The first is to decide whether a graph has a perfect matching in parallel. Our new NC algorithm uses fewer random bits while doing less work than the previously best NC algorithm by Chari, Rohatgi, and Srinivasan. The second application is to test the equality of two multisets of integers. Our new algorithm improves upon the previously best algorithms by Blum and Kannan and can speed up their checking algorithm for sorting programs on a large range of inputs.

Key words. polynomial identification, Galois theory, randomized algorithms, parallel algorithms, program checking, perfect matchings, multiset equality test

AMS subject classifications. 12F10, 68Q25, 68Q22, 65Y05, 68R10, 05C70

PII. S0097539798341600

1. Introduction.

Many algorithms involve checking whether certain polynomials with integer coefficients are identically zero. Oftentimes, these polynomials have exponential-sized standard representations while having succinct nonstandard representations [6, 17, 18, 22]. This paper focuses on testing such polynomials with integer coefficients.

Given a polynomial $Q(x_1, \ldots, x_q)$ in a succinct form, a naive method of testing it is to transform it into the standard simplified form and then test whether its coefficients are all zero. Since Q may have exponentially many monomials, this method may take exponential time. Let d_Q be the degree of Q. DeMillo and Lipton [6], Schwartz [18], and Zippel [22] proposed an advanced method, which we call *the DLSZ method*. It evaluates $Q(i_1, \ldots, i_q)$, where $i_1, \ldots, i_q$ are uniformly and independently chosen at random from a set S of $2d_Q$ integers. This method uses $q\lceil\log(2d_Q)\rceil$ random bits and has an error probability of at most $\frac{1}{2}$. (Every log in this paper is to base 2.) There are three general techniques that use additional random bits to lower the error probability to $\frac{1}{t}$ for any integer $t > 2$. These techniques have their own advantages and disadvantages in terms of the running time and the number of random bits used. The first performs $\lceil\log t\rceil$ independent evaluations of Q at $\lceil\log(2d_Q)\rceil$-bit integers, using $q\lceil\log(2d_Q)\rceil\lceil\log t\rceil$ random bits. The second enlarges the cardinality of S from $2d_Q$ to td_Q and performs one evaluation of Q at $\lceil\log(td_Q)\rceil$-bit integers, using $q\lceil\log d_Q + \log t\rceil$ random bits. The third is *probability amplification* [15]. A basic such technique works for $t \le 2^{q\lceil\log(2d_Q)\rceil}$ by performing t pairwise independent evaluations of Q at

*Received by the editors July 6, 1998; accepted for publication (in revised form) June 14, 1999; published electronically February 10, 2000. A preliminary version appeared in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997, pp. 200–209.

http://www.siam.org/journals/sicomp/29-4/34160.html

†Department of Mathematical Sciences, Tokyo Denki University, Hatoyama, Saitama 350-03, Japan. This work was performed while this author was visiting the Computer Science Division, University of California, Berkeley, CA 94720.

‡Department of Computer Science, Yale University. New Haven, CT 06520. The research of this author was supported in part by NSF grant CCR-9531028.

$\lceil \log(2d_Q) \rceil$-bit integers, using $2q\lceil \log(2d_Q) \rceil$ random bits. Stronger amplification can be obtained by means of random walks on expanders [5, 1, 8].

In section 2, we propose a new general methodology for testing $Q(x_1, \ldots, x_q)$. Our methodology computes $Q(\pi_1, \ldots, \pi_q)$, where $\pi_1, \ldots, \pi_q$ are suitable irrational numbers such that $Q(\pi_1, \ldots, \pi_q) = 0$ if and only if $Q(x_1, \ldots, x_q) \equiv 0$. Since rational arithmetic is used in actual computers, we replace each π_i with a rational approximation π'_i. A crucial question is how many bits each π'_i needs to ensure that $Q(\pi'_1, \ldots, \pi'_q) = 0$ if and only if $Q(x_1, \ldots, x_q) \equiv 0$. We give an explicit answer to this question, from which we obtain a new randomized algorithm for testing Q. Our algorithm runs in polynomial time and uses $\sum_{i=1}^{q} \lceil \log(d_i + 1) \rceil$ random bits, where d_i is the degree of x_i in Q. Moreover, the error probability can be made inverse polynomially small by increasing the bit length of each π'_i. Thus, our methodology has two main advantages over previous techniques:

- It uses fewer random bits if some d_i is less than d_Q.
- It can reduce the error probability without using one additional random bit.

In general, randomized algorithms that use the classical DeMillo, Lipton, Schwartz, and Zippel (DLSZ) method can be improved using the new methodology. To demonstrate the methodology, we discuss two nontrivial applications. In section 3, the first application is to decide whether a given graph has a perfect matching. This problem has deterministic polynomial-time sequential algorithms but is not known to have a deterministic NC algorithm [7, 10, 13, 21]. We focus on solving it in parallel using as few random bits as possible. Our new NC algorithm uses fewer random bits while doing less work than the previously best NC algorithm by Chari, Rohatgi, and Srinivasan [4]. In section 4, the second application is to test the equality of two given multisets of integers. This problem was initiated by Blum and Kannan [3] for checking the correctness of sorting programs. Our new algorithm improves upon the previously best algorithms developed by them and can speed up their checking algorithm for sorting programs on a large range of inputs.

2. A new general methodology for testing polynomials.

The following notation is used throughout this paper.

- Let $Q(x_1, \ldots, x_q)$ be a polynomial with integer coefficients; we wish to test whether $Q(x_1, \ldots, x_q) \equiv 0$.
- For each x_i, let d_i be an upper bound on the degree of x_i in Q. Let $k_i = \lceil \log(d_i + 1) \rceil$.
- Let $k = \max_{i=1}^{q} k_i$ and $K = \sum_{i=1}^{q} k_i$; K is the number of random bits used by the methodology as shown in Theorem 2.3.
- Let d be an integer upper bound on the degree of Q; without loss of generality, we assume $d \geq \max_{i=1}^{q} d_i$.
- Let c be an upper bound on the absolute value of a monomial's coefficient in Q.
- Let Z be an upper bound on the number of monomials in Q; without loss of generality, we assume $Z \leq \sum_{i=0}^{d} q^i$.
- Let $\psi = \log c + \log Z + d(\log k + \frac{\log K}{2} + \log \ln K)$. Let ℓ be an integer at least $\psi + 1 + \log d$; ℓ determines the precision of our approximation to the irrational numbers chosen for the variables x_i.

For example, if all $d_i = 1$, then $k_i = 1$, $K = q$, and our goal is to use exactly q random bits, i.e., one bit per variable x_i.

Lemma 2.1. *Let $p_{1,1}, \ldots, p_{1,k_1}, \ldots, p_{q,1}, \ldots, p_{q,k_q}$ be K distinct primes. For each $p_{i,j}$, let $b_{i,j}$ be a bit. For each x_i, let $\pi_i = \sum_{j=1}^{k_i} (-1)^{b_{i,j}} \sqrt{p_{i,j}}$. Then $Q(x_1, \ldots, x_q) \not\equiv 0$*

if and only if $Q(\pi_1, \ldots, \pi_q) \neq 0$.

Proof. This lemma follows from Galois theory in algebra [14]. Let $A_0 = B_0$ be the field of rational numbers. For each x_j, let $K_j = \sum_{i=1}^{j} k_i$. Let A_j be the field generated by $\pi_1, \pi_2, \ldots, \pi_j$ over A_0. Let B_j be the field generated by $p_{1,1}, \ldots, p_{1,k_1}, \ldots, p_{j,1}, \ldots, p_{j,k_j}$ over B_0. By induction, $A_j = B_j$, the dimension of A_j over A_0 is 2^{K_j}, and the dimension of A_j over A_{j-1} is 2^{k_j}. Thus, π_j is not a root of any nonzero single variate polynomial over A_{j-1} that has a degree less than 2^{k_j}. Since $d_j < 2^{k_j}$, by induction, $Q(\pi_1, \ldots, \pi_j, x_{j+1}, \ldots, x_q) \not\equiv 0$. The lemma is proved at $j = q$. $\square$

In light of Lemma 2.1, the next algorithm tests $Q(x_1, \ldots, x_q)$ by approximating the irrational numbers $\sqrt{p_{i,j}}$ and randomizing the bits $b_{i,j}$.

ALGORITHM 1.

```
(1) Compute $q, d_1, \ldots, d_q, k_1, \ldots, k_q, K, d, c, Z$.
(2) Choose $p_{1,1}, \ldots, p_{1,k_1}, \ldots, p_{q,1}, \ldots, p_{q,k_q}$ to be the $K$ smallest primes.
(3) Choose each $b_{i,j}$ independently with equal probability for 0 and 1.
(4) Pick $\ell$, which determines the precision of our approximation to $\sqrt{p_{i,j}}$.
(5) For each $p_{i,j}$, compute a rational number $r_{i,j}$ from $\sqrt{p_{i,j}}$ by cutting off the
    bits after the $\ell$th bit after the decimal point.
(6) Compute $\Delta = Q(\sum_{j=1}^{k_1} (-1)^{b_{1,j}} r_{1,j}, \ldots, \sum_{j=1}^{k_q} (-1)^{b_{q,j}} r_{q,j})$.
(7) Output "$Q(x_1, \ldots, x_q) \not\equiv 0$" if and only if $\Delta \neq 0$.
```

The next lemma shows how to choose an appropriate ℓ at Step 1 of Algorithm 1.

LEMMA 2.2. *If* $Q(x_1, \ldots, x_q) \not\equiv 0$, *then* $|\Delta| \geq 2^{-\ell}$ *with probability at least* $1 - \frac{\psi}{\ell - 1 - \log d}$.

Proof. For each combination of the bits $b_{i,j}$, $Q(\pi_1, \ldots, \pi_q)$ is called a *conjugate.* By the prime number theorem [11], $\sqrt{p_{i,j}} \leq \sqrt{K} \ln K$ and thus $|\pi_i| \leq k\sqrt{K} \ln K$. Then, since Q has at most Z monomials, each conjugate's absolute value is at most $2^{\psi} = cZ(k\sqrt{K} \ln K)^d$. Let $\ell' = \ell - \psi - 1 - \log d$. Let α be the number of the conjugates that are less than $2^{-\ell'}$. Let $\beta = 2^K - \alpha$ be the number of the other conjugates. Let Π be the product of all the conjugates. By Lemma 2.1, $\Pi \neq 0$, and by algebra [9], Π is an integer. Thus, $|\Pi| \geq 1$ and $\alpha(-\ell') + \beta\psi \geq 0$. Hence, $\frac{\beta}{2^K} \geq \frac{\ell'}{\ell' + \psi}$; i.e., $|Q(\pi_1, \ldots, \pi_q))| \geq 2^{-\ell'}$ with the desired probability. We next show that if $|Q(\pi_1, \ldots, \pi_q)| \geq 2^{-\ell'}$, then $|\Delta| \geq 2^{-\ell}$. Since $r_{i,j} > \sqrt{p_{i,j}} - 2^{-\ell}$, $\sum_{j=1}^{k_i} r_{i,j} > |\pi_i| - k2^{-\ell}$. So approximating $p_{i,j}$ reduces each monomial term's absolute value in $Q(\pi_1, \ldots, \pi_q)$ by at most $c(k\sqrt{K} \ln K)^{d-1} dk 2^{-\ell}$. Thus, $|\Delta| \geq |Q(\pi_1, \ldots, \pi_q)| - cZ(k\sqrt{K} \ln K)^d 2^{-\ell + \log d} \geq |Q(\pi_1, \ldots, \pi_q)| - 2^{-\ell' - 1} \geq 2^{-\ell}$. $\square$

THEOREM 2.3. *For a given* $t > 1$, *set* $\ell \geq t\psi + 1 + \log d$. *If* $Q(x_1, \ldots, x_m) \equiv 0$, *Algorithm* 1 *always outputs the correct answer; otherwise, it outputs the correct answer with probability at least* $1 - \frac{1}{t}$. *Moreover, it uses exactly* K *random bits, and its error probability can be decreased by increasing* t *without using one additional random bit.*

Proof. This theorem follows from Lemma 2.2 immediately. $\square$

Let $||Q||$ be the size of the input representation of Q. The next lemma supplements Theorem 2.3 by discussing sufficient conditions for Algorithm 1 to be efficient.

LEMMA 2.4. *With* $Z = \sum_{i=1}^{d} q^i$, *Algorithm* 1 *takes polynomial time in* $||Q||$ *and* t *under the following conditions:*

- *The parameters* $q, d_1, \ldots, d_q, d$ *are at most* $(t||Q||)^{O(1)}$ *and are computable in time polynomial in* $t||Q||$.
- *The parameter* c *is at most* $2^{O(t||Q||)}$ *and is computable in time polynomial in* $t||Q||$.

- *Given ℓ'-bit numbers p'_i, $Q(p'_1, \ldots, p'_q)$ is computable in time polynomial in $t||Q||$ and ℓ'.*

Proof. The proof is straightforward based on the following key facts. There are at most $(t||Q||)^{O(1)}$ primes $p_{i,j}$, which can be efficiently found via the prime number theorem. Each $r_{i,j}$ has at most $(t||Q||)^{O(1)}$ bits and can be efficiently computed by, say, Newton's method. $\square$

We can scale up the rationals $r_{i,j}$ to integers and then compute Δ modulo a reasonably small random integer. As shown in later sections, this may considerably improve the efficiency of Algorithm 1 by means of the next fact.

FACT 1 (Thrash [19]). *Let $h \geq 3$ be an integer. If H is a subset of $\{1, 2, \ldots, h^2\}$ with $|H| \geq \frac{h^2}{2}$, then the least common multiple of the elements in H exceeds 2^h. Thus, for a given positive integer $h' \leq 2^h$, a random integer from $\{1, 2, \ldots, h^2\}$ does not divide h' with probability at least $\frac{1}{2}$.*

3. Application to perfect matching test.

Let $G = (V, E)$ be a graph with n vertices and m edges. Let $V = \{1, 2, \ldots, n\}$. Without loss of generality, we assume that n is even and $m \geq \frac{n}{2}$. A *perfect matching* of G is a set L of edges in G such that no two edges in L have a common endpoint and every vertex of G is incident to an edge in L.

Given G, we wish to decide whether it has a perfect matching. This problem is not known to have a deterministic NC algorithm. The algorithm of Chari, Rohatgi, and Srinivasan [4] uses the fewest random bits among the previous NC algorithms. This paper gives a new algorithm that uses fewer random bits while doing less work. For ease of discussion, a detailed comparison is made right after Theorem 3.2.

3.1. Classical ideas.

The *Tutte matrix* of G is the following $n \times n$ skew-symmetric matrix M of m distinct indeterminates $y_{i,j}$:

$$M_{i,j} = \begin{cases} y_{i,j} & \text{if } \{i,j\} \in E \text{ and } i < j, \\ -y_{j,i} & \text{if } \{i,j\} \in E \text{ and } i > j, \\ 0 & \text{otherwise.} \end{cases}$$

Let $L = \{\{i_1, j_1\}, \ldots, \{i_{\frac{n}{2}}, j_{\frac{n}{2}}\}\}$ be a perfect matching of G, where $i_1 < j_1, i_2 < j_2, \ldots, i_{\frac{n}{2}} < j_{\frac{n}{2}}$ and $i_1 < i_2 < \cdots < i_{\frac{n}{2}}$. Let $\pi(L) = y_{i_1,j_1} y_{i_2,j_2} \cdots y_{i_{\frac{n}{2}}, j_{\frac{n}{2}}}$. Let $\sigma(L) = 1$ or -1 if the following permutation is even or odd, respectively:

$$\begin{pmatrix} 1 & 2 & \cdots & n-1 & n \\ i_1 & j_1 & \cdots & i_{\frac{n}{2}} & j_{\frac{n}{2}} \end{pmatrix}.$$

Let $\mathrm{Pf}(G) = \sum_L \pi(L)\sigma(L)$, where L ranges over all perfect matchings in G.

FACT 2 (Fisher and Kasteleyn [2], Tutte [20]).

- $\det M = (\mathrm{Pf}(G))^2$.
- *G has a perfect matching if and only if* $\det M \not\equiv 0$.

Combining Fact 2 and the DLSZ method, Lovasz [12] gave a randomized NC algorithm for the matching problem. Since the degree of $\det M$ is at most n, this algorithm assigns to each $x_{i,j}$ a random integer from $\{1, 2, \ldots, 2n\}$ uniformly and independently and outputs "G has a perfect matching" if and only if $\det M$ is nonzero at the chosen integers. Its error probability is at most $\frac{1}{2}$, using $m\lceil\log(2n)\rceil$ random bits. The time and processor complexities are dominated by those of computing the determinant of an $n \times n$ matrix with $O(\log n)$-bit integer entries.

3.2. A new randomized NC algorithm. A direct application of Theorem 2.3 to $\det M$ uses $O(m)$ random bits, but our goal is $O(n + \log m/n)$ bits. Therefore, we need to reduce the number of variables in $\det M$.

- Let G' be the acyclic digraph obtained from G by orienting each edge $\{i, j\}$ into the arc $(\min\{i, j\}, \max\{i, j\})$.
- For each vertex i in G', let n_i be the number of outgoing arcs from i.
- Let $\hat{n}_i = 0$ if $n_i = 0$; otherwise, $\hat{n}_i = \lceil \log n_i \rceil$.
- Let $q = \sum_{i=1}^{n} \hat{n}_i$. Note that $q < n + n \log \frac{m}{n}$.
- Let $x_1, x_2, \ldots, x_q$ be q distinct new indeterminates.

We label the outgoing arcs of each vertex as follows. If $n_1 = 0$, vertex 1 has no outgoing arc in G'. If $n_1 = 1$, label its unique outgoing arc with 1. If $n_1 \geq 2$, label its n_1 outgoing arcs each with a distinct monomial in $\{(x_1)^{a_1}(x_2)^{a_2} \cdots (x_{\hat{n}_1})^{a_{\hat{n}_1}} \mid \text{each } a_h \text{ is } 0 \text{ or } 1\}$, which is always possible since $2^{\hat{n}_1} \geq n_1$. We label the n_2 outgoing arcs of vertex 2 in the same manner using $x_{\hat{n}_1+1}, x_{\hat{n}_1+2}, \ldots, x_{\hat{n}_1+\hat{n}_2}$. We similarly process the other vertices i, each using the next $\hat{n}_i$ available indeterminates x_h.

Let $f_{i,j}$ be the label of arc (i, j) in G'. Let $Q(x_1, \ldots, x_q)$ be the polynomial obtained from $\mathrm{Pf}(G)$ by replacing each indeterminate $y_{i,j}$ with $f_{i,j}$.

LEMMA 3.1. *G has a perfect matching if and only if $Q(x_1, \ldots, x_q) \not\equiv 0$.*

Proof. For each L as described in section 3.1, let $Q_L = \sigma(L) f_{i_1,j_1} f_{i_2,j_2} \cdots f_{i_{\frac{n}{2}},j_{\frac{n}{2}}}$. Then $Q = \sum_L Q_L$, where L ranges over all the perfect matchings of G. It suffices to prove that for distinct perfect matchings L_1 and L_2, the monomials Q_{L_1} and Q_{L_2} differ by at least one x_h. Let H be the subgraph of G induced by $(L_1 \cup L_2) - (L_1 \cap L_2)$. H is a set of vertex-disjoint cycles. Since $L_1 \neq L_2$, H contains at least one cycle C. Let C' be the acyclic digraph obtained from C by replacing each edge $\{i, j\}$ with the arc $(\min\{i, j\}, \max\{i, j\})$. C' contains two outgoing arcs (i, j_1) and (i, j_2) of some vertex i. So there is an indeterminate x_h used in arc labels for vertex i, whose degree is 1 in one of f_{i,j_1} and f_{i,j_2} but is 0 in the other. Hence, the degree of x_h is 1 in one of Q_{L_1} and Q_{L_2} but is 0 in the other, which makes Q_{L_1} and Q_{L_2} distinct as desired. ∎

To test whether G has a perfect matching, we use Algorithm 1 to test Q by means of Theorem 2.3 and Lemma 3.1. Below we detail each step of Algorithm 1.

Step 1. Compute q. Then set $d_1 = d_2 = \cdots = d_q = 1$, $k_1 = k_2 = \cdots = k_q = 1$, $K = q$, $d = q$, $c = 1$. Further set $Z = (\frac{2m}{n})^n$ since the number of perfect matchings in G is at most $\Pi_{i=1}^{n} m_i \leq (\frac{2m}{n})^n$, where m_i is the degree of node i in G.

Step 2. This step computes the q smallest primes $p_{1,1}, p_{2,1}, \ldots, p_{q,1}$, each at most $q \ln^2 q$. Since a positive integer p is prime if and only if it is indivisible by any integer i with $2 \leq i \leq \sqrt{p}$, these primes can be found in $O(\log q)$ parallel arithmetic steps on integers of at most $\lceil \log(1 + q \ln^2 q) \rceil$ bits using $O(q^{1.5} \log^3 q)$ processors.

Step 3. This step is straightforward.

Step 4. Set $\ell = \lceil t\psi \rceil + \lceil q \rceil + 1$, where $\psi = n \log \frac{2m}{n} + q \log(\sqrt{q} \ln q)$.

Step 5. We use Newton's method to compute $r_{i,1}$ from $p_{i,1}$. For the convenience of the reader, we briefly sketch the method here. We use $g_0 = p_{i,1}$ as the initial estimate. After the jth estimate g_j is obtained, we compute $g_{j+1} = \frac{1}{2}(g_j + \frac{p_{i,1}}{g_j})$, maintaining only the bits of g_{j+1} before the $(\ell + 1)$th bit after the decimal point. Thus, $g_{j+1} \leq \frac{1}{2}(g_j + \frac{p_{i,1}}{g_j})$. With g_{j+1} obtained, we check whether $g_{j+1}^2 > p_{i,1}$. If not, we stop; otherwise, we proceed to compute g_{j+2}. Since the convergence order of the method is 2, we take the $\lceil \log(\lceil \log p_{i,1} \rceil + \ell) \rceil$th estimate as $r_{i,1}$. Thus $r_{1,1}, \ldots, r_{q,1}$ can be computed in $O(\log(\ell + \log q))$ parallel arithmetic steps with q processors. Note that each g_j has at most $\lceil \log(1 + q \ln^2 q) \rceil + \ell$ bits.

Step 6. Evaluating Δ is equivalent to computing Δ^2. Δ^2 is the determinant of an $n \times n$ skew-symmetric matrix M' whose nonzero entries above the main diagonal in the ith row are either 1 or products of at most $\hat{n}_i$ rationals among $r_{1,1}, \dots, r_{q,1}$. Thus, each matrix entry has at most $\lceil \log n \rceil (\lceil \log(1 + q \ln^2 q) \rceil + \ell)$ bits. Setting up M' takes $O(\log n)$ arithemetic steps on $O(n^2)$ processors.

Step 7. This step is straightforward.

The next theorem summarizes the above discussion.

THEOREM 3.2. *For any given $t > 1$, whether G has a perfect matching can be determined in $O(\log(nt))$ parallel arithmetic steps on rationals of $O(tn \log^3 n)$ bits using $O(n^2)$ processors together with one evaluation of the determinant of an $n \times n$ matrix of $O(tn \log^3 n)$-bit rational entries. The error probability is at most $\frac{1}{t}$, using $q < n + n \log \frac{m}{n}$ random bits.*

Remark. The best known NC algorithm for computing the determinant of an $n \times n$ matrix takes $O(\log^2 n)$ parallel arithmetic steps using $O(n^{2.376})$ processors [16].

Proof. We separate the total complexity of Algorithm 1 into that for computing $\det M'$ and that for all the other computation. For the latter, the running time is dominated by that of Step 5; the bit length by that of the entries in M' at Step 6; and the processor count by that of setting up M'. $\square$

The work of Chari, Rohatgi, and Srinivasan [4] aims to use few random bits when the number of perfect matchings is small. Indeed, their algorithm uses the fewest random bits among the previous NC algorithms. For an error probability at most $\frac{3}{4}$, it uses $\min\{28 \sum_{i=1}^{n} \lceil \log \hat{d}_i \rceil, 6m + 4 \sum_{i=1}^{n} \lceil \log \hat{d}_i \rceil\} + O(\log n)$ random bits, where $\hat{d}_i$ is the degree of vertex i in G. It also computes the determinant of an $n \times n$ matrix with $O(n^7)$-bit entries. In contrast, with $t = 2$ in Theorem 3.2, Algorithm 1 has an error probability at most $\frac{1}{2}$ while using fewer random bits, i.e., $q < n + n \log \frac{m}{n}$ bits. Moreover, using the best known NC algorithm for determinants, the work of Algorithm 1 is dominated by that of computing the determinant of an $n \times n$ matrix with entries of shorter length, i.e., $O(n \log^3 n)$ bits.

The next theorem modifies the above implementation of Algorithm 1 by means of Fact 1 so that it computes the determinants of matrices with only $O(\log(nt))$-bit integer entries but uses slightly more random bits.

THEOREM 3.3. *For any given $t > 2$, whether G has a perfect matching can be determined in $O(\log(nt))$ parallel arithmetic steps on rationals of $O(tn \log^3 n)$ bits using $O(n^2)$ processors together with $\lceil \log t \rceil$ evaluations of the determinant of an $n \times n$ matrix of $O(\log(nt))$-bit integer entries. The error probability is at most $\frac{2}{t}$, using $q + O(\log t \log(nt))$ random bits, which is at most $n + n \log \frac{m}{n} + O(\log t \log(nt))$.*

Proof. We modify Steps 6 and 7 of the above implementation as follows.

Step 6.

- Compute M' as above.
- For each (i, j)th entry of M', we multiply it with $2^{(\hat{n}_i + \hat{n}_j)\ell}$ in $O(1)$ parallel arithmetic steps using $O(n^2)$ processors. Let M'' be the resulting matrix; note that $\det M'' = 2^{2q\ell} \det M'$ and each entry of M'' is an integer of at most $3\lceil \log n \rceil (\ell + \lceil \log n \rceil)$ bits.
- Let $\lambda = \lceil \log t \rceil$. Let $u = n! \cdot 2^{3n \lceil \log n \rceil (\ell + \lceil \log n \rceil)}$; note that $|\det M''| \le u$. We uniformly and independently choose λ random positive integers $w \le \lceil \log u \rceil^2$ using $O(\lambda \log(nt))$ random bits in $O(\lambda)$ steps on a single processor. For each chosen w, we first compute $M''' = M'' \bmod w$ in $O(1)$ parallel arithmetic steps using $O(n^2)$ processors; and then compute $\det M'''$ instead of $\det M'$.

Step 7. Output "G has a perfect matching" if and only if some $\det M'''$ is nonzero.

By Fact 1, if $\det M'' \neq 0$, then some chosen w does not divide $\det M''$ with probability at least $1-2^{-\lambda}$. Thus, the overall error probability is at most $\frac{1}{t}+2^{-\lambda} \leq \frac{2}{t}$. We separate the total complexity of Algorithm 1 into that for computing $\det M'''$ and that for all the other computation. As with Theorem 3.2, the running time of the latter remains dominated by that of Step 5, the bit length by that of the entries in M' at Step 6, and the processor count by that of setting up M'. $\square$

4. Application to multiset equality test. Let $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_n\}$ be two multisets of positive integers. Let a be the largest possible value for any element of $A \cup B$. Given A, B, and a as input, the *multiset equality test problem* is that of deciding whether $A \equiv B$, i.e., whether they contain the same number of copies for each element in $A \cup B$. This problem was initiated by Blum and Kannan [3] to study how to check the correctness of sorting programs. They gave two randomized algorithms on a useful model of computation which reflects many sorting scenarios better than the usual RAM model. For brevity, we denote their model by MBK and the two algorithms by ABK_1 and ABK_2.

This section modifies the MBK model to cover a broader range of sorting applications. It then gives a new randomized algorithm, which improves upon ABK_1 and ABK_2 and can speed up the checking algorithm for sorting by Blum and Kannan [3] on a large range of inputs.

4.1. Models of computation and previous results. In both the MBK model and the modified model, the computer has $O(1)$ tapes as well as a random access memory of $O(\log n + \log a)$ words. The allowed elementary operations are $+$, $-$, $\times$, $/$, $<$, $=$, and two bit operations shift-to-left and shift-to-right, where $/$ is integer division. Each of these operations takes one step on integers that are one word long; thus the division of an integer of m_1 words by another of m_2 words takes $O(m_1 m_2)$ time. In addition, it takes one step to copy a word on tape to a word in the random access memory or vice versa.

The only difference between the two models is that the modified model has a shorter word length relative to a and therefore is applicable to sorting applications with a larger range of keys. To be precise, in the MBK model, each word has $1+\lfloor \log a \rfloor$ bits, and thus can hold a nonnegative integer at most a. In the modified model, each word has $\xi = 1 + \lfloor \log \max\{\lceil \log n \rceil, \lceil \log a \rceil\} \rfloor$ bits and thus can hold a nonnegative integer at most $\max\{\lceil \log n \rceil, \lceil \log a \rceil\}$.

Note that sorting A and B by comparison takes $O(n \log n)$ time in the MBK model and $O(\frac{\log a}{\xi} n \log n)$ time in the modified model. However, in both models, if $n \geq 2^a$, the equality of A and B can be tested in optimal $O(n)$ time with bucket sort. Hence, we hereafter assume $n < 2^a$. We briefly review ABK_1 and ABK_2 as follows.

Let $Q_1(x)$ be the polynomial $\sum_{i=1}^n x^{a_i} - \sum_{i=1}^n x^{b_i}$. ABK_1 selects a random prime $w \leq 3a\lceil \log(n+1) \rceil$ uniformly and computes $Q_1(n+1) \bmod w$ in a straightforward manner. It outputs "$A \equiv B$" if and only if $Q_1(n+1) \bmod w$ is zero. Excluding the cost of computing w, ABK_1 takes $O(n \log a)$ time in the MBK model and $O\big((\frac{\log a}{\xi})^2 n \log a\big)$ time in the modified model. The error probability is at most $\frac{1}{2}$.

Let $Q_2(x)$ be the polynomial $\Pi_{i=1}^n (x - a_i) - \Pi_{i=1}^n (x - b_i)$. ABK_2 uniformly selects a random positive integer $z \leq 4n$ and a random prime $w \leq 3n\lceil \log(a + 4n) \rceil$; and computes $P(z) \bmod w$ in a straightforward manner. It outputs "$A \equiv B$" if and only if $P(z) \bmod w$ is zero. Excluding the cost of computing w, ABK_2 takes $O\big(n \max\{1, (\frac{\log n}{\log a})^2\}\big)$ time in the MBK model and $O\big(n \frac{(\log n + \log a)(\log n + \log\log a)}{\xi^2}\big)$ time in the modified model. The error probability is at most $\frac{3}{4}$.

Generating the random primes w is a crucial step of ABK_1 and ABK_2. It is unclear how this step can be performed efficiently in terms of running time and random bits. We modify this step by means of Fact 1 as follows. In ABK_1, $|Q_1(n+1)| \leq 2^{1+a\log(n+1)+\log n}$; in ABK_2, $|Q_2(2n)| \leq 2^{1+n\log(a+4n)}$. Thus, we can replace w in ABK_1 and ABK_2 with two random positive integers $w_1 \leq (1+a\log(n+1)+\log n)^2$ and $w_2 \leq (1+n\log(a+4n))^2$, respectively. With these modifications, ABK_1 and ABK_2 use at most $2\log a + 2\log\log n + O(1)$ and $3\log n + 2\log\log(a+n) + O(1)$ random bits, respectively. The time complexities and error probabilities remain as stated above.

4.2. A new randomized algorithm. Our goal in this section is to design an algorithm for multiset equality test for the modified model that is faster than ABK_1 for $n = \omega((\log\log a)^2)$ and faster than ABK_2 for $n = \omega\left((\log a)^{\log\log a}\right)$. We can then use it to speed up the previously best checking algorithm for sorting [3].

- Let $q = \lfloor \log a \rfloor + 1$.
- Let $x_1, \ldots, x_q$ be q distinct indeterminates.
- For each $u \in A \cup B$, let f_u denote the monomial $(x_1)^{u_1}(x_2)^{u_2}\cdots(x_q)^{u_q}$, where $u_1u_2\cdots u_q$ is the standard q-bit binary representation of u.
- Let $Q(x_1,\ldots,x_q)$ denote the polynomial $\sum_{i=1}^n f_{a_i} - \sum_{i=1}^n f_{b_i}$.

Note that $Q(x_1,\ldots,x_q) \equiv 0$ if and only if $A \equiv B$. To test whether $A \equiv B$, we detail how to implement the steps of Algorithm 1 to test Q as follows. The algorithm is analyzed only with respect to the modified model.

Remark. In the implementation, the parameter t of Theorem 2.3 needs to be a constant so that the algorithm can be performed inside the random access memory together with straightforward management of the tapes. At the end of this section, we set $t = 4$, but for the benefit of future research, we analyze the running time and the random bit count in terms of a general t.

Step 1. Compute q by finding the index of the most significant bit in the binary representation of a. Since a takes up $O(\frac{\log a}{\xi})$ words, this computation takes $O(q)$ time by shifting the most significant nonzero word to the left at most ξ times. Afterwards, set $d_1 = d_2 = \cdots = d_q = k_1 = k_2 = \cdots = k_q = k = 1$, $K = d = q$, $c = n$, and $Z = 2n$ in $O(q)$ time. This step takes $O(q)$ time.

Step 2. Compute the q smallest primes $p_{1,1}, p_{2,1}, \ldots, p_{q,1} \leq q\ln^2 q$. We compute these primes by inspecting $i = 2, 3, \ldots$ one at a time up to $q\ln^2 q$ until exactly q primes are found. Since i can fit into $O(1)$ words, it takes $O(\sqrt{q}\log q)$ time to check the primality of each i using the square root test for primes in a straightforward manner. Thus, this step takes $O(q^{3/2}\log^3 q)$ time.

Step 3. This step is straightforward and uses q random bits and $O(\frac{q}{\xi})$ time.

Step 4. Set $\ell = \lceil t \rceil \psi' + \lceil q \rceil + 1$, where t is a given positive number and $\psi' = 2\lceil \log n \rceil + \lceil \frac{q\lceil \log q \rceil}{2} \rceil + q\lceil \log\lceil \log q \rceil \rceil + 1$. The number $\lceil \log n \rceil$ can be computed from the input in $O(n)$ time. The computations of $\lceil \frac{\log q}{2} \rceil$ and $\lceil \log\lceil \log q \rceil \rceil$ are similar to Step 1 and take $O(\log q)$ time. Thus, this step takes $O(n + \log q + \frac{\log t}{\xi})$ time.

Step 5. As at Step 5 in section 3.2, we use Newton's method to compute $r_{i,1}$ for each $p_{i,1}$. With only integer operations allowed, we use $2^\ell g_j$ as the jth estimate for $2^\ell\sqrt{p_{i,1}}$; i.e., $2^\ell g_{j+1} = (2^\ell g_j + 2^{2\ell}p_{i,1}/(2^\ell g_j))/2$. The last estimate computed in this manner is $2^\ell r_{i,1}$. Since 2^ℓ can be computed in $O((\frac{\ell}{\xi})^2)$ time using a doubling process, the first estimate $2^\ell p_{i,1}$ can be computed in the same amount of time. Since the other estimates all are $O(\frac{\ell}{\xi})$ words long, the $(j+1)$th estimate can be obtained from the jth in $O((\frac{\ell}{\xi})^2)$ time. Since only $O(\log \ell)$ iterations for each $2^\ell\sqrt{p_{i,1}}$ are needed, this step takes $O(q(\frac{\ell}{\xi})^2 \log \ell)$ time.

Step 6. We compute $\Delta = Q((-1)^{b_{1,1}} r_{1,1}, \ldots, (-1)^{b_{q,1}} r_{q,1})$ by means of Fact 1 as follows. Let $\lambda = \lceil \log t \rceil$. Since $|2^{q\ell}\Delta|$ is an integer at most $2^{\psi' + q\ell}$, we uniformly and independently select λ random positive integers $w \le (\psi' + q\ell)^2$ using $2\lambda(\log t + \log\log n + 2\log\log a + o(\log\log a))$ random bits and $O(\lambda \frac{\log \ell}{\xi})$ time. Note that if $2^{q\ell}\Delta \neq 0$, then with probability at least $1 - \frac{1}{t}$, some $2^{q\ell}\Delta \bmod w$ is nonzero. We next compute all $2^{ql}\Delta \bmod w$. For each element $u \in A \cup B$, let $e(u)$ be the number of 0's in the standard q-bit binary representation of u. Let $h(u) = f_u((-1)^{b_{1,1}} 2^\ell r_{1,1}, \ldots, (-1)^{b_{q,1}} 2^\ell r_{q,1})$. Then, $2^{q\ell}\Delta = \sum_{i=1}^n 2^{e(a_i)\ell} h(a_i) - \sum_{i=1}^n 2^{e(b_i)\ell} h(b_i)$, which we use to compute all $2^{q\ell}\Delta \bmod w$ as follows.

- Compute the numbers $e(u)$ for all $u \in A \cup B$ in $O(nq)$ time.
- For all w, compute all $2^\ell r_{i,1} \bmod w$ in $O(\lambda q \frac{\ell}{\xi} \frac{\log \ell}{\xi})$ time.
- For all w, use values obtained above to compute $h(u) \bmod w$ for all u in $O(\lambda nq(\frac{\log \ell}{\xi})^2)$ time.
- For all w, compute $2^\ell \bmod w$ in $O(\lambda \frac{\ell}{\xi} \frac{\log \ell}{\xi})$ time.
- For all w, use values obtained above to compute $2^{e(u)\ell} \bmod w$ for all u in $O(\lambda n(\frac{\log \ell}{\xi})^2 \log q)$ time.
- For all w, use values obtained above to compute $2^{q\ell}\Delta \bmod w$ in $O(\lambda n(\frac{\log \ell}{\xi})^2)$ time.

This step uses $2\lambda(\log t + \log\log n + 2\log\log a + o(\log\log a))$ random bits and takes $O(\lambda q \frac{\ell}{\xi} \frac{\log \ell}{\xi} + \lambda nq(\frac{\log \ell}{\xi})^2)$ time.

Step 7. Output "$A \not\equiv B$" if and only if some $2^{ql}\Delta \bmod w$ is nonzero.

The next theorem summarizes the above discussion.

THEOREM 4.1. *For any given $t > 2$, whether $A \equiv B$ can be determined in time*

$$O\left(q \log \ell \left(\frac{\ell}{\xi}\right)^2 + \lambda nq \left(\frac{\log \ell}{\xi}\right)^2\right),$$

where $q = \Theta(\log a)$; $\ell = \Theta(t(\log n + \log a \log\log a))$; $\xi = \Theta(\log\log(n+a))$; $\lambda = \Theta(\log t)$. The error probability is at most $\frac{2}{t}$ using $\log a + 2\lceil \log t \rceil(\log t + \log\log n + 2\log\log a + o(\log\log a))$ random bits.

Proof. The running time of Algorithm 1 is dominated by those of Steps 5 and 6. The error probability follows from Theorem 2.3 and Fact 1. $\square$

We use the next corollary of Theorem 4.1 to compare Algorithm 1 with ABK$_1$ and ABK$_2$ in the modified model.

COROLLARY 4.2. *With $t = 4$, Algorithm 1 has an error probability at most $\frac{1}{2}$ using $\log a + 4\log\log n + 8\log\log a + o(\log\log a)$ random bits, while running in time*

$$O\left(n \log a + \log a \frac{(\log n + \log a \log\log a)^2}{\log\log(n+a)}\right).$$

By Corollary 4.2, Algorithm 1 is faster than ABK$_1$ for $n = \omega((\log\log a)^2)$ and faster than ABK$_2$ for $n = \omega\left((\log a)^{\log\log a}\right)$. Thus, it can replace ABK$_1$ and ABK$_2$ to speed up the previously best checking algorithm for sorting [3] as follows. We use bucket sort for $2^a \le n$, Algorithm 1 for $(\log a)^{\log\log a} \le n < 2^a$, and ABK$_2$ otherwise.

Acknowledgments. We are very grateful to Steve Tate for useful discussions and to the anonymous referees for extremely thorough and helpful comments.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *Deterministic simulation in Logspace*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 132–140.

[2] C. Berge, *Graphs*, second revised ed., North-Holland, Amsterdam, 1985.

[3] M. Blum and S. Kannan, *Designing programs that check their work*, J. ACM, 42 (1995), pp. 269–291.

[4] S. Chari, P. Rohatgi, and A. Srinivasan, *Randomness-optimal unique element isolation with applications to perfect matching and related problems*, SIAM J. Comput., 24 (1995), pp. 1036–1050.

[5] A. Cohen and A. Wigderson, *Dispersers, deterministic amplification, and weak random sources (extended abstract)*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 14–19.

[6] R. A. DeMillo and R. J. Lipton, *A probabilistic remark on algebraic program testing*, Inform. Process. Lett., 7 (1978), pp. 193–195.

[7] Z. Galil, S. Micali, and H. Gabow, *An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs*, SIAM J. Comput., 15 (1986), pp. 120–130.

[8] R. Impagliazzo and D. Zuckerman, *How to recycle random bits*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 248–253.

[9] N. Jacobson, *Basic Algebra*, W. H. Freeman, San Francisco, CA, 1974.

[10] R. M. Karp, E. Upfal, and A. Wigderson, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[11] W. J. LeVeque, *Topics in Number Theory*, Vol. 1, Addison-Wesley, Reading, MA, 1956.

[12] L. Lovasz, *On determinants, matchings and random algorithms*, in Fundamentals of Computing Theory (Proceedings of the Conference on Algebraic, Arith. and Categorical Methods in Comput. Theory, Berlin, Wendisch-Rietz, 1979), Math. Res. 2, L. Budach, ed., Akademia-Verlag, Berlin, 1979, pp. 565–574.

[13] S. Micali and V. V. Vazirani, *An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs*, in Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1980, pp. 17–27.

[14] P. Morandi, *Field and Galois Theory*, Grad. Texts in Math. 167, Springer-Verlag, New York, 1996.

[15] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK, 1995.

[16] V. Pan, *Complexity of parallel matrix computations*, Theoret. Comput. Sci., 54 (1987), pp. 65–85.

[17] J. H. Rowland and J. R. Cowles, *Small sample algorithms for the identification of polynomials*, J. ACM, 33 (1986), pp. 822–829.

[18] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. ACM, 27 (1980), pp. 701–717.

[19] W. Thrash, *A Note on the Least Common Multiples of Dense Sets of Integers*, Tech. Report 93-02-04, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[20] W. T. Tutte, *The factors of graphs*, Canad. J. Math., 4 (1952), pp. 314–328.

[21] V. V. Vazirani, *Maximum Matchings without Blossoms*, Ph.D. thesis, University of California, Berkeley, CA, 1984.

[22] R. E. Zippel, *Probabilistic algorithms for sparse polynomials*, in Proceedings of EUROSAM '79, an International Symposium on Symbolic and Algebraic Manipulation, Lecture Notes in Comput. Sci. 72, E. W. Ng, ed., Springer-Verlag, New York, 1979, pp. 216–226.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1257–1289

ON THE POWER OF LOGIC RESYNTHESIS*

WEI-LIANG LIN[†], AMIR H. FARRAHI[‡], AND M. SARRAFZADEH[§]

Abstract. A linear arrangement problem, called the minmax mincut problem, emerging from circuit design is investigated. Its input is a series-parallel directed hypergraph (SPDH), and the output is a linear arrangement (and a layout). The primary objective is to minimize the longest path, and the secondary objective is to minimize the cutwidth. It is shown that cutwidth D, subject to longest path minimization, is affected by two terms: *pattern number* k and *balancing number* m. Also, k and m are both lower bounds on the cutwidth. An algorithm, running in linear time, produces layouts with cutwidths $D \leq 2(k+m)$. There exist examples with $k = \Omega(N)$, where N is the number of vertices; however, m is always $O(\log N)$. We show that every SPDH, after *local logic resynthesis* (specifically, after reordering the serial paths), can be linearly placed with cutwidth $D = O(\log N)$. Simultaneously, its dual SPDH can be linearly placed with the same vertex order and with cutwidth $D = O(\log N)$. Therefore, after local resynthesis the area can be reduced by a factor of $N/\log N$.

Application to gate-matrix layout style is demonstrated.

Key words. VLSI design, linear arrangement, graphs and large scale networks, resynthesis, timing, VLSI layout

AMS subject classifications. 94C15, 94C10, 8R05, 68R10, 68M07

PII. S0097539796335480

1. Introduction.

This section begins by defining basic terms. Then we formulate the concerned problem and compare it with related problems in the literature.

1.1. Definitions.

The following definitions originate from a practical circuit application, which will be further explained in section 6. Most terms are natural extensions of terms from graph theory.

An *undirected hypergraph* $G = (V, E)$ consists of two sets V and E, where V is a set of vertices and E is a set of undirected hyperedges. An *undirected hyperedge* is a nonempty subset of V.

A *directed hypergraph* $H = (V, E)$ also consists of two sets V and E, where V is a set of vertices and E is a set of directed hyperedges. A *directed hyperedge* $e = (V_1, V_2)$ is an ordered pair of nonempty subsets of V, where V_1 is the *source set* of hyperedge (V_1, V_2) and V_2 is the *sink set* of hyperedge (V_1, V_2): $V_1 \cap V_2 = \emptyset$. Figure 1.1 shows a directed hypergraph $H = (V, E)$ with five vertices $V = \{a, b, c, d, e\}$ and two hyperedges $E = \{(\{a, b\}, \{c, d\}), (\{d\}, \{e\})\}$. $\{a, b\}$ and $\{c, d\}$ are the source set and the sink set of hyperedge $(\{a, b\}, \{c, d\})$, respectively. $\{d\}$ and $\{e\}$ are the source set and the sink set of hyperedge $(\{d\}, \{e\})$, respectively.

A *source* s of a directed hypergraph is a vertex without any incident hyperedge containing s in its sink set. Similarly, a *sink* t of a directed hypergraph is a vertex without any incident hyperedge containing t in its source set. Thus in Figure 1.1, a, b are sources and c, e are sinks.

*Received by the editors January 1, 1996; accepted for publication (in revised form) April 28, 1998; published electronically February 16, 2000. This work was supported in part by the National Science Foundation under grants MIP-9207267 and MIP-9527389.

http://www.siam.org/journals/sicomp/29-4/33548.html

[†]Intel Corporation, 2111 NE 25th Ave., MS: JF1-19, Hillsboro, OR 97124 (wllin@ichips.intel.com).

[‡]IBM Corporation, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (ahf@watson.ibm.com).

[§]Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208 (majid@ece.nwu.edu).

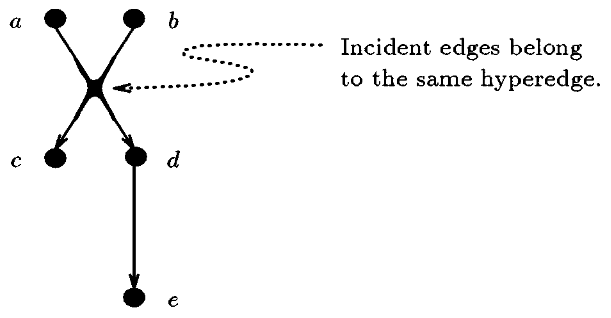


FIG. 1.1. *A directed hypergraph $H = (V, E)$ with five vertices $V = \{a, b, c, d, e\}$ and two hyperedges $E = \{(\{a, b\}, \{c, d\}), (\{d\}, \{e\})\}$. $\{a, b\}$ and $\{c, d\}$ are the only source set and the only sink set of hyperedge $(\{a, b\}, \{c, d\})$, respectively. $\{d\}$ and $\{e\}$ are the only source set and the only sink set of hyperedge $(\{d\}, \{e\})$, respectively. Also, a, b are sources and c, e are sinks.*

A *two-terminal directed hypergraph* is a directed hypergraph with only one source and only one sink. Notice that source sets and sink sets refer to hyperedges and sources and sinks refer to directed hypergraphs. A directed hypergraph is *acyclic* if there does not exist a sequence (with a cardinality larger than 1) of pairs of ordered vertices $(v_1, v_2), (v_2, v_3), \ldots, (v_m, v_{m+1} = v_1)$, where in (v_i, v_{i+1}), v_i and v_{i+1} belong to the source set and the sink set of a distinct hyperedge, respectively. Therefore, the directed hypergraph H in Figure 1.1 is acyclic but is not a two-terminal one.

We recursively define a *series-parallel directed hypergraph* (SPDH) as follows. A directed hypergraph, denoted as an *SPDH basis* (SPDHB), consisting of three vertices s, v, t connected by two hyperedges, as shown in Figure 1.2(a), is an SPDH. In addition to an SPDHB, an SPDH can be constructed by applying two operations, *series* and *parallel*, on two SPDHs, recursively. A *series* operation combines two SPDHs H_1, H_2 by removing the sink of H_1 and the source of H_2 and merging the remaining two "dangling" hyperedges into one hyperedge, as Figure 1.2(c) shows. A *parallel* operation combines two SPDHs by merging the two sources and the corresponding incident hyperedges and merging the two sinks and the corresponding hyperedges, as Figure 1.2(d) shows.

Obviously, an SPDH is acyclic. In addition, each vertex v has only two incident hyperedges: one *incoming hyperedge* containing vertex v in the sink set of the hyperedge and one *outgoing hyperedge* containing vertex v in the source set of the hyperedge. However, the source has only one incident hyperedge, called the *source hyperedge*, and the sink has also only one incident hyperedge, called the *sink hyperedge*.

A *series-parallel* (S-P) *tree* [13] T can represent an SPDH H. Since H is recursively constructed, we define T in a recursive manner. Each leaf of T corresponds to an SPDHB. Let H_1 and H_2 be two subgraphs of H that are combined either in series (labeled S) or in parallel (labeled P). The tree corresponding to the union of H_1 and H_2 consists of a node label S or P (depending on the combination step) with T_1 and T_2 as its subtrees, where T_1 and T_2 are trees corresponding to H_1 and H_2, respectively.

In order to simplify our discussion in linear arrangements of an SPDH, a modified S-P tree, called a "marked" S-P tree, is defined as follows. A double line below an S-type node is used to emphasize that the SPDH of the corresponding subtree should be located closer to the source. Figure 1.3(a) shows an SPDH constructed by applying 3 series and 3 parallel operations on 7 SPDHBs, and Figure 1.3(b) shows a corresponding

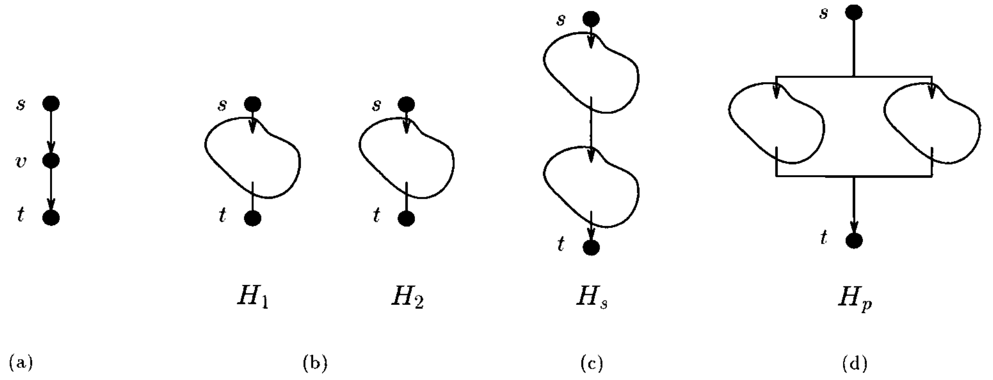


FIG. 1.2. (a) *An SPDHB.* (b) *Two SPDHs, H_1 and H_2, result in another SPDH,* (c) *H_s after a series operation or* (d) *H_p after a parallel operation.*

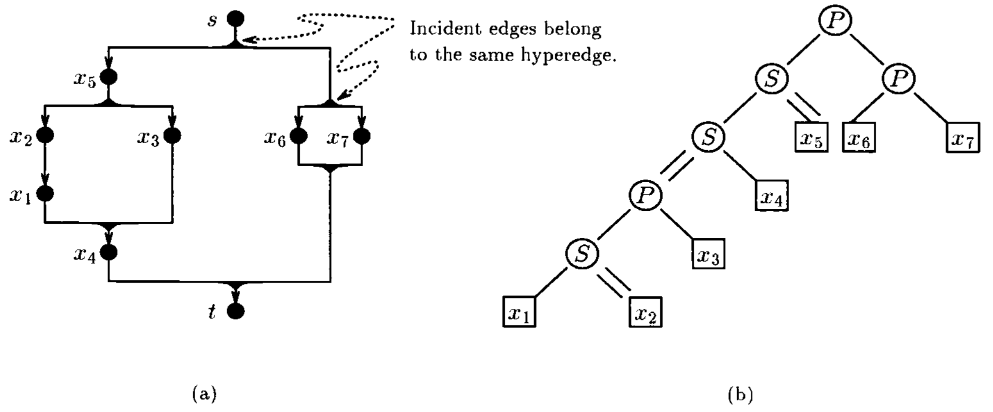


FIG. 1.3. (a) *An SPDH constructed by applying* 3 *series and* 3 *parallel operations on* 7 *SPDHBs.* (b) *A corresponding S-P tree. Notice that vertices s and t do not appear in the S-P tree because of the definition of an S-P tree.*

S-P tree.[1] The *dual* of an SPDH H is also an SPDH H_d constructed from a similar procedure as the one constructing H. However, when the procedure which constructs H employs a series operation, the dual procedure which constructs H_d employs a parallel operation, and vice versa. An S-P tree of H_d is a dual of an S-P tree of H. To clarify, we employ the name "nodes" instead of "vertices" in the context of S-P trees and employ "vertices" in the context of directed hypergraphs. Figures 1.4(a) and (b) show duals of the SPDH and S-P tree of Figure 1.3.

A *linear arrangement* (also called a *linear layout* or simply a *layout*) of a directed hypergraph H is a one-to-one function $L : V \to 1, 2, \ldots, |V|$. The cutwidth D of a layout is $D = \max(|E(i)| \mid 1 \leq i < |V|)$, where $E(i) = \{(V_1, V_2) \in E \mid \exists x \in V_1, y \in V_2, or, x \in V_2, y \in V_1$ such that $L(x) \leq i < L(y)\}$. In other words, $|E(i)|$ is the least number of tracks needed at position i in order to lay out all hyperedges crossing position i in distinct tracks. Figure 1.5 shows two layouts of the SPDH of Figure 1.3(a), where each filled circle represents a vertex: one layout needs three

[1]An SPDH may correspond to more than one S-P tree.

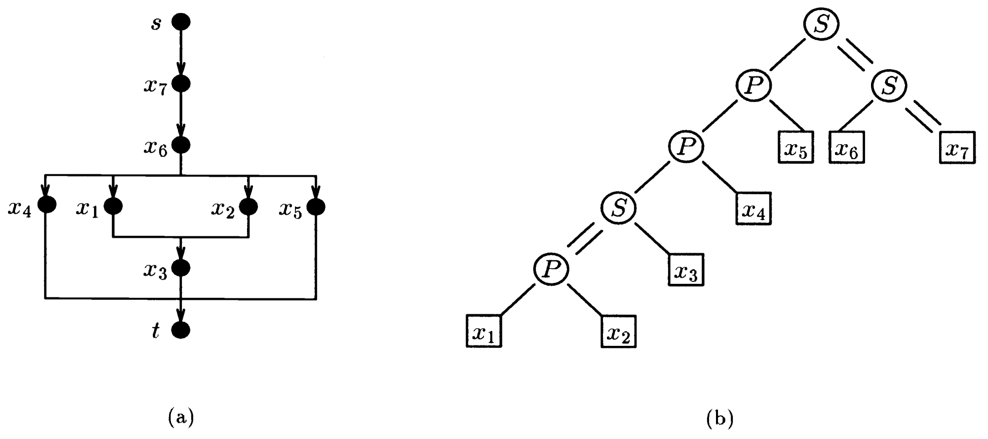


FIG. 1.4. (a) *A dual of the SPDH in Figure* 1.3(a). (b) *A dual of the S-P tree in Figure* 1.3(b).

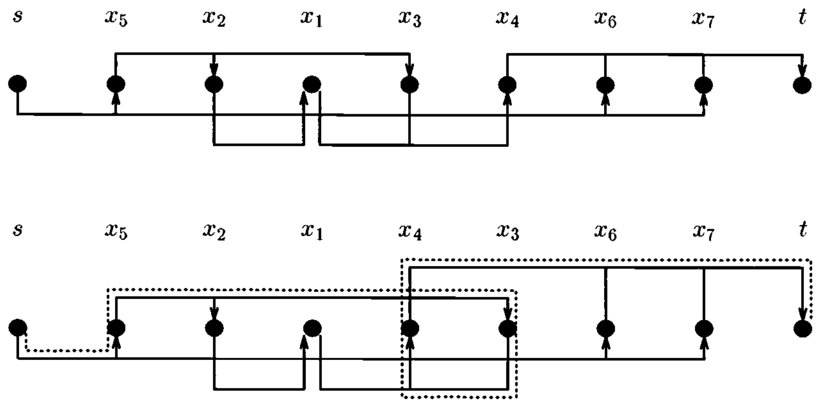


FIG. 1.5. *Two layouts for the SPDH in Figure* 1.3(a), *one with three tracks and the other with four tracks. Each filled circle represents a placed vertex, and the dotted line shows a path with a bend which will be discussed in section* 2.

tracks and the other needs four tracks.

1.2. Problem formulation. We formulate a novel linear arrangement problem as follows.

A minmax mincut problem.

Input: An SPDH.

Output: A minimum cutwidth layout with the constraints that the source is at the leftmost position, the sink is at the rightmost position, and the longest path[2] is minimized.

1.3. Related problems. In the literature, problems related to the minmax mincut problem are the single-machine sequencing with precedence relations problem [11], the mincut linear arrangement problem [4], and the gate-matrix area (height)

[2]Only horizontal lengths are taken into consideration since vertical lengths will become negligible when the number of vertices increases.

minimization problem [8]. However, there are no obvious methods to transform the minmax mincut problem to one of these problems.

The single-machine sequencing with precedence relations problem accepts an input of precedence relations, which forms a directed graph. The goal is to linearly arrange the vertices under the constraints of the precedence relations. Nevertheless, the problem of including the cutwidth in the cost function has not yet been addressed.

The mincut linear arrangement problem focuses on an undirected graph. The considered edges are not directed hyperedges. Minimizing the cutwidth of a linear layout of a general undirected graph is NP-complete [3]. If the graph is a tree, it can be solved in polynomial time [15].

In terms of our above definitions, the gate-matrix minimization problem is actually the "mincut linear arrangement of a general directed hypergraph" problem. However, it does not have the constraint of minimizing the longest path. The gate-matrix minimization problem is also NP-complete and a number of effective heuristics have been proposed [2, 5, 7, 6]. Also, a tight bound $\Theta(\log N)$ on the cutwidth was proved [12].

This paper is further organized as follows. Section 2 investigates the constraint of the longest path minimization problems. Section 3 shows that the *pattern number* k is a lower bound on the cutwidth D, and that there exist examples with $k = \Omega(N)$. An algorithm on the minmax mincut problem ensuring cutwidth $D \leq 2(k + m)$ is presented in section 4, where the *balancing number* m is also introduced. Section 5 proposes a reordering and layout algorithm, which reorders serial paths and simultaneously reduces the *pattern number*s of an SPDH and its dual from k to $O(\log N)$. The algorithm also lays out the SPDH and its dual in the same layout order. Therefore, every SPDH and its dual can be simultaneously laid out with cutwidth $O(\log N)$. Section 6 illustrates applications and shows two examples. The last section summarizes and concludes this paper.

2. Minimizing the longest path.

This section describes a class of layouts subject to two constraints: (1) the longest path is minimized, (2) the source is at the leftmost position and the sink is at the rightmost position.

THEOREM 2.1. *A layout of an SPDH, with the constraints that the source is at the leftmost position and the sink is at the rightmost position, has the minimum longest path if and only if it does not have any "bends" in its paths.*

Proof. There are two observations: (1) the source is at the leftmost position and the sink is at the rightmost position,[3] (2) since each path must begin at the source and end at the sink, the minimum path length is the straight-line distance from the source to the sink. If there is no bend in any path, each path (including the longest one) has the same minimum length, the straight-line distance from the source to the sink. Obviously, any bend will increase the lengths of some paths and, consequently, will violate the constraint of minimizing the longest path. Figure 1.5 shows layouts with and without bends. $\square$

It is not difficult to see that the set of all layouts produced by *topological sorting* [1] of directed hypergraphs is equal to the set of layouts without bends. (As will be discussed in section 6, layouts obtained from topological sorting are of particular importance in designing high-performance VLSI circuits.) A *topological sorting* of an acyclic directed hypergraph $H = (V, E)$ is a layout of its vertices such that if H contains a directed hyperedge (V_1, V_2), each vertex in V_1 will be laid out to the left of

[3]Recall that an SPDH has only one source and only one sink.

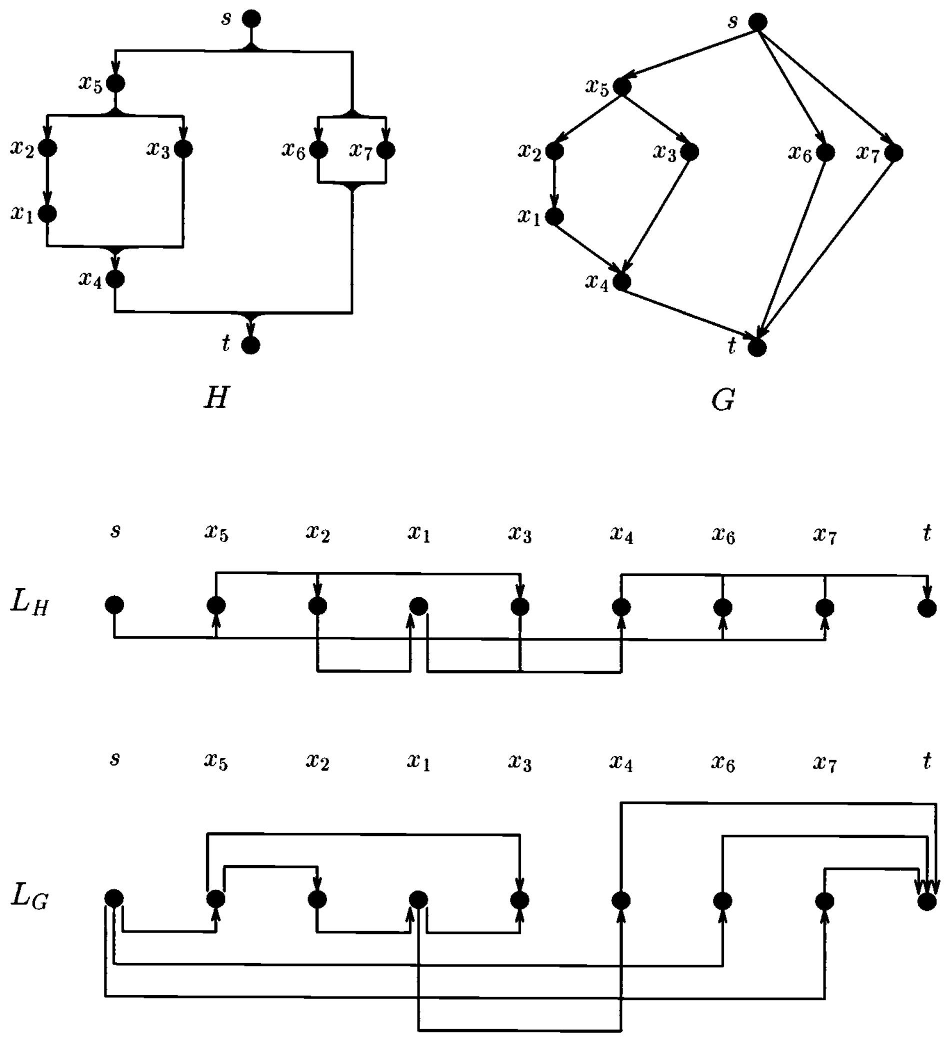


FIG. 2.1. *A directed hypergraph H is transformed to a directed graph G, and the topologically sorted layout L_H of H corresponds to the topologically sorted layout L_G of G.*

each vertex in V_2.

Notice that by substituting each directed hyperedge with two-terminal directed edges from each vertex of the source set to each vertex of the sink set, we transform any directed hypergraph H to a directed graph G. It is easy to see that there is a one-to-one relation between topological sortings of G and H. In Figure 2.1, a directed hypergraph H is transformed to a directed graph G and the topologically sorted layout L_H of H corresponds to the topologically sorted layout L_G of G.

3. Cutwidth consideration in topologically sorted layouts. Recall that for an SPDH layout the number of tracks needed is lower bounded by the cutwidth. In this section, among the *topologically sorted* (TS) *layouts* an SPDH with $D = \Omega(N)$ tracks is shown, where N is the number of vertices. In addition, the *pattern number* k is defined and shown to be a lower bound on the number of tracks D. However, this

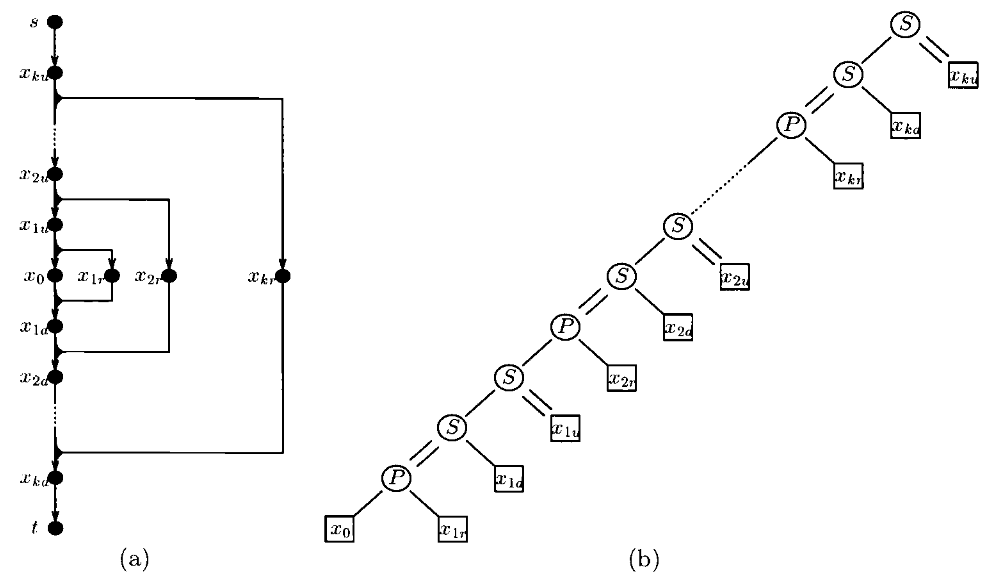


FIG. 3.1. (a) *A TS-difficult SPDH requiring $\Omega(N)$ tracks.* (b) *An S-P tree for the TS-difficult SPDH.*

bound is not tight.

Consider a class of SPDHs $TS\_d(k)$, called *TS-difficult SPDHs*, which is recursively defined as follows. The basis $TS\_d(0)$ is an SPDHB. $TS\_d(k)$ is constructed from $TS\_d(k-1)$ by adding an SPDHB in parallel and then serially merging an SPDHB to the source of the resulting SPDH and then adding another SPDHB to the sink of the resulting SPDH. The general form is shown in Figure 3.1(a) with $N = 3k + 3$.

LEMMA 3.1. *All TS-layouts of a TS-difficult SPDH $TS\_d(k)$ require $\Omega(N)$ tracks, where N is the number of vertices.*

Proof. Let $\mathrm{tr}(TS\_d(k))$ denote the minimum track number of $TS\_d(k)$ under TS. This lemma is proved by induction on k.

Basis: It is clear that $\mathrm{tr}(TS\_d(1)) = 2 \geq 1$.

Induction hypothesis: $\mathrm{tr}(TS\_d(k)) \geq k$.

Induction step: We need to show that $\mathrm{tr}(TS\_d(k+1)) \geq k+1$. Suppose TS_$d(k)$ has been laid out using the minimum number of tracks, as shown in Figure 3.2(a), where the only possible position for x_{ku} is at the leftmost and for x_{kd} is at the rightmost because of the TS constraint.

There are three possible positions for $x_{(k+1)r}$, as shown in Figure 3.2(b).

Case 1. *$x_{(k+1)r}$ is to the left of x_{ku}.*

Since $x_{(k+1)r}$ must be connected to $x_{(k+1)d}$ and this hyperedge is distinct from other hyperedges, we need a new track.

Case 2. *$x_{(k+1)r}$ is to the right of x_{kd}.*

Since $x_{(k+1)r}$ must be connected to $x_{(k+1)u}$ and this hyperedge is distinct from other hyperedges, we need a new track.

Case 3. *$x_{(k+1)r}$ is between x_{ku} and x_{kd}.*

If we can place the source hyperedge of $x_{(k+1)r}$, which is to the left of $x_{(k+1)r}$, in a track i and place the sink hyperedge of $x_{(k+1)r}$, which is to the right of $x_{(k+1)r}$ in another track j, these two tracks could be combined to reduce

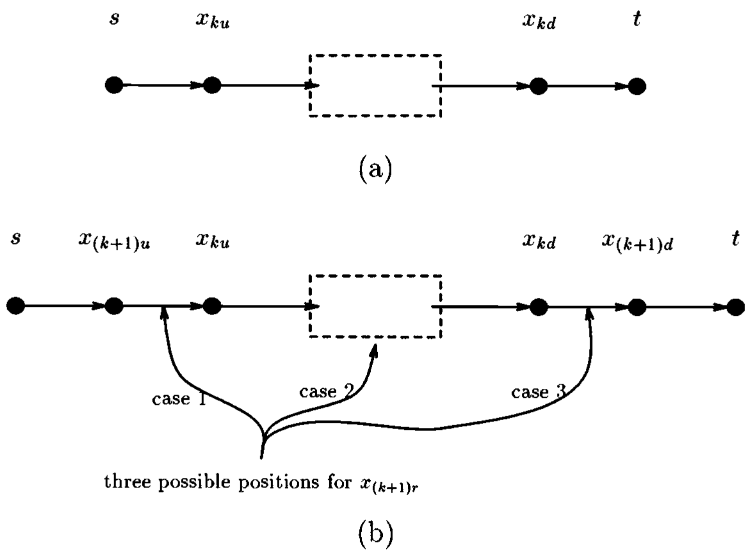


FIG. 3.2. (a) *A $TS\_d(k)$ layout with a minimal number of tracks.* (b) *There are three possible positions for the $x_{(k+1)r}$ of a $TS\_d(k+1)$.*

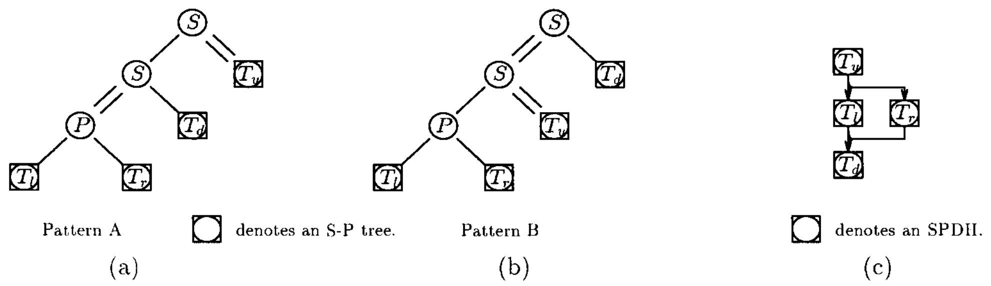


FIG. 3.3. (a) *Pattern A,* (b) *pattern B, and* (c) *the represented SPDH.*

tr($TS\_d(k)$) by 1. This is a contradiction to the assumption that $TS\_d(k)$ has been laid out using a minimal number of tracks. Therefore, we need a new track.

As discussed above, the number of tracks is bounded from below by $k = \frac{N-3}{3}$. That is, the lower bound is $\Omega(N)$. □

In Figure 3.1, a "pattern," called pattern A, depicted in Figure 3.3(a), is repeated along the longest path. Figure 3.3(b) shows an equivalent pattern, called pattern B. Patterns A and B are equivalent because the corresponding SPDHs, shown in Figure 3.3(c), are identical.

In an S-P tree, an *external path* is a path from a leaf to the root; it contains a sequence of inner nodes and a set of edges interconnecting the nodes. An external path can be decomposed into three kinds of elements as follows:

$$\left\{ \begin{matrix} P \\ | \end{matrix}, \begin{matrix} S \\ | \end{matrix}, \begin{matrix} S \\ || \end{matrix} \right\}.$$

We use $P|, S|, S||$ to represent these three elements. The *pattern number* of an external

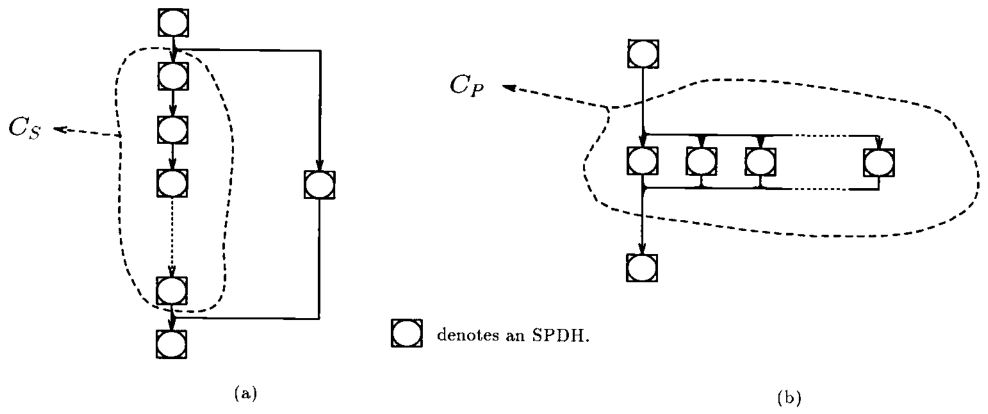


FIG. 3.4. (a) *A set of subgraphs are combined in series, called C_s.* (b) *Another set of subgraphs are combined in parallel, called C_p.*

path is defined in the following procedure:

0. Set the *pattern number* equal to 0.
1. Trace up the path from the corresponding leaf.
2. When encountering $P|$, consider it as part of a new pattern.
3. Continue to trace up until at least one $S|$ and at least one $S||$ (not necessarily in that order) are visited; call these two parts of the current pattern; add 1 to the *pattern number.*
4. Repeat steps 2 and 3 until the path ends.

Among the pattern numbers of external paths, the largest one defines the *pattern number* of an S-P tree and the corresponding SPDH. The following lemma essentially shows that the pattern number of an SPDH is well defined.

LEMMA 3.2. *All S-P trees corresponding to a given SPDH have a unique pattern number.*

Proof. When an S-P tree is constructed, either a set of subgraphs are combined in series, called C_s, or a set of subgraphs are combined in parallel, called C_p (see Figure 3.4).

For the series case, along any path from the corresponding subtree we meet a sequence of S-type nodes until another subgraph combines in parallel with C_s, and we will meet the same P-type node. If there is a construction of an S-P tree where along a specific subtree we meet only $S|$, but in another construction we meet $S||$, this means that in the previous SPDH other subgraphs need to be closer to the left in the final layout, but not in the latter SPDH. This is impossible for the same set of serially connected subgraphs. Using the same argument, we conclude that for an S-P tree, each path from subtrees of C_s meets at least one $S|$, one $S||$, or one $S|$ and one $S||$, which is independent of the construction method.

For the parallel case, each path from a subtree of C_p will meet at least a P-type node before this set of subgraphs is further combined in series. Consequently, any construction of this set will contribute the same to the pattern number of each path.

Therefore, each path has the same pattern number under any S-P tree construction algorithm, and the pattern number of an SPDH is unique. ◻

LEMMA 3.3. *The pattern number k of an S-P tree is a lower bound on the number of tracks needed to lay out the corresponding SPDH.*

Proof. The idea is to "peel" the S-P tree until it is isomorphic (including the labels) to the tree of a TS-difficult SPDH $TS\_d(k)$ (see Figure 3.1). This can be done by repeatedly deleting inner nodes which have either one or two leaves and which do not contribute to the pattern number and deleting one of the attached leaves which correspond to vertices in the SPDH.

Recall that in an SPDH, a vertex can have only two incident hyperedges, the incoming and the outgoing hyperedges. If a vertex v is attached to an S-type inner node, v has an incoming hyperedge of the form $(V_{i_s}, \{v\} \cup V_{i_t})$ and an outgoing hyperedge of the form $(\{v\} \cup V_{o_s}, V_{o_t})$, where V_{i_s}, V_{o_t} are nonempty vertex sets and V_{i_t}, V_{o_s} are general vertex sets. Deleting an S-type inner node and one of the attached leaves necessitates removing the corresponding vertex v in the SPDH such that the incoming and outgoing hyperedges $(V_{i_s}, \{v\} \cup V_{i_t})$, $(\{v\} \cup V_{o_s}, V_{o_t})$ are merged into one hyperedge $(V_{i_s} \cup V_{o_s}, V_{i_t} \cup V_{o_t})$.

If a vertex v is attached to a P-type inner node, the incoming hyperedge is of the form $(V_{i_s}, \{v\} \cup V_{i_t})$ and the outgoing hyperedge is of the form $(\{v\} \cup V_{o_s}, V_{o_t})$, where $V_{i_s}, V_{i_t}, V_{o_s}$, and V_{o_t} are nonempty vertex sets. Deleting a P-type inner node and one of the attached leaves means to delete the corresponding vertex v in the SPDH such that the incoming and outgoing hyperedges $(V_{i_s}, \{v\} \cup V_{i_t})$, $(\{v\} \cup V_{o_s}, V_{o_t})$ become (V_{i_s}, V_{i_t}) and (V_{o_s}, V_{o_t}), respectively.

As for the layout, in addition to removing corresponding vertices, in order not to change the number of tracks, the incident hyperedges are either connected using original tracks or remain the same. It depends on whether the deleted inner nodes are S-type or P-type, as shown in the following example.

Figure 3.5 gives an example of peeling. When x_1 and the related S-type inner node is peeled in the S-P tree, the corresponding vertex in the SPDH is removed, the corresponding vertex in the layout is deleted, and the incident hyperedges in the layout are merged. For x_6, which is connected to a P-type node, the corresponding vertices in the SPDH and in the layout are deleted. The resulting layout needs no more tracks than the original one because of the reduced constraints of merging and deleting.

If the original layout requires $o(k)$ tracks (i.e., less than $\Theta(k)$), the resulting SPDH, which has k patterns and is isomorphic to a TS-difficult SPDH shown in Figure 3.1, has $o(k)$ tracks. This is a contradiction of Lemma 3.1. $\square$

However, pattern number is not the only parameter that dictates the number of tracks. This is shown in the following lemma.

LEMMA 3.4. *There exist SPDHs with pattern number* 1 *that require* $\Omega(\log N)$ *tracks.*

Proof. [12] introduced a circuit which can be transformed, as will be mentioned in section 6, to a so-called difficult-SPDH. One of its S-P trees is shown in Figure 3.6. The pattern number is 1, but it was proved in [12] that $\Omega(\log N)$ tracks are necessary to lay it out. $\square$

4. A layout algorithm.

In this section, we propose a layout algorithm which traverses an S-P tree in a bottom-up manner. We will prove that the algorithm needs at most $2(k + m)$ tracks, where k is the pattern number of the given SPDH and the *balancing number* m (which will be defined in Theorem 4.1) is at most $\log N$ (but could be much less), where N is the number of vertices.

The proposed data structure is as follows. For each inner node n, two records, $D(n)$ and $pattern(n)$, are kept. $D(n)$ is the number of necessary tracks. $Pattern(n)$ is the *pattern configuration* of the larger track number of the two subgraphs of node n after tracing up to node n. The possible pattern configurations—denoted by

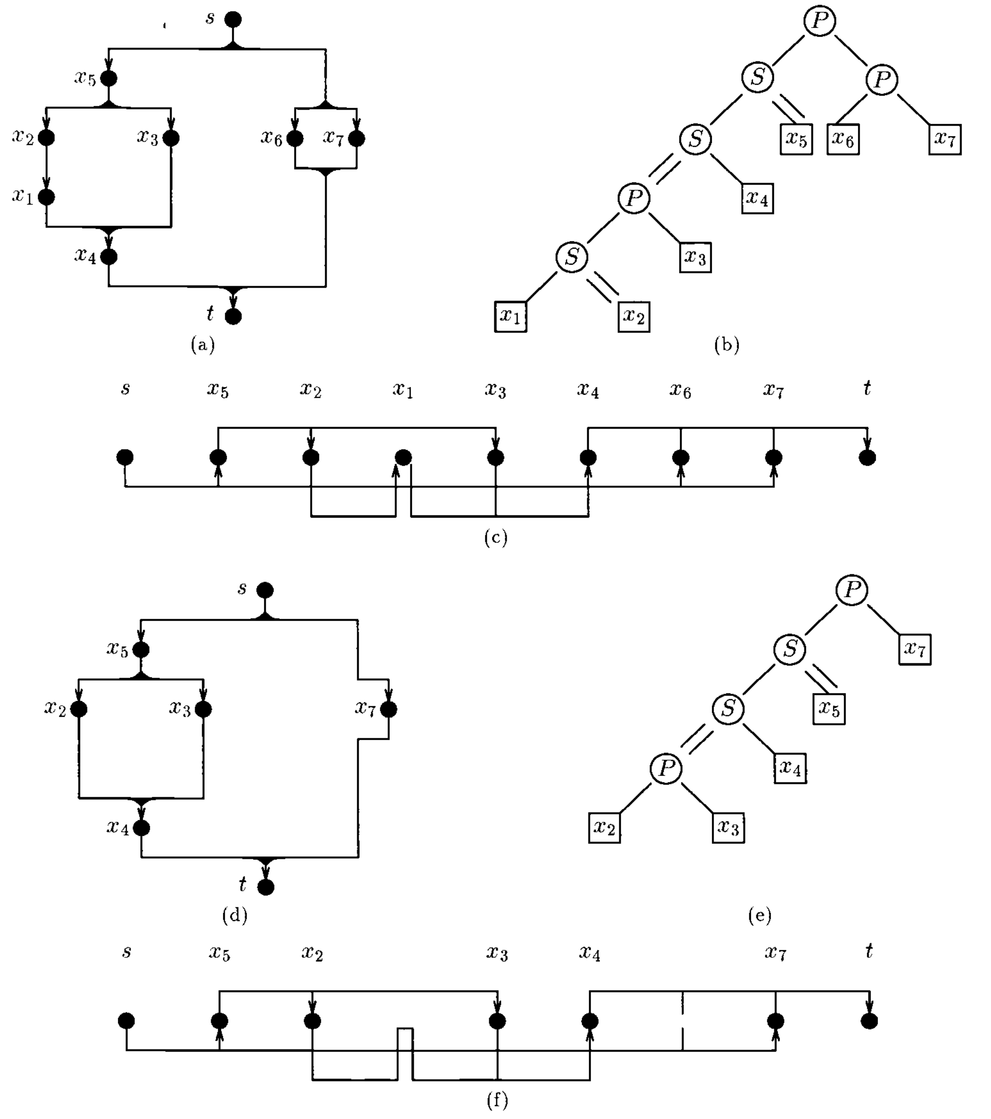


FIG. 3.5. (a) *An SPDH,* (b) *its S-P tree, and* (c) *one possible layout. After vertices* x_1 *and* x_6 *are peeled,* (d) *the resulting SPDH,* (e) *its S-P tree, and* (f) *one possible layout.*

pattern(n)—are $(k), (k, P|), (k, P|, S|)$, and $(k, P|, S||)$. The symbol (k) means that k patterns have been traversed and another $P|$ is now being looked for. $(k, P|)$ means that k patterns and an additional $P|$ have been traversed and another $S|$ or another $S||$ is being looked for. $(k, P|, S|)$ means that k patterns and an additional $P|$ and $S|$ have been traversed, while another $S||$ is being looked for. The last one, $(k, P|, S||)$, means that k patterns and an additional $P|$ and $S||$ have been traversed, while another $S|$ is being looked for.

In the algorithm, each pattern configuration is laid out with a special connection configuration, called a *layout representative.* Layout representatives are shown in

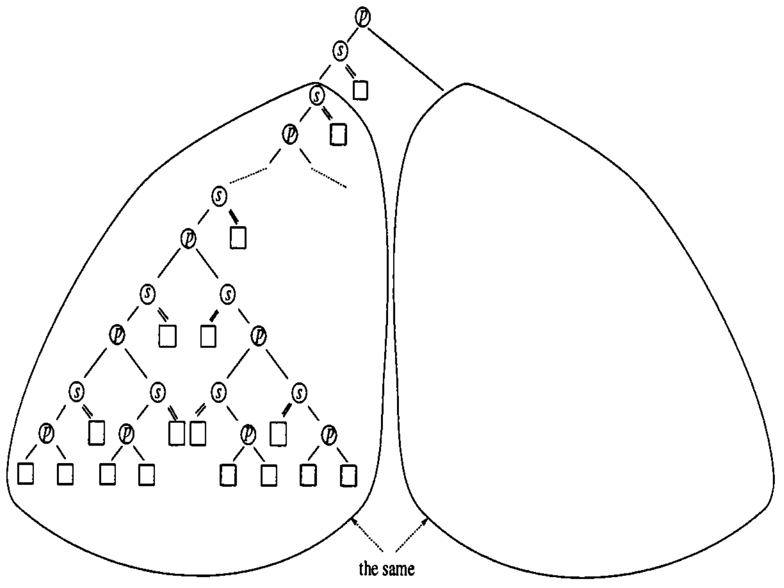


FIG. 3.6. *An S-P tree for a difficult-SPDH.*

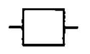
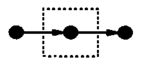
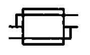
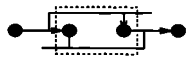
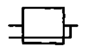
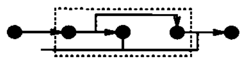
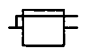
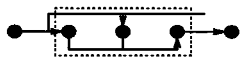
Pattern Configuration	Layout Representative	k	Layout Basis
$\left\langle k \right\rangle$		0	
$\left\langle k \overset{P}{\vert} \right\rangle$		0	
$\left\langle k \overset{P}{\vert} \overset{S}{\vert} \right\rangle$		0	
$\left\langle k \overset{P}{\vert} \overset{S}{\Vert} \right\rangle$		0	

FIG. 4.1. *Pattern configurations and layout representatives for the TS layout algorithm.*

Figure 4.1. For (k), the source and the sink hyperedges are available at the left and the right of the subgraph layout, respectively. For $(k, P|)$, both the source and the sink hyperedges are available at the left and the right. For $(k, P|, S|)$, both the source and the sink hyperedges are available at the left and only the sink hyperedge is available at the right. As for $(k, P|, S||)$, only the source hyperedge is available at the left, but both the source and the sink hyperedges are available at the right. In Figure 4.1, their basis configurations are shown.

We denote the left and the right children of n by l and r, respectively. Also, in order to simplify the proof, if the number of tracks is increased, two tracks will be added, by assigning pseudotrack in some cases (obviously, this is not done in practice). Thus, if $D(l) > D(r)$, $D(l) \geq D(r) + 2$. The main idea of the algorithm is to maintain layout representatives and to control the number of tracks.

TS LAYOUT ALGORITHM.

Input: An S-P tree.

Output: A TS layout.

Traverse each node in a bottom-up manner, and at the same time, lay out the corresponding subgraph according to the following cases until the entire S-P tree has been traversed. Furthermore, reduce the number of tracks by cleaning up the pseudotracks.

There are two cases according to the type of the currently traversed node n. Node n may be either an S-type node or a P-type node.

Case 1. *n is a P-type node.*

Rename n as P_n. According to whether P_n contributes to the pattern configuration or not, there are four cases.

Case 1.1. *P_n is in the pattern*[4] *of the left path but not in the pattern of the right path.*

This condition means that the pattern configuration of the left path below P_n is $(k(l))$ and that of the right path is $(k(r), P|)$, $(k(r), P|, S|)$, or $(k(r), P|, S||)$.

According to the relative values of $D(l)$ and $D(r)$, there are three cases.

Case 1.1.1. $D(l) > D(r)$.

The layout is shown in Figure 4.2, and the result of $D(P_n)$ and $pattern(P_n)$ is

$$D(P_n) = D(l) + 2, \tag{4.1}$$

$$pattern(P_n) = (k(l), P|). \tag{4.2}$$

We can see that the layout has the connection configuration of $(k(l), P|)$.

Case 1.1.2. $D(l) = D(r)$.

According to the pattern numbers, there are two cases.

Case 1.1.2.1. $k(l) \geq k(r)$.

It is similar to Case 1.1.1.

Case 1.1.2.2. $k(l) < k(r)$.

The layout is shown in Figure 4.2, and the result is

$$D(P_n) = D(r) + 2, \tag{4.3}$$

$$pattern(P_n) = pattern(r). \tag{4.4}$$

The three situations correspond to three possible pattern configurations of the right path. All the connection configurations remain the same as those of the layout of the right subgraphs.

Case 1.1.3. $D(l) < D(r)$.

The layout is shown in Figure 4.2, and the result is

$$D(P_n) = D(r), \tag{4.5}$$

$$pattern(P_n) = pattern(r). \tag{4.6}$$

[4]Recall the definition of pattern number in section 3.

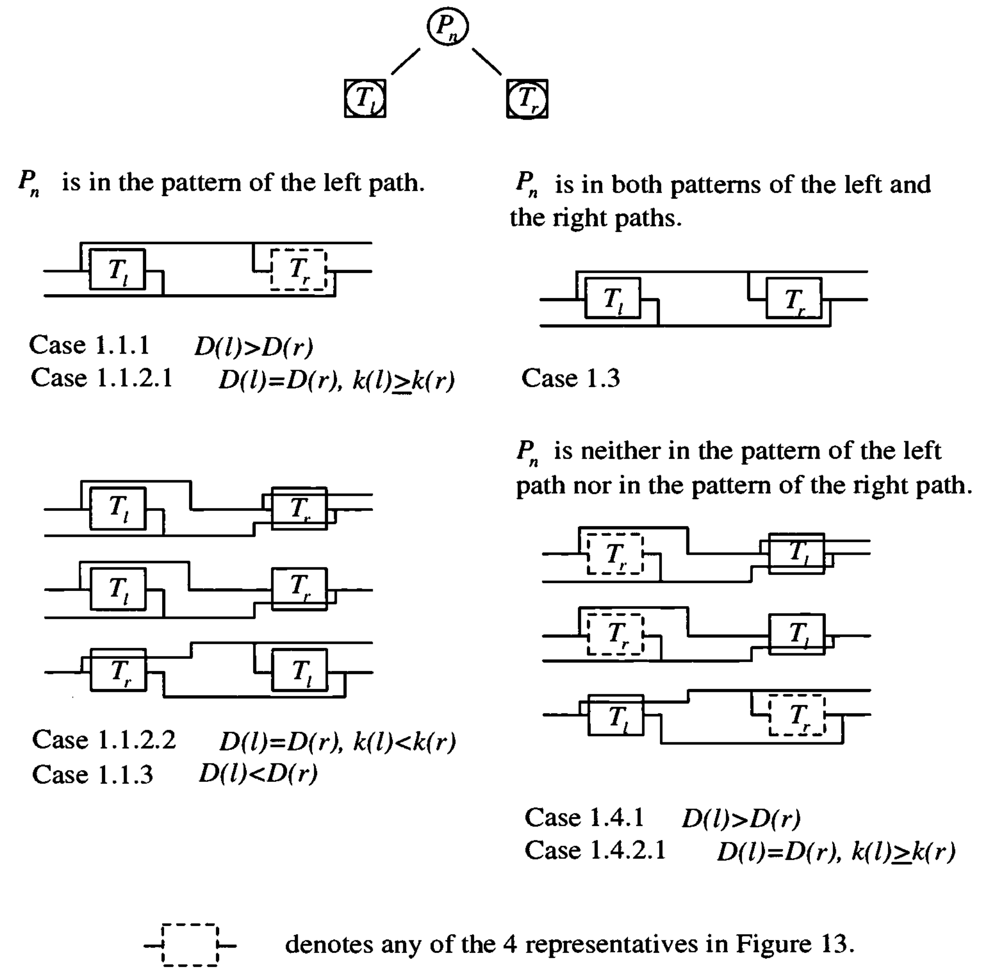

P_n
T_l
T_r
P_n is in the pattern of the left path.
Case 1.1.1 D(l)>D(r)
Case 1.1.2.1 D(l)=D(r), k(l)≥k(r)
Case 1.1.2.2 D(l)=D(r), k(l)<k(r)
Case 1.1.3 D(l)<D(r)
P_n is in both patterns of the left and the right paths.
Case 1.3
P_n is neither in the pattern of the left path nor in the pattern of the right path.
Case 1.4.1 D(l)>D(r)
Case 1.4.2.1 D(l)=D(r), k(l)≥k(r)
denotes any of the 4 representatives in Figure 13.

FIG. 4.2. *Layouts for P-type nodes.*

Case 1.2. *P_n is in the pattern of the right path but not in the pattern of the left path.*

It is similar to Case 1.1.

Case 1.3. *P_n is in both patterns of the left and the right paths.*

This condition means that the pattern configurations of the left path and the right path below P_n are $(k(l))$ and $(k(r))$, respectively.

Case 1.3.1. $D(l) > D(r)$.

It is similar to Case 1.1.1.

Case 1.3.2. $D(l) = D(r)$.

The layout is shown in Figure 4.2, and the result is

$$D(P_n) = D(l) + 2. \tag{4.7}$$

Case 1.3.2.1. $k(l) \geq k(r)$.

$$pattern(P_n) = (k(l), P|). \tag{4.8}$$

Case 1.3.2.2. $k(l) < k(r)$.

$$pattern(P_n) = (k(r), P|). \tag{4.9}$$

Case 1.3.3. $D(l) < D(r)$.

It is symmetrical to Case 1.3.1.

Case 1.4. *P_n is neither in the pattern of the left path nor in the pattern of the right path.*

This condition means that the pattern configuration of the left path below P_n is $(k(l), P|)$, $(k(l), P|, S|)$, or $(k(l), P|, S||)$ and that of the right path is $(k(r), P|)$, $(k(r), P|, S|)$, or $(k(r), P|, S||)$.

Case 1.4.1. $D(l) > D(r)$.

The layout is shown in Figure 4.2, and the result is

$$D(P_n) = D(l), \tag{4.10}$$

$$pattern(P_n) = pattern(l). \tag{4.11}$$

Case 1.4.2. $D(l) = D(r)$.

Case 1.4.2.1. $k(l) \geq k(r)$.

The layout is shown in Figure 4.2, and the result is

$$D(P_n) = D(l) + 2, \tag{4.12}$$

$$pattern(P_n) = pattern(l). \tag{4.13}$$

Case 1.4.2.2. $k(l) < k(r)$.

It is symmetrical to Case 1.4.2.1.

Case 1.4.3. $D(l) < D(r)$.

It is symmetrical to Case 1.4.1.

Case 2. *n is an S-type node.*

Rename n as S_n and put the subtree with the double line to the left side of S_n.

Case 2.1. *S_n is in the pattern of the left path but not in the pattern of the right path.*

This condition means that the pattern configuration of the left path below S_n is either $(k(l), P|)$ or $(k(l), P|, S|)$ and that of the right path is either $(k(r))$ or $(k(r), P|, S|)$.

Case 2.1.1. $D(l) > D(r)$.

The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(l), \tag{4.14}$$

$$pattern(S_n) = pattern(l) \cup (S||), \tag{4.15}$$

where $\cup$ means the original $pattern(l)$ meets one more $S||$. For now, that is either $(k(l), P|, S||)$ or $(k(l) + 1)$.

Case 2.1.2. $D(l) = D(r)$.

Case 2.1.2.1. $k(l) \geq k(r)$.

The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(l) + 2 \quad or \quad D(l), \tag{4.16}$$

$$pattern(S_n) = pattern(l) \cup (S||). \tag{4.17}$$

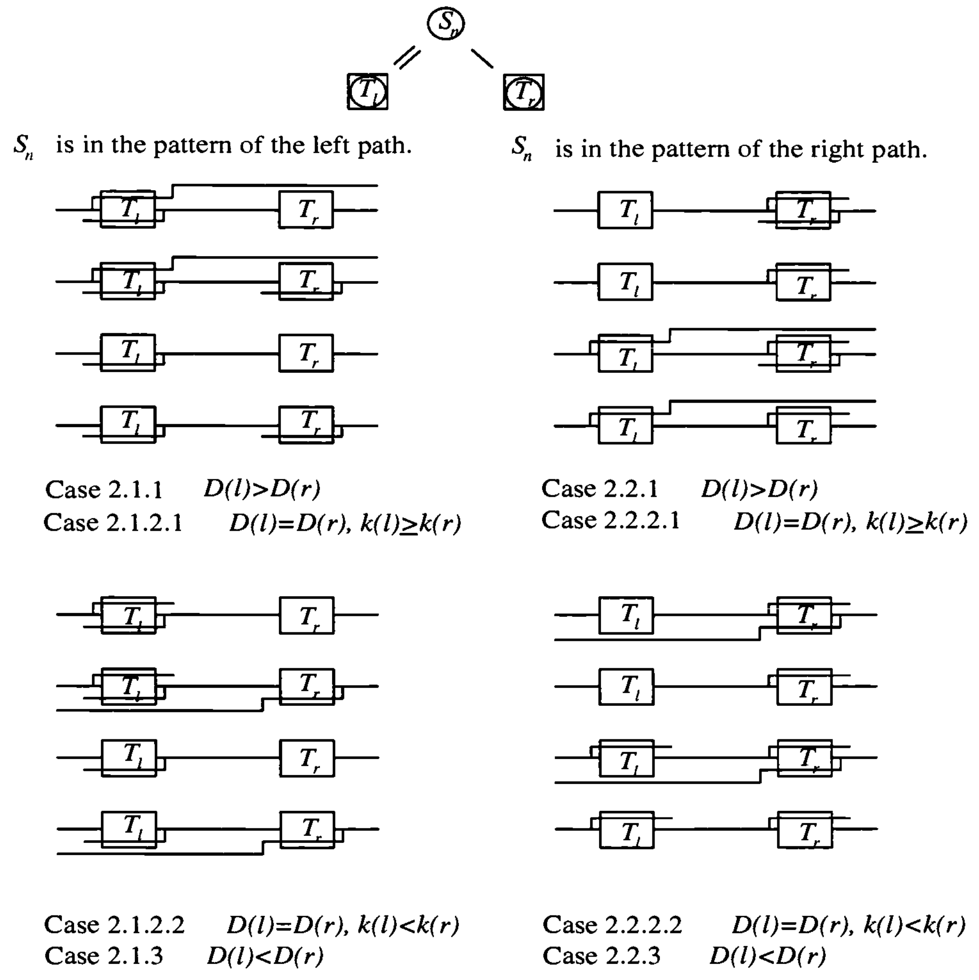

S_n
T_l
T_r
S_n is in the pattern of the left path.
S_n is in the pattern of the right path.
Case 2.1.1 D(l)>D(r)
Case 2.1.2.1 D(l)=D(r), k(l)≥k(r)
Case 2.2.1 D(l)>D(r)
Case 2.2.2.1 D(l)=D(r), k(l)≥k(r)
Case 2.1.2.2 D(l)=D(r), k(l)<k(r)
Case 2.1.3 D(l)<D(r)
Case 2.2.2.2 D(l)=D(r), k(l)<k(r)
Case 2.2.3 D(l)<D(r)

FIG. 4.3. *Layouts for S-type nodes: part* 1.

Case 2.1.2.2. $k(l) < k(r)$.
The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(r) + 2 \quad or \quad D(r), \tag{4.18}$$

$$pattern(S_n) = pattern(r). \tag{4.19}$$

Case 2.1.3. $D(l) < D(r)$.
The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(r), \tag{4.20}$$

$$pattern(S_n) = pattern(r). \tag{4.21}$$

Case 2.2. *S_n is not in the pattern of the left path but is in the pattern of the right path.*

This condition means that the pattern configuration of the left path below S_n is either $(k(l))$ or $(k(l), P|, S||)$ and that of the right path is either $(k(r), P|)$ or $(k(r), P|, S||)$. Therefore, there are four situations for different cases.

Case 2.2.1. $D(l) > D(r)$.

The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(l), \tag{4.22}$$

$$pattern(S_n) = pattern(l). \tag{4.23}$$

Case 2.2.2. $D(l) = D(r)$.

Case 2.2.2.1. $k(l) \geq k(r)$.

It is similar to Case 2.2.1, except that $D(S_n)$ is $D(l)+2$ in some cases.

Case 2.2.2.2. $k(l) < k(r)$.

The layout is shown in Figure 4.3, and the result is

$$D(S_n) = D(r) + 2 \quad or \quad D(r), \tag{4.24}$$

$$pattern(S_n) = pattern(r) \cup (S|). \tag{4.25}$$

Case 2.2.3. $D(l) < D(r)$.

It is similar to Case 2.2.2.2 except that $D(S_n)$ is only $D(l)$.

Case 2.3. *S_n is in both patterns of the left and the right paths.*

This condition means that the pattern configuration of the left path below S_n is either $(k(l), P|)$ or $(k(l), P|, S|)$ and that of right path below S_n is either $(k(r), P|)$ or $(k(r), P|, S||)$.

Case 2.3.1. $D(l) > D(r)$.

The layout is shown in Figure 4.4, and the result is

$$D(S_n) = D(l), \tag{4.26}$$

$$pattern(S_n) = pattern(l) \cup (S||). \tag{4.27}$$

Case 2.3.2. $D(l) = D(r)$.

Case 2.3.2.1. $k(l) \geq k(r)$.

It is similar to Case 2.3.1 except that $D(S_n)$ is $D(l)+2$ in some cases.

Case 2.3.2.2. $k(l) < k(r)$.

The layout is shown in Figure 4.4, and the result is

$$D(S_n) = D(r) + 2 \quad or \quad D(r), \tag{4.28}$$

$$pattern(S_n) = pattern(r) \cup (S|). \tag{4.29}$$

Case 2.3.3. $D(l) < D(r)$.

It is similar to Case 2.3.2.2 except that $D(S_n)$ is only $D(r)$.

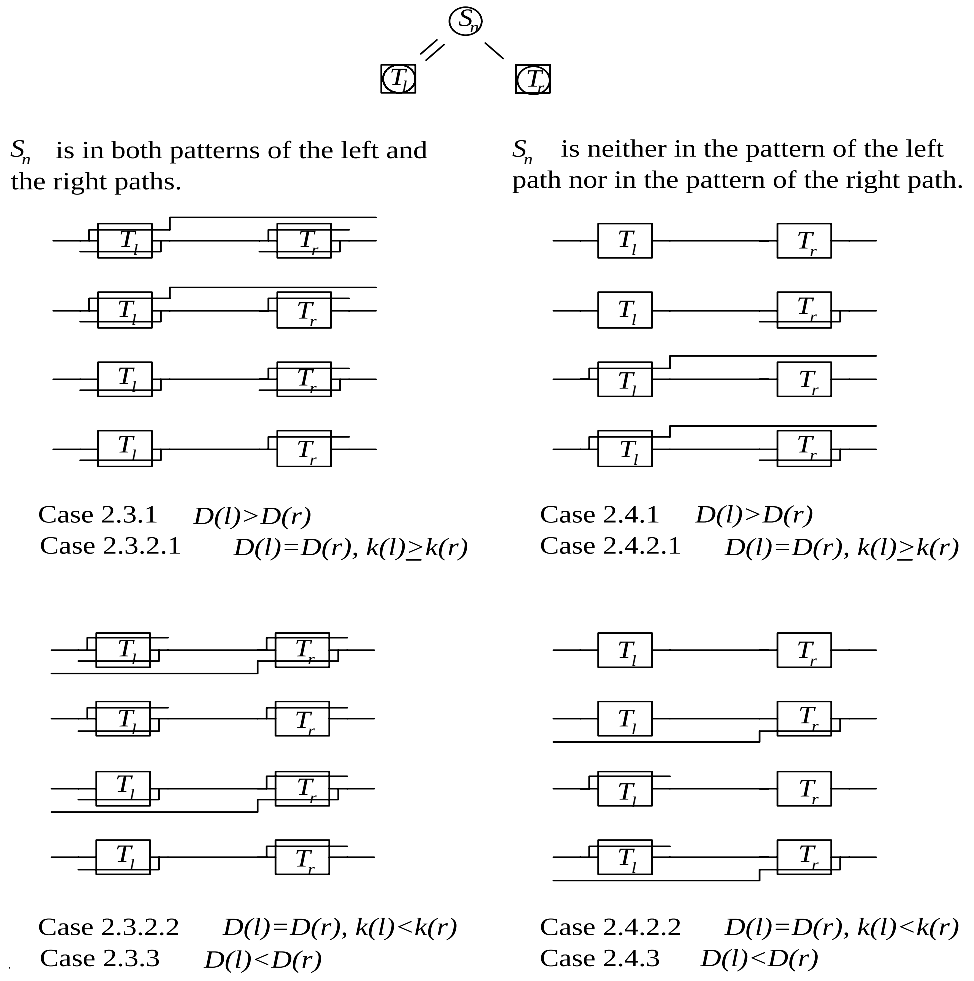

S_n
T_l
T_r
S_n is in both patterns of the left and the right paths.
S_n is neither in the pattern of the left path nor in the pattern of the right path.
Case 2.3.1 D(l)>D(r)
Case 2.3.2.1 D(l)=D(r), k(l)≥k(r)
Case 2.4.1 D(l)>D(r)
Case 2.4.2.1 D(l)=D(r), k(l)≥k(r)
Case 2.3.2.2 D(l)=D(r), k(l)<k(r)
Case 2.3.3 D(l)<D(r)
Case 2.4.2.2 D(l)=D(r), k(l)<k(r)
Case 2.4.3 D(l)<D(r)

FIG. 4.4. *Layouts for S-type nodes: part 2.*

Case 2.4. *S_n is neither in the pattern of the left path nor in the pattern of the right path.*

This condition means that the *pattern configuration* of the left path below S_n is either $(k(l))$ or $(k(l), P|, S||)$ and that of the right path is either $(k(r))$ or $(k(r), P|, S|)$.

Case 2.4.1. $D(l) > D(r)$.

The layout is shown in Figure 4.4, and the result is similar to Case 2.2.1.

Case 2.4.2. $D(l) = D(r)$.

Case 2.4.2.1. $k(l) \geq k(r)$.

The layout is shown in Figure 4.4, and the result is similar to Case 2.2.2.1.

Case 2.4.2.2. $k(l) < k(r)$.

The layout is shown in Figure 4.4, and the result is similar to Case 2.1.2.2.

Case 2.4.3. $D(l) < D(r)$.

The layout is shown in Figure 4.4, and the result is similar to Case 2.1.3.

Since this algorithm traverses each node only once and it takes constant time to process each node, it runs in linear time. After the above layout is obtained, we can trace down the S-P tree from the root and along the larger track number subtree. If the number of tracks is equal, we choose the one with the larger pattern number. In addition, if both track numbers and pattern numbers are the same, we choose the left one. We call this path the *largest track number path*. Along this path down, the number of tracks will decrease by two only if either of the following two events happen. We pass a $P|$ node which belongs to a pattern of the largest track number path or a node with the two subtrees having the same number of tracks.

THEOREM 4.1. *The layout produced by the TS layout algorithm uses at most $D = 2(k+m)$ tracks, where k is the pattern number and m is the balancing number; $m \leq \log N$.*

Proof. In the following, **A** represents a $P|$ which belongs to a pattern and during the layout procedure makes the number of tracks increase by two. **B** denotes a node with its two subtrees having the same number of tracks and it also causes the number of tracks to increase by two.

Suppose we already have a layout that is produced by the TS layout algorithm and the pattern number for the corresponding S-P tree is k. The minimal vertex number will occur when the largest track number path has k patterns and other paths have pattern numbers as small as possible. The reason is that the more patterns, the more tracks are needed for the layout. Thus, when the track number is fixed, fewer vertices are needed.

Here, $N(k,m)$ denotes the minimal vertex number when there are k **A**-type nodes and m **B**-type nodes in the largest track number path. The value m is called the *balancing number*.

Next, we apply induction on m to prove that $N(k,m) \geq 2^m$, where $k \geq 0$.

Basis: $m = 0$. It is clear that $N(k,0) \geq 2^0$.

Induction hypothesis: $N(k,m) \geq 2^m$.

Induction step: For $N(k,m+1)$, we trace down the largest track number path until we meet the first **B**-type node, meaning the subtrees of this **B**-type node have the same number of layout tracks. Suppose we have passed k_1 **A**-type nodes. The corresponding subtrees have $N(k_l,m_l)$ and $N(k_r,m_r)$ vertices, and the largest track number path is along the left path. Then $k_1 + k_l = k$ and $m_l = m$. The following two reasons show $m_r \geq m_l = m$. First, the right subtree needs to have the same number of tracks as the left subtree. Second, $k_r \leq k_l$, as a result of that the largest track number path chooses the larger pattern number under the situation of equal number of tracks. Therefore,

$$N(k,m+1) \geq N(k_l,m_l)+N(k_r,m_r) \geq N(k_l,m)+N(k_r,m) \geq 2^m+2^m \geq 2^{m+1} \tag{4.30}$$

We conclude that $D = 2(k+m) \leq 2k + 2\log N$. □

5. A reordering and layout algorithm for an SPDH and its dual. In this section we show how serial path reordering can reduce necessary layout tracks of an SPDH $\mathcal{P}$ and its dual $\mathcal{N}$ simultaneously to $O(\log N)$, where N is the number of vertices. In addition, we propose an algorithm which reorders and lays out an SPDH.

A *simultaneous layout* of an SPDH $\mathcal{P}$ consists of two layouts—one corresponding to $\mathcal{P}$ and the other corresponding to its dual $\mathcal{N}$—with the same vertex order. The

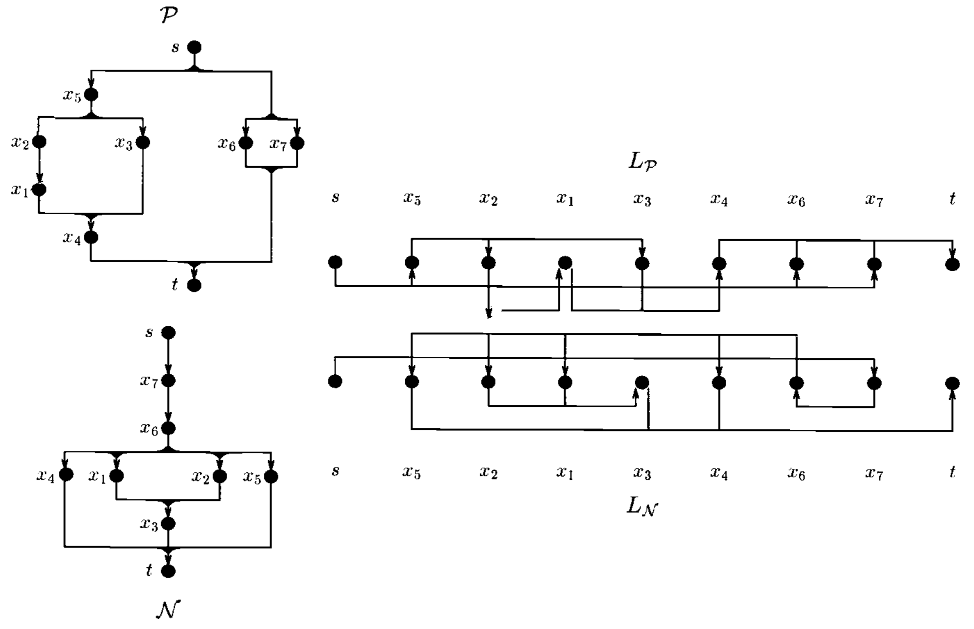


FIG. 5.1. *A simultaneous layout, including $L_{\mathcal{P}}$ and $L_{\mathcal{N}}$, of an SPDH $\mathcal{P}$ and its dual $\mathcal{N}$.*

names—$\mathcal{P}$ and $\mathcal{N}$—are motivated by applications in complementary metal oxide-silicon (CMOS) circuits, which consist of p-channel MOSFET (PMOS) and n-channel MOSFET (NMOS) parts. Figure 5.1 shows a simultaneous layout, consisting of $L_{\mathcal{P}}$ and $L_{\mathcal{N}}$, of an SPDH $\mathcal{P}$ and its dual $\mathcal{N}$. Notice that the ordering of the vertices in $L_{\mathcal{P}}$ and $L_{\mathcal{N}}$ is the same. *Serial path reordering* rearranges the serial connections of SPDHs. The effect of serial path reordering is illustrated in Figure 5.2, where the reordered SPDH needs only two tracks instead of the $O(N)$ tracks needed for the original TS-difficult SPDH of Figure 3.1. The resulting SPDH, S-P tree, and layout are also shown.

Recall the definitions of section 1.1. An SPDH and its dual have distinct S-P trees. These two trees are dual to each other. In our construction procedure, if one inner node in a tree is S-type, the corresponding inner node in the other tree is P-type, and vice versa.

The reordering and layout algorithm is summarized as follows:

1. Traverse each inner node of the S-P trees in a bottom-up manner.
2. Decide the relative positions of subtree layouts under the P-type node of the two currently traversed inner nodes.
3. Consequently, fix the single line and double line positions of the corresponding S-type node in the other tree.
4. Keep a *track number D* of necessary tracks for each node of each S-P tree.

There are only two types of layout representatives participating in the algorithm. Recall that a layout representative is a special connection configuration. These two layout representatives are shown in Figure 5.3. One is called a *Double-Double*, with both the source and the sink hyperedges available at the left and the right. The other is called a *Single-Double*, with only the source hyperedge available at the left but both the source and the sink hyperedges available at the right.

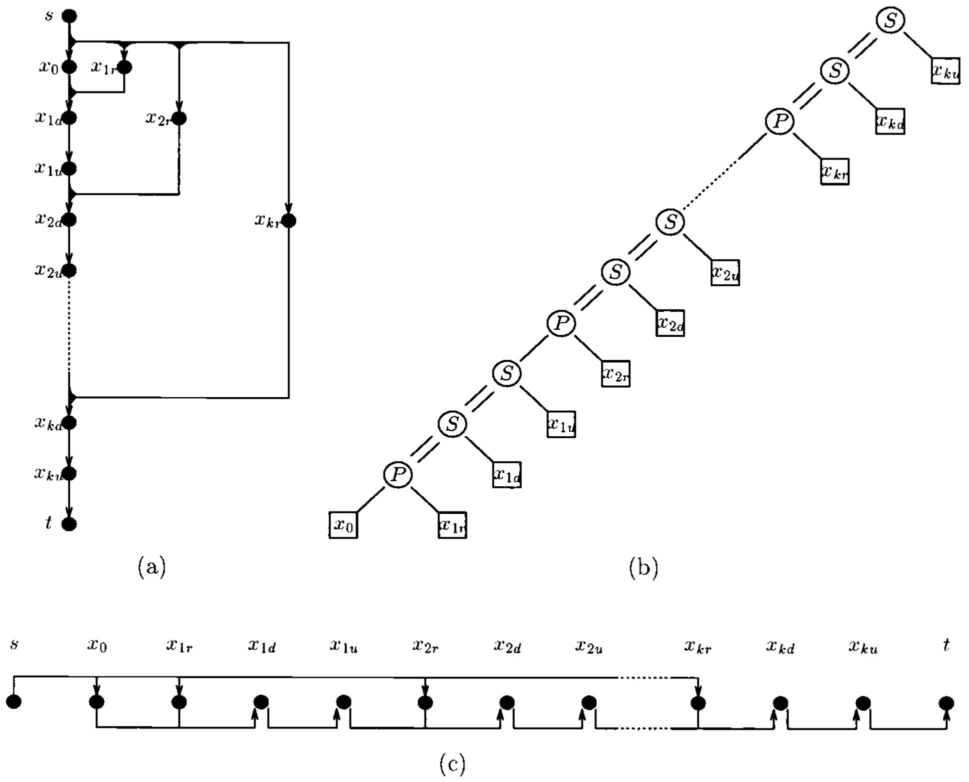


FIG. 5.2. *After reordering the TS-difficult SPDH in Figure* 3.1*, the corresponding* (a) *SPDH,* (b) *S-P tree, and* (c) *layout are shown.*

The algorithm has three characteristics:

1. During a bottom-up traversal of an inner node, each layout of its two subtrees are one of the two layout representatives, and the resulting layout will also be one of these two layout representatives.
2. The track number D of each P-type node is kept (by assigning pseudotracks in some cases) the same as that of the corresponding S-type node.
3. A track number is larger than the maximum of the track numbers of the two children if and only if the two subtrees of the corresponding node have the same track number. Also, all track numbers are kept odd.

We benefit from limiting the layout of each subtree to one of the two layout representatives as mentioned in the first characteristic and keeping track numbers as mentioned in the second characteristic. The benefit is that when traversing inner nodes we need to consider only three cases. These cases are divided according to the relative track numbers of the two subtrees of the currently traversed P-type inner node. The third characteristic (which is also about the track numbers) simplifies the proof of Theorem 5.1: this algorithm uses $O(\log N)$ tracks, where N is the vertex number of the input SPDH or its dual.

Regarding data structure, for each inner node n we keep two records: a track number, $D(n)$, and the layout representative for the n-rooted subtree, *layout-representative(n)*.

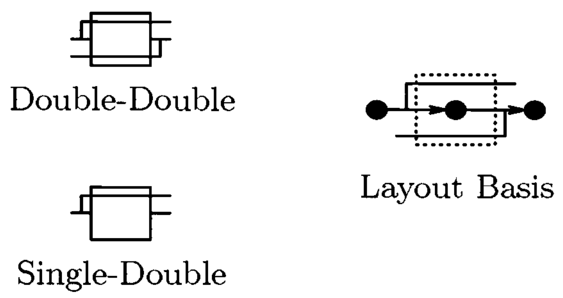


FIG. 5.3. *The two layout representatives and the layout basis.*

REORDERING AND LAYOUT ALGORITHM.

Input: An S-P tree and its dual.

Output: A TS simultaneous layout.

We lay out each vertex, a leaf in the trees, as a layout basis shown in Figure 5.3 and assign each layout basis $D = 3$ tracks. Then we traverse the nodes of both S-P trees simultaneously in a bottom-up manner and maintain the layout representatives according to the P-type node as in the cases described below. Finally, unnecessary tracks are cleaned up.

Case 1. *$D(P_l) = D(P_r)$ and $D(S_l) = D(S_r)$,*
where P_l, P_r are the left child and the right child of the P-type node P_n, respectively. Similarly, S_l, S_r are the left child and the right child of the corresponding S-type node S_n in the other S-P tree.
The resulting layouts are shown in Figure 5.4, and there are subcases depending on what the layout representatives of the corresponding subtrees are. The results of $D(P_n)$ and $D(S_n)$ are

$$D(P_n) = D(P_l) + 2 \ \ \textit{and} \ \ D(S_n) = D(S_l) + 2. \tag{5.1}$$

We can see that each of the resulting layout connections is one of the two types of layout representatives.

Case 2. *$D(P_l) > D(P_r)$ and $D(S_l) > D(S_r)$.*
The resulting layouts are also shown in Figure 5.4. The results of $D(P_n)$ and $D(S_n)$ are

$$D(P_n) = D(P_l) \ \ \textit{and} \ \ D(S_n) = D(S_l). \tag{5.2}$$

We can see that the resulting layout connections are still among the two layout representatives.

Case 3. *$D(P_l) < D(P_r)$ and $D(S_l) < D(S_r)$.*
It is similar to Case 2.

THEOREM 5.1. *The layout produced by the reordering and layout algorithm uses at most $D = 2\log(N-2) + 3$ tracks, where $N \geq 3$ is the number of vertices of SPDH $\mathcal{P}$ or SPDH $\mathcal{N}$.*

Proof. Assume $N(D)$ is the vertex number of an SPDH when the algorithm requires D tracks for the layout. Thus, $N' = N - 2$ is the number of vertices excluding the source and the sink. This lemma is proved by induction on N'.

Basis: $N' = 1$ and $D = 3$. It is clear that $N' = 1 \geq 2^0 = 2^{\frac{D-3}{2}}$.

Induction hypothesis: $N'(D) \geq 2^{\frac{D-3}{2}}$.

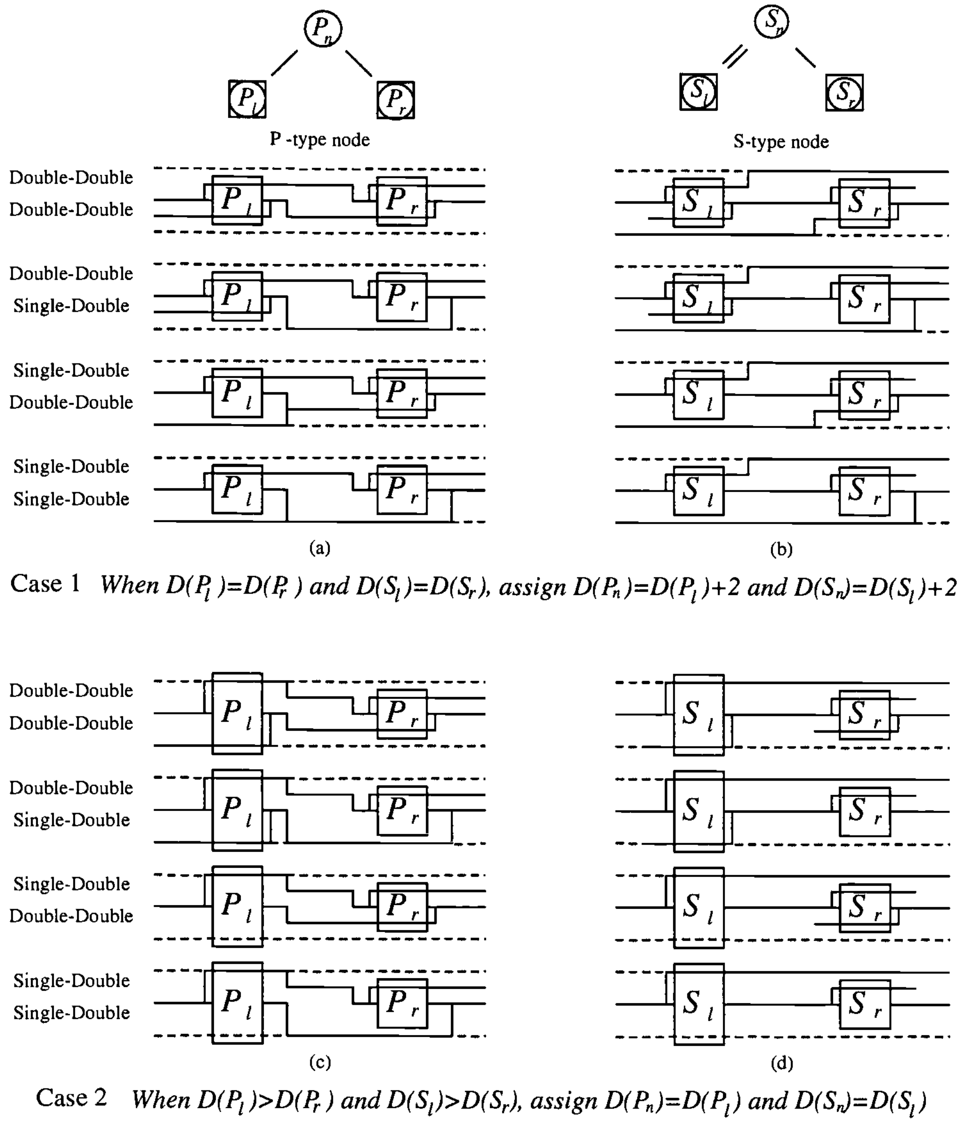

P_n
P_l
P_r
P -type node
S_n
S_l
S_r
S-type node
Double-Double
Double-Double
Double-Double
Single-Double
Single-Double
Double-Double
Single-Double
Single-Double
(a)
(b)
Case 1 When $D(P_l)=D(P_r)$ and $D(S_l)=D(S_r)$, assign $D(P_n)=D(P_l)+2$ and $D(S_n)=D(S_l)+2$
(c)
(d)
Case 2 When $D(P_l)>D(P_r)$ and $D(S_l)>D(S_r)$, assign $D(P_n)=D(P_l)$ and $D(S_n)=D(S_l)$

FIG. 5.4. (a), (c) *For a P-type node, there are two cases shown. Each case has four subcases of subtree layouts.* (b), (d) *The layouts of the corresponding S-type node are shown. Dashed lines are pseudotracks.*

Induction step: Because of the specific construction procedure in the algorithm, D is increased by 2 when the two subtrees have the same track number. Therefore,

$$N'(D+2) \geq N'(D) + N'(D) \geq 2^{\frac{(D+2)-3}{2}}, \tag{5.3}$$

and

$$D \leq 2\log N' + 3 = 2\log(N-2) + 3. \qquad \square \tag{5.4}$$

Similar to the TS layout algorithm, the reordering and layout algorithm also tra-

$$\text{output} = (x_5 \times (x_2 \times x_1 + x_3) \times x_4) + (x_6 + x_7)$$

(a)

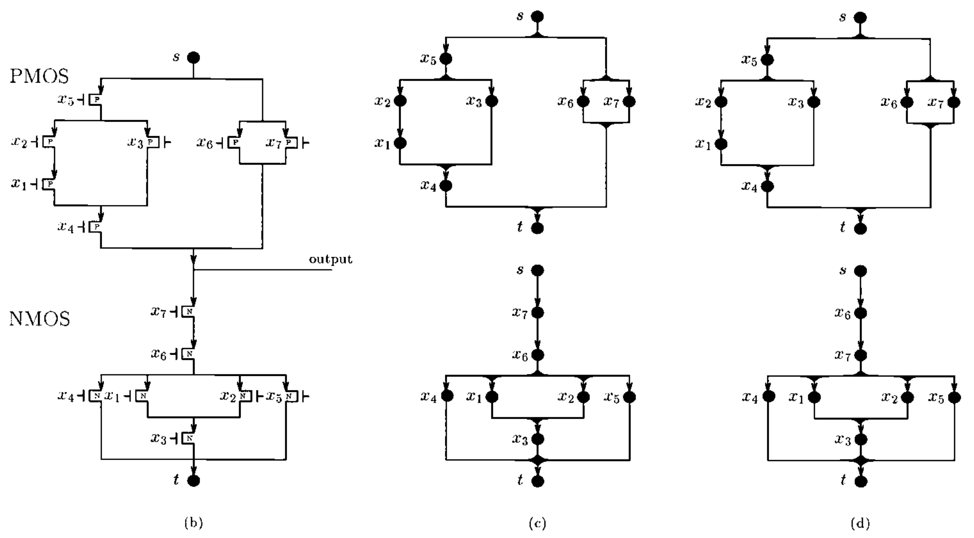


FIG. 6.1. (a) *A function* $(x_5 \times (x_2 \times x_1 + x_3) \times x_4) + (x_6 + x_7)$. (b) *A CMOS circuit which implements the function.* (c) *The SPDHs transformed from the circuit.* (d) *The SPDHs after the serial paths are reordered. Notice that* x_6 *and* x_7 *in the SPDH of the NMOS circuit have been reordered.*

verses each node once, and it takes constant time to process each node. Consequently, it runs in linear time. The algorithm is demonstrated in the next section.

6. Applications and examples.

This section applies the minmax mincut algorithms to circuit design. This application produces layouts of CMOS functional cells using the gate-matrix layout style. Two circuits are used to demonstrate the reordering and layout algorithm.

In [10], the gate-matrix style was introduced to implement circuits. There, the "intersections" of polysilicon columns and diffusion rows form transistors and horizontal metal lines are used as connections. Gate-matrix design includes assigning transistors to columns and using metal lines to implement net lists. A CMOS functional cell is formed by series-parallel connections of transistors [14]; the PMOS and NMOS sides of the circuit are the dual of each other; a general CMOS circuit is an interconnection of CMOS functional cells.

Figures 6.1(a), (b), and Figure 6.2(b) show a function, a circuit which implements the function, and a layout of the circuit, respectively. Notice that since a MOSFET transistor is bidirectional, NMOS and PMOS circuits will function the same even if the positions of V_{DD} and V_{SS} are switched. Therefore, the NMOS layout of Figure 6.2(b) will function the same when current flows either from right to left or from left to right.

A CMOS functional cell is represented by two SPDHs as follows: Both the PMOS circuit and the NMOS circuit are represented by SPDHs. Each MOSFET transistor is represented by a vertex. A net is represented by a hyperedge. Node V_{DD} is represented by a source. The output of the CMOS functional cell is represented

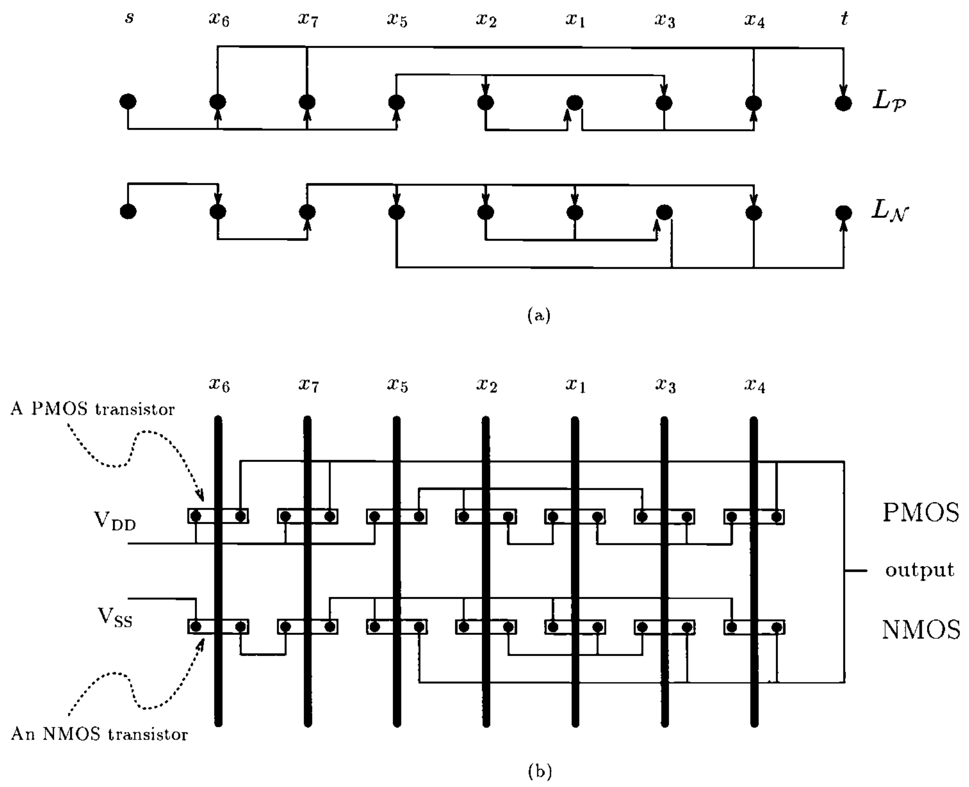


FIG. 6.2. (a) *A simultaneous layout of the SPDHs in Figure* 6.1(d). (b) *A layout of the circuit in Figure* 6.1(b).

by either the sink of an SPDH corresponding to the PMOS circuit or the source of an SPDH corresponding to the NMOS circuit. Node V_{SS} is represented by a sink. In this transformation, the gate-matrix area (height) minimization problem with CMOS functional cells as inputs is equal to the mincut linear arrangement of a SPDH problem.

Minimizing the longest path in an SPDH layout, which has been discussed in this paper, corresponds to minimizing the longest current path in circuit design. Such minimization speeds up the corresponding circuits. Furthermore, routing a path with bends usually requires more vias to connect horizontal and vertical wires than routing a path without any bends. Vias have high resistances, and thus they may slow the circuit. Therefore, our layouts, which are topologically sorted and have no bends, reduce resistances. Consequently, this reduction speeds up the corresponding circuits as well. In other words, the minmax mincut problem corresponds to an area minimization problem that addresses delay minimization. Timing issues in the gate-matrix layout style have not been addressed in the literature.

Notice that here we assume V_{DD} is at the leftmost upper position and V_{SS} is at the leftmost lower position, as shown in Figure 6.2(b). However, the power line may be available on the top and the ground line available on the bottom in some cases. In these cases, at most two tracks, the track for the power line and the track for the ground line, are saved. Therefore, the bounds mentioned remain the same.

Our timing-driven CMOS layout procedure is (conceptually) illustrated in Figures 6.1 and 6.2. The procedure is summarized as follows:

1. Accept an equation as an input (see Figure 6.1(a)).
2. Form a CMOS circuit to implement the function (Figure 6.1(b)).
3. Convert the circuit to two SPDHs: one corresponding to the PMOS circuit and the other corresponding to the NMOS circuit (Figure 6.1(c)).
4. Apply the reordering and layout algorithm to produce a simultaneous layout (Figures 6.1(d), 6.2(a)).
5. Finally, convert the simultaneous layout to a circuit layout (Figure 6.2(b)).

In circuit applications, we modify the reordering and layout algorithm to directly accept circuits as inputs and to produce circuit layouts as outputs. Two examples are demonstrated in the following.

Figure 6.3 shows a carry look-ahead circuit [14], an S-P tree corresponding to the PMOS circuit, and another S-P tree corresponding to the NMOS circuit. Figure 6.4 shows the resulting circuit and S-P trees after the reordering and layout algorithm is applied.

Figure 6.5(a) shows the resulting layouts after the first three steps of the reordering and layout algorithm. In the first step of the bottom-up traverse, the P-type node of the NMOS S-P tree belongs to Case 1 (see Figure 5.4). Both leaves are Double-Double-type layout representatives, and the resulting layouts are Double-Double type for both the NMOS and the PMOS circuits. In the second step, the P-type node of the PMOS S-P tree belongs to Case 2. The two subcircuits of the P-type node are Double-Double type. The resulting PMOS circuit layout is Double-Double type, and the resulting NMOS circuit layout is Single-Double type. The algorithm continues until it finishes traversing the S-P trees. Figure 6.5(b) shows the result, and Figure 6.5(c) shows the layout after redundant wires are cleaned up and the output signal line is connected.

Figure 6.6 shows another example from [12]. Its S-P trees corresponding to the PMOS circuit and the NMOS circuit are shown in Figures 6.7 and 6.8, respectively. Our layout of [12]'s circuit is shown in Figure 6.9. For either the PMOS or the NMOS circuit, 11 tracks are necessary for the 38 transistors. The layout in [12] uses the same number of tracks without minimizing the longest path. The number of tracks of our layout can be further reduced after the redundant wires are cleaned up. Notice that we have taken advantage of reordering the serial paths.

7. Summary and conclusion. After introducing the terminology, which basically defines what an SPDH and a layout are, we have formulated the minmax mincut problem. It accepts an SPDH and produces a minimum cutwidth linear layout subject to the constraints that the source is at the leftmost position, the sink is at the rightmost position, and the longest path is minimized.

The constraints of the minmax mincut problem limit the solution space to TS-layouts, which are created by topological sorting. To explore the possible cutwidths (the track numbers), we define the pattern number k and prove that it is a lower bound of the cutwidths for a specific instance. Another lower bound is the balancing number m.

We have proposed two algorithms; both algorithms traverse corresponding S-P trees in a bottom-up manner and run in linear time. The main idea is to maintain suitable layout representatives to ensure proper growth of the number of necessary tracks.

The TS layout algorithm lays out an SPDH with at most $2(k+m)$ tracks, where k

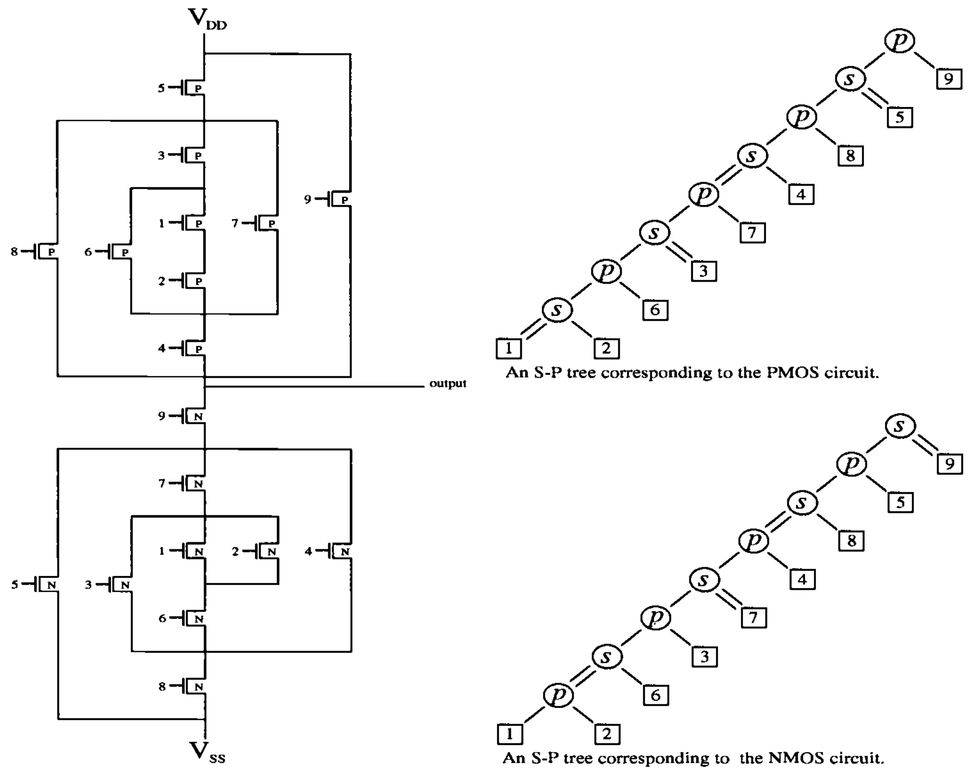

V_DD
output
V_SS
An S-P tree corresponding to the PMOS circuit.
An S-P tree corresponding to the NMOS circuit.

FIG. 6.3. *A carry look-ahead circuit from* [14], *an S-P tree corresponding to the PMOS circuit, and an S-P tree corresponding to the NMOS circuit.*

is the pattern number and m is the balancing number of the SPDH. If reordering the serial paths is allowed, the reordering and layout algorithm simultaneously lays out an SPDH and its dual with the same vertex order, and this algorithm uses $O(\log N)$ tracks, where N is the number of vertices.

We discussed CMOS circuits and their gate-matrix layout style. Then we interpreted the minmax mincut problem as a timing-driven gate-matrix layout problem for CMOS since a topologically sorted solution not only minimizes the longest path but also reduces resistances in circuit design. The timing-driven gate-matrix layout problem is an issue which has not been addressed in the literature.

One possible approach to reduce the number of tracks is as follows. We use the techniques proposed here to fix the ordering of the columns (or transistors). Then we use one of the classical channel routers (e.g., [9, 16]) to obtain a detour-free layout.

In short, motivated by CMOS circuit design we have formulated a novel linear arrangement problem, explored basic properties, and proposed two algorithms. Since performance issues are being paid close attention to, we expect more applications to emerge.

There are several possible research directions. To further improve the number of tracks is a problem that needs further research. To relax the input constraint from an SPDH to a general directed acyclic hypergraph remains open (e.g., interconnection of two or more functional cells).

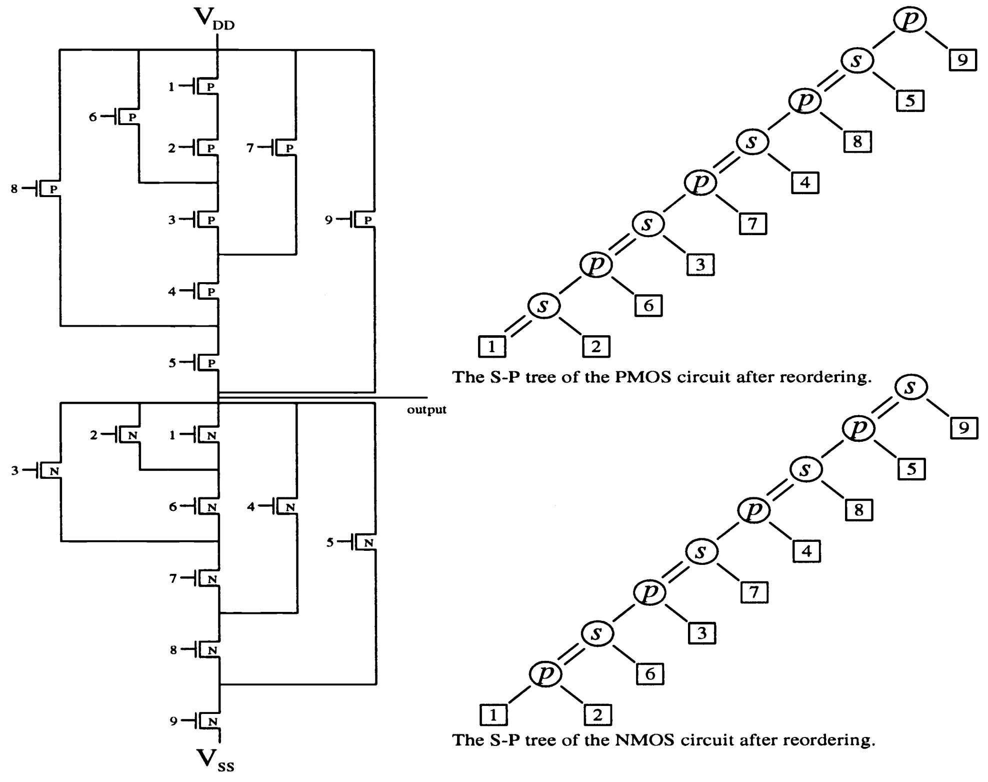

V_DD
V_SS
output
The S-P tree of the PMOS circuit after reordering.
The S-P tree of the NMOS circuit after reordering.

FIG. 6.4. *The carry look-ahead circuit and its S-P trees of the PMOS and the NMOS circuits after the serial paths are reordered.*

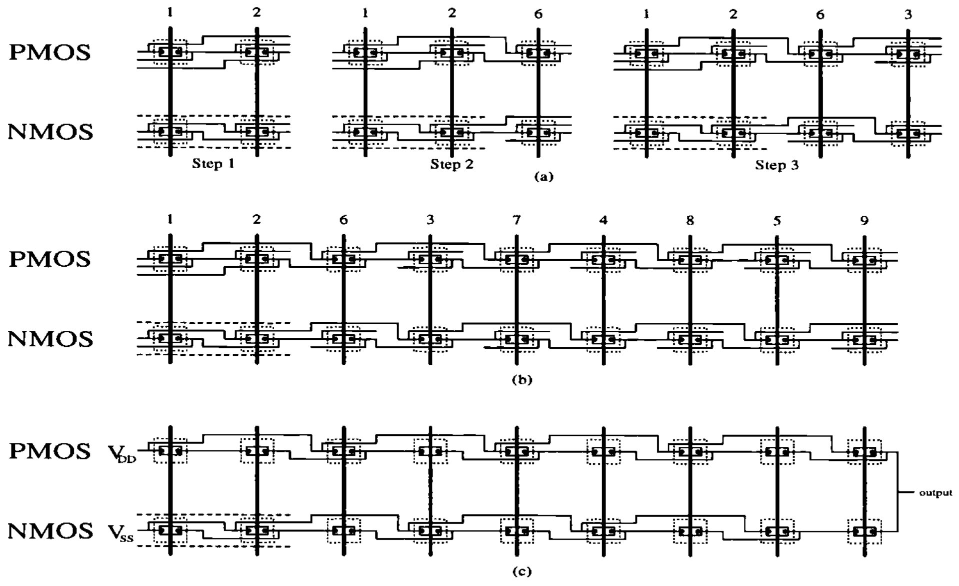


FIG. 6.5. (a) *The first three steps of the reordering and layout algorithm.* (b) *The layout after the S-P trees are traversed.* (c) *The layout after redundant wires are removed.*

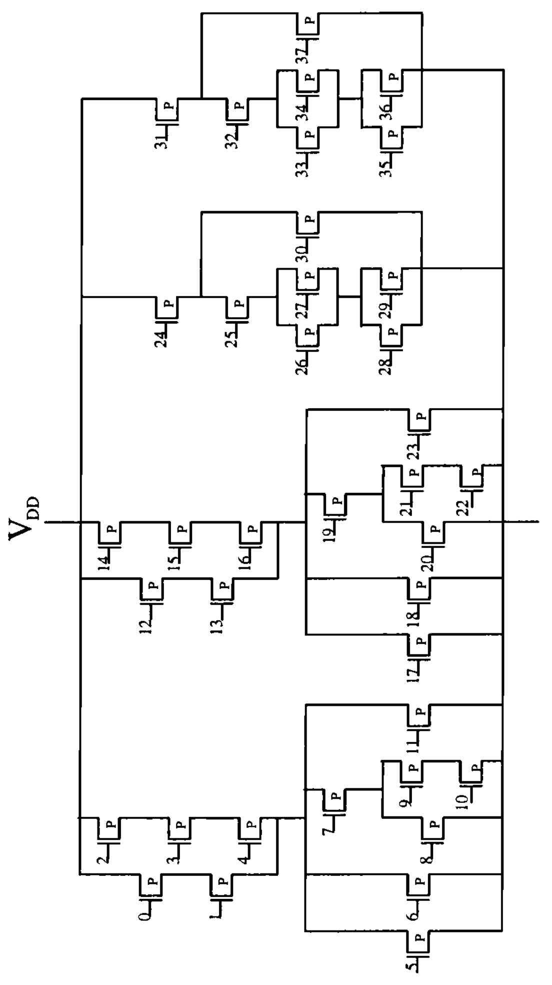


FIG. 6.6. *An example from* [12]. *The PMOS circuit is shown.*

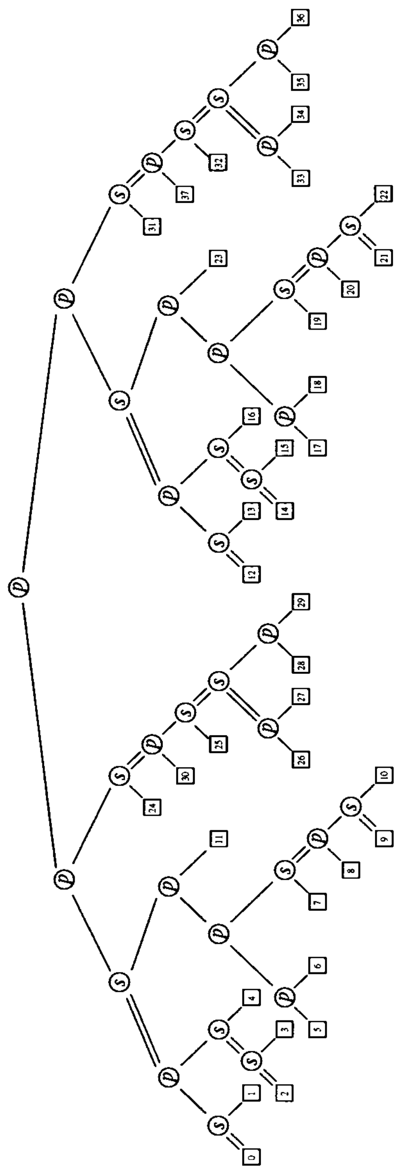


FIG. 6.7. *An S-P tree of the PMOS circuit in the example from* [12].

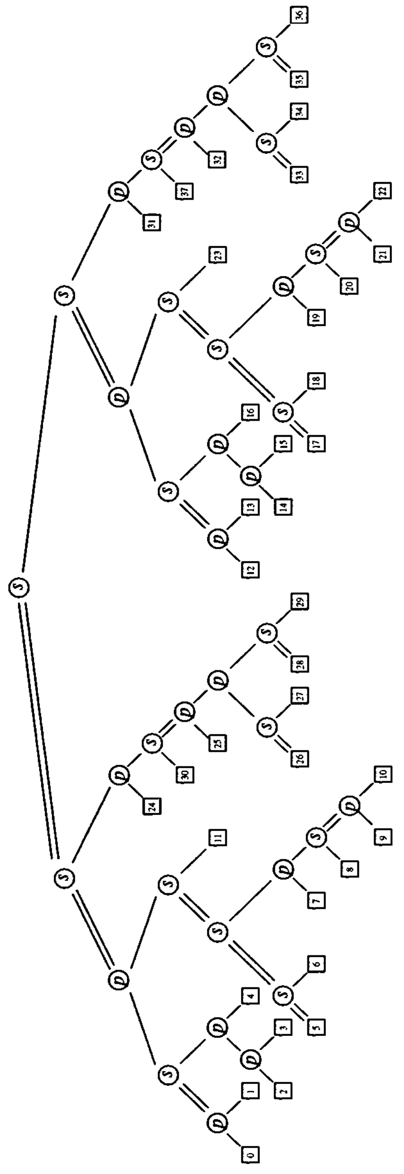


Fig. 6.8. *An S-P tree of the NMOS circuit in the example from* [12].

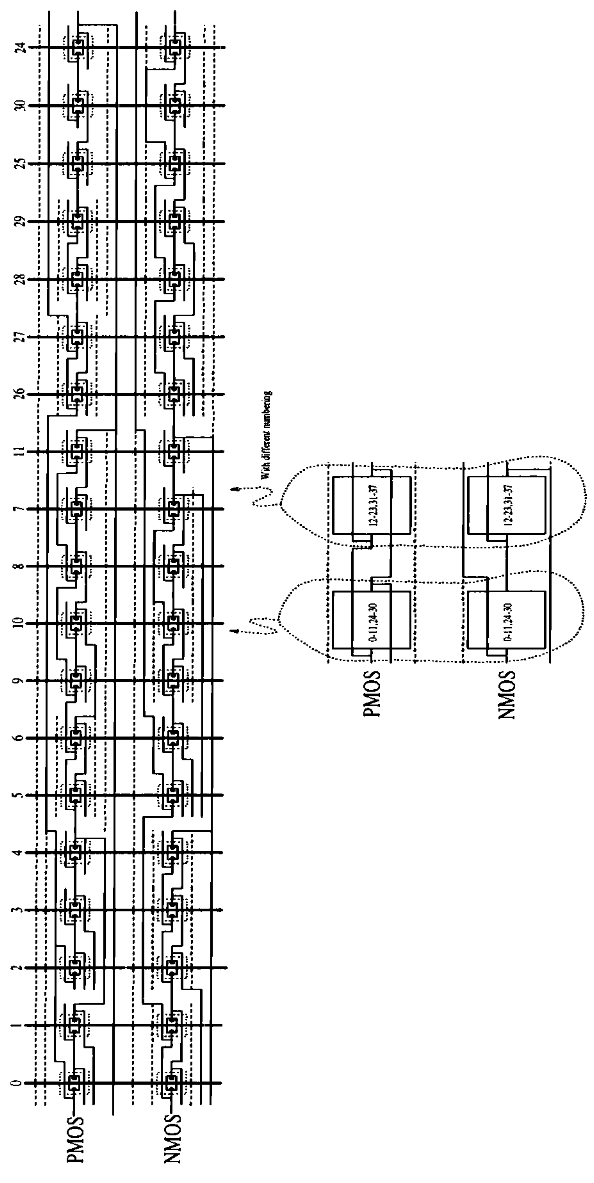


FIG. 6.9. *Our layout for the example from* [12]. *The transistors are numbered. Due to space limitations, only half of the layout details are shown. The other half has a similar layout but with different numbering.*

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990, pp. 485–488.

[2] N. Deo, M. S. Krishnamoorthy, and M. A. Langston, *Exact and approximate solutions for the gate matrix layout problem*, IEEE Trans. Computer Aided Design, CAD-6 (1987), pp. 79–84.

[3] F. Gavril, *Some NP-complete problems on graphs*, in Proceedings of the 3rd Conference on Information Sciences and Systems, The Johns Hopkins University, Baltimore, MD, 1995, pp. 91–95.

[4] E. M. Gurari and I. H. Sudborough, *Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem*, J. Algorithms, 5 (1984), pp. 531–546.

[5] Y. H. Hu and S. J. Chen, *GM-plan: A gate matrix layout algorithm based on artificial intelligence planning techniques*, IEEE Trans. Computer Aided Design, CAD-9 (1990), pp. 836–845.

[6] D. K. Hwang, W. K. Fuchs, and S. M. Kang, *An efficient approach to gate matrix layout*, IEEE Trans. Computer Aided Design, CAD-6 (1987), pp. 802–809.

[7] S. Huang and O. Wing, *Improved gate matrix layout*, IEEE Trans. Computer Aided Design, CAD-8 (1989), pp. 875–889.

[8] T. Kashiwabara and T. Fujisawa, *An NP-complete problem on interval graph*, in Proceedings of the IEEE International Symposium on Circuits and Systems, IEEE, Japan, 1979, pp. 82–83.

[9] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, New York, 1990, pp. 534–551, Chapter 9.

[10] A. D. Lopez and H. S. Law, *A dense gate matrix layout method for MOS VLSI*, IEEE Trans. Electronic Devices, ED-27 (1980), pp. 1671–1675.

[11] J. B. Sidney, *Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs*, Oper. Res., 23 (1975), pp. 283–298.

[12] C. C. Su and M. Sarrafzadeh, *Optimal gate-matrix layout of CMOS functional cells*, INTEGRATION: VLSI J., 10 (1990), pp. 3–23.

[13] K. Takamizawa, T. Nishizeki, and N. Saito, *Linear-time Computability of combinatorial problems on series-parallel graphs*, J. ACM, 29 (1982), pp. 623–641.

[14] T. Uehara and W. M. VanCleemput, *Optimal layout of CMOS functional arrays*, IEEE Trans. Comput., C-30 (1981), pp. 305–312.

[15] M. Yannakakis, *A polynomial algorithm for the min cut linear arrangement of trees*, in Annual Symposium on Foundations of Computer Science, ACM, Baltimore, MD, 1983, pp. 274–281.

[16] T. Yoshimura, *An efficient channel router*, in Proceedings of the 21st IEEE/ACM Conference on Design Automation, ACM, New York, NY, 1984, pp. 38–44.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1290–1303

APPLICATION-CONTROLLED PAGING FOR A SHARED CACHE*

RAKESH D. BARVE†, EDWARD F. GROVE‡, AND JEFFREY SCOTT VITTER§

Abstract. We propose a provably efficient application-controlled global strategy for organizing a cache of size k shared among P application processes. Each application has access to information about its own future page requests, and by using that local information along with randomization in the context of a global caching algorithm, we are able to break through the conventional $H_k \sim \ln k$ lower bound on the competitive ratio for the caching problem. If the P application processes always make good cache replacement decisions, our online application-controlled caching algorithm attains a competitive ratio of $2H_{P-1} + 2 \sim 2 \ln P$. Typically, P is much smaller than k, perhaps by several orders of magnitude. Our competitive ratio improves upon the $2P + 2$ competitive ratio achieved by the deterministic application-controlled strategy of Cao, Felten, and Li. We show that no online application-controlled algorithm can have a competitive ratio better than $\min\{H_{P-1}, H_k\}$, even if each application process has perfect knowledge of its individual page request sequence. Our results are with respect to a worst-case interleaving of the individual page request sequences of the P application processes.

We introduce a notion of fairness in the more realistic situation when application processes do not always make good cache replacement decisions. We show that our algorithm ensures that no application process needs to evict one of its cached pages to service some page fault caused by a mistake of some other application. Our algorithm not only is fair but remains efficient; the global paging performance can be bounded in terms of the number of mistakes that application processes make.

Key words. caching, application-controlled, competitive, online, randomized

AMS subject classifications. 68N25, 68P01, 68P15, 68Q25, 68W20

PII. S0097539797324278

1. Introduction.

Caching is a useful technique for obtaining high performance in these days where the latency of disk access is relatively high. Today's computers typically have several application processes running concurrently on them, by means of time sharing and multiple processors. Some processes have special knowledge of their future access patterns. Cao, Felten, and Li [CFL94a, CFL94b] exploit this special knowledge to develop effective file caching strategies.

An application providing specific information about its future needs is equivalent to the application having its own caching strategy for managing its own pages in cache. We consider the *multiapplication caching* problem, formally defined in section 3, in which P concurrently executing application processes share a common cache of size k. In section 4 we propose an online application-controlled caching scheme in which decisions need to be taken at two levels: when a page needs to be evicted from cache,

*Received by the editors July 14, 1997; accepted for publication (in revised form) September 10, 1998; published electronically February 18, 2000. An extended abstract of the results in this paper appeared in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 1995, pp. 204–213.

http://www.siam.org/journals/sicomp/29-4/32427.html

†Department of Computer Science, Duke University, Durham, NC 27708-0129 (rbarve@cs.duke.edu). This author was supported in part by an IBM graduate fellowship.

‡Department of Computer Science, Duke University, Durham, NC 27708-0129 (efg@cs.duke.edu). This author was supported in part by Army Research Office grant DAAH04-93-G-0076.

§Department of Computer Science, Duke University, Durham, NC 27708-0129 (jsv@cs.duke.edu). This author was supported in part by the National Science Foundation under grants CCR-9007851 and CCR-9522047 and by Air Force Office of Scientific Research grants F49620-92-J-0515 and F49620-94-1-0217. Part of this work was done while the author was visiting Lucent Technologies, Bell Laboratories, Murray Hill, NJ.

the global strategy chooses a victim process, but the process itself decides which of its pages will be evicted from cache.

Each application process may use any available information about its future page requests when deciding which of its pages to evict. However, we assume *no global information* about the interleaving of the individual page request sequences; all our bounds are with respect to a *worst-case interleaving* of the individual request sequences.

Competitive ratios smaller than the H_k lower bound for classical caching [FKL$^+$91] are possible for multiapplication caching, because each application may employ future information about its individual page request sequence.[1] The deterministic application-controlled algorithm proposed by Cao, Felten, and Li [CFL94a] achieves a competitive ratio of $2P+2$, which we prove in the appendix. We show in sections 5–7 that our new randomized online application-controlled caching algorithm improves the competitive ratio to $2H_{P-1}+2 \sim 2\ln P$, which is optimal up to a factor of 2 in the realistic scenario when $P < k$. (If we use the algorithm of [FKL$^+$91] for the case $P \geq k$, the resulting bound is optimal up to a factor of 2 for all P.) Our results are significant since P is often *much* smaller than k, perhaps by several orders of magnitude.

In the appendix, we also prove that no deterministic online application-controlled algorithm can have a competitive ratio better than $P+1$. Thus the difference between our competitive ratio and that attained by the algorithm by Cao, Felten, and Li is a further indication of the (well-known) power of randomization while solving online problems in general and online paging in particular.

In the scenario where application processes occasionally make bad page replacement decisions (or "mistakes"), we show in section 8 that our online algorithm incurs very few page faults globally as a function of the number of mistakes. Our algorithm is also *fair*, in the sense that the mistakes made by one processor in its page replacement decisions do not worsen the page fault rate of other processors.

2. Classical caching and competitive analysis.

The well-known *classical caching (or paging) problem* deals with a two-level memory hierarchy consisting of a fast *cache* of size k and *slow memory* of arbitrary size. A sequence of requests to pages is to be satisfied in their order of occurrence. In order to satisfy a page request, the page must be in fast memory. When a requested page is not in fast memory, a *page fault* occurs, and some page must be evicted from fast memory to slow memory in order to make room for the new page to be put into fast memory. The caching (or paging) problem is to decide which page must be evicted from the cache. The cost to be minimized is the number of page faults incurred over the course of servicing the page requests.

Belady [Bel66] gives a simple optimum offline algorithm for the caching problem; the page chosen for eviction is the one in cache whose next request is furthest in the future. In order to quantify the performance of an online algorithm, Sleator and Tarjan [ST85] introduce the notion of competitiveness, which in the context of caching can be defined as follows: for a caching algorithm A, let $F_A(\sigma)$ be the number of page faults generated by A while processing page request sequence σ. If A is a randomized algorithm, we let $F_A(\sigma)$ be the *expected* number of page faults generated by A on processing σ, where the expectation is with respect to the random choices made by the algorithm. An online algorithm A is called *c-competitive* if for every

[1] Here H_n represents the nth harmonic number $\sum_{i=1}^{n} 1/i \sim \ln n$.

page request sequence σ, we have $F_A(\sigma) \leq c \cdot F_{OPT}(\sigma) + b$, where b is some fixed constant. The constant c is called the *competitive ratio* of A. Under this measure, an online algorithm's performance needs to be relatively good on worst-case page request sequences in order for the algorithm to be considered good.

For cache size k, Sleator and Tarjan [ST85] show a lower bound of k on the competitive ratio of deterministic caching algorithms. Fiat et al. [FKL$^+$91] prove a lower bound of H_k if randomized algorithms are allowed. They also give a simple and elegant randomized algorithm for the problem that achieves a competitive ratio of $2H_k$. McGeoch and Sleator [MS91] give a rather involved randomized algorithm that attains the theoretically optimal competitive ratio of H_k.

3. Multiapplication caching problem.

In this paper we take up the theoretical issue of how best to use application processes' knowledge about their individual future page requests so as to optimize caching performance. For analysis purposes we use an online framework similar to that of [FKL$^+$91, MS91]. As mentioned before, the caching algorithms in [FKL$^+$91, MS91] use *absolutely no information* about future page requests. Intuitively, knowledge about future page requests can be exploited to decide which page to evict from the cache at the time of a page fault. In practice an application often has advance knowledge of its individual future page requests. Cao, Felten, and Li [CFL94a, CFL94b] introduced strategies that try to combine the advance knowledge of the processors in order to make intelligent page replacement decisions.

In the *multiapplication caching problem* we consider a cache capable of storing k pages that is shared by P different application processes, which we denote $P_1, P_2, \ldots, P_P$. Each page in cache and memory belongs to exactly one process. The individual request sequences of the processes may be interleaved in an arbitrary (worst-case) manner.

Worst-case measure is often criticized when used for evaluating caching algorithms for individual application request sequences [BIRS91, KPR92], but we feel that the worst-case measure is appropriate for considering a global paging strategy for a cache shared by concurrent application processes that have knowledge of their individual page request sequences. The locality of reference within each application's individual request sequence is accounted for in our model by each application process's knowledge of its own future requests. The worst-case nature of our model is that it assumes nothing about the order and duration of time for which application processes are active. In this model our worst-case measure of competitive performance amounts to considering a *worst-case interleaving* of individual sequences.

The approach of Cao, Felten, and Li [CFL94a] is to have the kernel deterministically choose the process owning the least recently used page at the time of a page fault and ask that process to evict a page of its choice (which may be different from the least recently used (LRU) page). In the appendix we show under the assumption that processes always make good page replacement decisions that Cao, Felten, and Li's algorithm has a competitive ratio between $P + 1$ and $2P + 2$. The algorithm we present in the next section and analyze thereafter improves the competitive ratio to $2H_{P-1} + 2 \sim 2 \ln P$.

4. Online algorithm for multiapplication caching.

Our algorithm is an online application-controlled caching strategy for an operating system kernel to manage a shared cache in an efficient and fair manner. We show in the subsequent sections that the competitive ratio of our algorithm is $2H_{P-1}+2 \sim 2 \ln P$ and that it is optimal to within a factor of about 2 among all online algorithms. (If $P \geq k$, we can use the algorithm of [FKL$^+$91].)

On a page fault, we first choose a victim process and then ask it to evict a suitable page. Our algorithm can detect mistakes made by application processes, which enables us to reprimand such application processes by having them pay for their mistakes. In our scheme, we mark pages as well as processes in a systematic way while processing the requests that constitute a phase.

DEFINITION 4.1. *The global sequence of page requests is partitioned into a consecutive sequence of* phases*; each phase is a sequence of page requests. At the beginning of each phase, all pages and processes are unmarked. A page gets marked during a phase when it is requested. A process is marked when all of its pages in cache are marked. A new phase begins when a page is requested that is not in cache and all the pages in cache are marked. A page accessed during a phase is called* clean *with respect to that phase if it was not in the online algorithm's cache at the beginning of a phase. A request to a clean page is called a* clean page request*. Each phase always begins with a clean page request.*

Our marking scheme is similar to the one in [FKL$^+$91] for the classical caching problem. However, unlike the algorithm in [FKL$^+$91], the algorithm we develop is a nonmarking algorithm, in the sense that our algorithm may evict marked pages. In addition, our notion of phase in Definintion 4.1 is different from the notion of phase in [FKL$^+$91], which can be looked upon as a special case of our more general notion. We put the differences into perspective in section 4.1.

Our algorithm works as follows when a page p belonging to process P_r is requested:

```
(1) If $p$ is in cache:
    (a) If $p$ is not marked, we mark it.
    (b) If process $P_r$ has no unmarked pages in cache, we mark $P_r$.
(2) If $p$ is not in cache:
    (a) If process $P_r$ is unmarked and page $p$ is *not* a clean page with respect
        to the ongoing phase (i.e., $P_r$ has made a mistake earlier in the phase
        by evicting $p$), then
        (i) We ask process $P_r$ to make a page replacement decision and evict
            one of its pages from cache in order to bring page $p$ into cache. We
            mark page $p$ and also mark process $P_r$ if it now has no unmarked
            pages in cache.
    (b) Else (process $P_r$ is marked or page $p$ is a clean page, or both)
        (i) If all pages in cache are marked, we remove marks from all pages
            and processes, and we start a new phase, beginning with the current
            request for $p$.
        (ii) Let $S$ denote the set of unmarked processes having pages in the
            cache. We randomly choose a process $P_e$ from $S$, each process being
            chosen with a uniform probability $1/|S|$.
        (iii) We ask process $P_e$ to make a page replacement decision and evict
            one of its pages from cache in order to bring page $p$ into cache. We
            mark page $p$ and also mark process $P_e$ if it now has no unmarked
            page in cache.
```

Note that in steps 2(a)(i) and 2(b)(iii) our algorithm seeks paging decisions from application processes that are unmarked. Consider an unmarked process P_i that has been asked to evict a page in a phase, and consider P_i's pages in cache at that time. Let u_i denote the *farthest unmarked* page of process P_i; that is, u_i is the unmarked page of process P_i whose next request occurs furthest in the future among all of P_i's unmarked cached pages. Note that process P_i may have marked pages in cache whose next requests occur after the request for u_i.

DEFINITION 4.2. *The* good set *of an unmarked process P_i at the current point in the phase is the set consisting of its farthest unmarked page u_i in cache and every marked page of P_i in cache whose next request occurs* after *the next request for page u_i. A page replacement decision made by an unmarked process P_i in either step* 2(a)(i) *or step* 2(b)(iii) *that evicts a page from its good set is regarded as a* good decision *with respect to the ongoing phase. Any page from the good set of P_i is a* good page *for eviction purposes at the time of the decision. Any decision made by an unmarked process P_i that is not a good decision is regarded as a* mistake *by process P_i.*

If a process P_i makes a mistake by evicting a certain page from cache, we can detect the mistake made by P_i if and when the same page is requested again by P_i in the same phase while P_i is still unmarked.

In sections 6 and 7 we specifically assume that application processes are always able to make good decisions about page replacement. In section 8 we consider fairness properties of our algorithm in the more realistic scenario where processes can make mistakes.

4.1. Relation to previous work on classical caching. Our marking scheme approach is inspired by a similar approach for the classical caching problem in [FKL$^+$91]. However, the phases defined by our algorithm are significantly different in nature from those in [FKL$^+$91]. Our phase ends when there are k distinct marked pages in cache; more than k distinct pages may be requested in the phase. The phases depend on the random choices made by the algorithm and are probabilistic in nature. On the other hand, a phase defined in [FKL$^+$91] ends when exactly k distinct pages have been accessed, so that given the input request sequence, the phases can be determined independently of the caching algorithm being used.

The definition in [FKL$^+$91] is suited to facilitate the analysis of online caching algorithms that never evict marked pages, called *marking algorithms.* In the case of marking algorithms, since marked pages are never evicted, as soon as k distinct pages are requested, there are k distinct marked pages *in* cache. This means that the phases determined by our definition for the special case of marking algorithms are exactly the same as the phases determined by the definition in [FKL$^+$91]. Note that our algorithm is in general *not* a marking algorithm since it may evict marked pages. While marking algorithms always evict unmarked pages, our algorithm always calls on unmarked processes to evict pages; the actual pages evicted may be marked.

4.2. Relation to the algorithm of Cao, Felten, and Li. The algorithm proposed by Cao, Felten, and Li [CFL94a] for the multiapplication caching problem amounts to evicting, at the time of a page fault, the farthest page from cache belonging to the process that owns the LRU page in cache. Thus, for a given interleaving of individual request sequences, the paging decisions made by that algorithm are deterministic. We prove in the appendix that no deterministic online algorithm for the multiapplication caching problem can have a competitive ratio better than $P+1$, and that the competitive ratio attained by the algorithm proposed by Cao, Felten, and Li [CFL94a] attains a competitive ratio of $2P+2$. Thus, the performance of the algorithm in [CFL94a] is within a factor of 2 of the best possible performance by any deterministic online algorithm for the multiapplication caching problem.

Given the notion of good pages that we developed above, it turns out that we can define a slightly more general version of the algorithm in [CFL94a] without changing its paging performance. Basically, in order to attain the competitive ratio of $2P + 2$, it is enough, at the time of a page fault, to evict from cache *any good page* belonging to the process that owns the LRU page in cache. We say that this is a slightly more general version than the algorithm presented in [CFL94a] because it may very often be the case that the good set of the process that owns the LRU page contains several pages other than the farthest page of that process.

5. Lower bounds for *OPT* and competitive ratio.

In this section we prove that the competitive ratio of any online caching algorithm can be no better than $\min\{H_{P-1}, H_k\}$. Let us denote by OPT the optimal offline algorithm for caching that works as follows: when a page fault occurs, OPT evicts the page whose next request is furthest in the future request sequence among all pages in cache.

As in [FKL$^+$91], we will compare the number of page faults generated by our online algorithm during a phase with the number of page faults generated by OPT during that phase. We express the number of page fronts as a function of the number of clean page requests during the phase. Here we state and prove a lower bound on the (amortized) number of page faults generated by OPT in a single phase. The proof is a simple generalization of an analogous proof in [FKL$^+$91], which deals only with the deterministic phases of marking algorithms.

LEMMA 5.1. *Consider any phase σ_i of our online algorithm in which ℓ_i clean pages are requested. Then OPT incurs an amortized cost of at least $\ell_i/2$ on the requests made in that phase.*[2]

Proof. Let d_i be the number of clean pages in OPT's cache at the beginning of phase σ_i; that is, d_i is the number of pages requested in σ_i that are in OPT's cache but not in our algorithm's cache at the beginning of σ_i. Let d_{i+1} represent the same quantity for the next phase σ_{i+1}. Let $d_{i+1} = d_m + d_u$, where d_m of the d_{i+1} clean pages in OPT's cache at the beginning of σ_{i+1} are marked during σ_i and d_u of them are not marked during σ_i. Note that $d_1 = 0$ and $d_i \le k$ for all i.

Of the ℓ_i clean pages requested during σ_i, only d_i are in OPT's cache, so OPT generates at least $\ell_i - d_i$ page faults during σ_i. On the other hand, while processing the requests in σ_i, OPT cannot use d_u of the cache locations, since at the beginning of σ_{i+1} there are d_u pages in OPT's cache that are not marked during σ_i. (These d_u pages would have to be in OPT's cache before σ_i even began.) There are k marked pages in our algorithm's cache at the end of σ_i, and there are d_m other pages marked during σ_i that are out of our algorithm's cache. So the number of distinct pages requested during σ_i is at least $d_m + k$. Hence, OPT serves at least $d_m + k$ requests corresponding to σ_i without using d_u of the cache locations. This means that OPT generates at least $(k + d_m) - (k - d_u) = d_{i+1}$ faults during σ_i. Therefore, the number of faults OPT generates on σ_i is at least

$$\max\{\ell_i - d_i, d_{i+1}\} \ge \frac{\ell_i - d_i + d_{i+1}}{2}. \tag{5.1}$$

Let us consider the number of page faults made by OPT in the first j phases of page request sequence σ. We can use the lower bound (5.1) for phases $2, 3, \dots, j-1$. In the jth phase, OPT makes at least $\ell_j - d_j \ge (\ell_j - d_j)/2$ faults. In the first phase, OPT generates k faults and we have $\ell_1 = k$. Thus the sum of OPT's faults over all

[2]By "amortized" in Lemma 5.1 we mean for each $j \ge 1$ that the number of page faults made by OPT while serving the first j phases is at least $\sum_{i=1}^{j} \ell_i/2$, where ℓ_i is the number of clean page requests in the ith phase.

j phases is at least

$$\ell_1 + \sum_{i=2}^{j-1} \frac{\ell_i - d_i + d_{i+1}}{2} + \frac{\ell_j - d_j}{2} \geq \sum_{i=1}^{j} \ell_i/2,$$

where we use the fact that $d_2 \leq k = \ell_1$. Thus by definition, the amortized number of faults OPT generates over any phase σ_i is at least $\ell_i/2$. ∎

Next we will construct a lower bound for the competitive ratio of *any* randomized online algorithm even when application processes have perfect knowledge of their individual request sequences. The proof is a straightforward adaptation of the proof of the H_k lower bound for classical caching [FKL+91]. However, in the situation at hand, the adversary has more restrictions on the request sequence that he can use to prove the lower bound, thereby resulting in a lowering of the lower bound.

Theorem 5.2. *The competitive ratio of any randomized algorithm for the multi-application caching problem is at least* $\min\{H_{P-1}, H_k\}$ *even if application processes have perfect knowledge of their individual request sequences.*

Proof. If $P > k$, the H_k lower bound on the classical caching problem from [FKL+91] is directly applicable by considering the case where each process accesses only one page each. This gives a lower bound of H_k on the competitive ratio.

In the case when $P \leq k$, we construct a multiapplication caching problem based on the nemesis sequence used in [FKL+91] for classical caching. In [FKL+91] a lower bound of $H_{k'}$ is proved for the special case of a cache of size k' and a total of $k'+1$ pages, which we denote $c_1, c_2, \ldots, c_{k'+1}$. All but one of the pages can fit in cache at the same time. Our corresponding multiapplication caching problem consists of $P = k'+1$ application processes $P_1, P_2, \ldots, P_P$ so that there is one process corresponding to each page of the classical caching lower bound instance for a k'-sized cache. Process P_i owns r_i pages $p_{i1}, p_{i2}, \ldots, p_{ir_i}$. The total number $\sum_{i=1}^{P} r_i$ of pages among all the processes is $k+1$, where k is the cache size; that is, all but one of the pages among all the processes can fit in memory simultaneously.

In the instance of the multiapplication caching problem we construct, the request sequence for each process P_i consists of repetitions of the double round-robin sequence

$$p_{i1}, p_{i2}, \ldots, p_{ir_i}, p_{i1}, p_{i2}, \ldots, p_{ir_i} \tag{5.2}$$

of length $2r_i$. We refer to the double round-robin sequence (5.2) as a *touch* of process P_i. When the adversary generates requests corresponding to a touch of process P_i, we say that it "touches process P_i."

Given an arbitrary adversarial sequence for the classical caching problem described above, we construct an adversarial sequence for the multiapplication caching problem by replacing each request for page c_i in the former problem by a touch of process P_i in the latter problem. We can transform an algorithm for this instance of multiapplication caching into one for the classical caching problem by the following correspondence: if the multiapplication algorithm evicts a page from process P_j while servicing the touch of process P_i, the classical caching algorithm evicts page c_j in order to service the request to page c_i. In Lemma 5.3 below, we show that there is an optimum online algorithm for the above instance of multiapplication caching that never evicts a page belonging to process P_i while servicing a fault on a request for a page from process P_i. Thus the transformation is valid, in that page c_i is always resident in cache after the page request to c_i is serviced. This reduction immediately implies that the competitive ratio for this instance of multiapplication caching must be at least $H_{k'} = H_{P-1}$. ∎

LEMMA 5.3. *For the above instance of multiapplication caching, any online algorithm A can be converted into an online algorithm A' that is at least as good in an amortized sense and that has the property that all the pages for process P_i are in cache immediately after a touch of P_i is processed.*

Proof. Intuitively, the double round-robin sequences force an optimal online algorithm to service the touch of a process by evicting a page belonging to *another* process. We construct online algorithm A' from A in an online manner. Suppose that both A and A' fault during a touch of process P_i. If algorithm A evicts a page of P_j, for some $j \neq i$, then A' does the same. If algorithm A evicts a page of P_i during the first round-robin while servicing a touch of P_i, then there will be a page fault during the second round-robin. If A then evicts a page of another process during the second round-robin, then A' evicts that page during the first round-robin and incurs no fault during the second round-robin. The first page fault of A was wasted; the other page could have been evicted instead during the first round-robin. If instead A evicts another page of P_i during the second round-robin, then A' evicts an arbitrary page of another process during the first round-robin, and A' incurs no page fault during the second round-robin. Thus, if A evicts a page of P_i, it incurs at least one more page fault than does A'.

If A faults during a touch of P_i, but A' doesn't, there is no paging decision for A' to make. If A does not fault during a touch of P_i, but A' does fault, then A' evicts the page that is not in A's cache. The page fault for A' is charged to the extra page fault that A incurred earlier when A' evicted one of P_i's pages.

Thus the number of page faults that A' incurs is no more than the number of page faults that A incurs. By construction, all pages of process P_i are in algorithm A''s cache immediately after a touch of process P_i. ◻

The double round-robin sequences in the above reduction can be replaced by single round-robin sequences by redoing the explicit lower bound argument of [FKL$^+$91].

6. Holes.

In this section, we introduce the notion of holes, which plays a key role in the analysis of our online caching algorithm. In section 6.2, we mention some crucial properties of holes of our algorithm under the assumption that applications always make good page replacement decisions. These properties are also useful in bounding the page faults that can occur in a phase when applications make mistakes in their page replacement decisions.

DEFINITION 6.1. *The eviction of a cached page at the time of a page fault on a clean page request is said to create a* hole *at the evicted page. Intuitively, a hole is the lack of space for some page, so that that page's place in cache contains a hole and not the page. If page p_1 is evicted for servicing the clean page request, page p_1 is said to be associated with the hole. If page p_1 is subsequently requested and another page p_2 is evicted to service the request, the hole is said to* move *to p_2, and now p_2 is said to be associated with the hole, and so on, until the end of the phase. We say that* hole h moves to process P_i *to mean that the hole h moves to some page p belonging to process P_i.*

6.1. General observations about holes.

All requests to clean pages during a phase are page faults and create holes. The number of holes created during a particular phase equals the number of clean pages requested during that phase. Apart from clean page requests, requests to holes also cause page faults to occur. By a *request to a hole* we mean a request for the page associated with that hole. As we proceed down the request sequence during a phase, the page associated with a particular hole varies with time. Consider a hole h that is created at a page p_1 that is evicted to serve a request for clean page p_c. When a request is made for page p_1, some page p_2 is evicted, and h moves to p_2. Similarly when page p_2 is requested, h moves to some p_3, and so on. Let $p_1, p_2, \ldots, p_m$ be the temporal sequence of pages all associated with hole h in a particular phase such that page p_1 is evicted when clean page p_c is requested, page p_i, where $i > 1$, is evicted when p_{i-1} is requested and the request for p_m falls in the next phase. Then the number of faults incurred in the particular phase being considered due to requests to h is $m - 1$.

6.2. Useful properties of holes. In this section we make the following observations about holes under the assumption that application processes make only good decisions.

LEMMA 6.2. *Let u_i be the farthest unmarked page in cache of process P_i at some point in a phase. Then process P_i is a marked process by the time the request for page u_i is served.*

Proof. This follows from the definition of farthest unmarked page and the nature of the marking scheme employed in our algorithm. □

LEMMA 6.3. *Suppose that there is a request for page p_i, which is associated with hole h. Suppose that process P_i owns page p_i. Then process P_i is already marked at the time of the present request for page p_i.*

Proof. Page p_i is associated with hole h because process P_i evicted page p_i when asked to make a page replacement decision in order to serve either a clean request or a page fault at the previous page associated with h. In either case, page p_i was a good page at the time process P_i made the particular paging decision. Since process P_i was unmarked at the time the decision was made, p_i was either the farthest unmarked page of process P_i then or some marked page of process P_i whose next request is after the request for P_i's farthest unmarked page. By Lemma 6.2, process P_i is a marked process at the time of the request for page p_i. □

LEMMA 6.4. *Suppose that page p_i is associated with hole h. Let P_i denote the process owning page p_i. Suppose page p_i is requested at some time during the phase. Then hole h does not move to process P_i subsequently during the current phase.*

Proof. The hole h belongs to process P_i. By Lemma 6.3 when a request is made to h, P_i is already marked and will remain marked until the end of the phase. Since only unmarked processes are chosen to evict pages, a request for h thereafter cannot result in eviction of any page belonging to P_i, so a hole can never move to a process more than once. □

Let there be R unmarked processes at the time of a request to a hole h. For any unmarked process P_j, $1 \leq j \leq R$, let u_j denote the farthest unmarked page of process P_j at the time of the request to hole h. Without loss of generality, let us relabel the processes so that

$$u_1, u_2, u_3, \ldots, u_R \tag{6.1}$$

is the temporal order of the first subsequent appearance of the pages u_j in the global page request sequence.

LEMMA 6.5. *In the situation described in* (6.1) *above, suppose during the page request for hole h that the hole moves to a good page p_i of unmarked process P_i to serve the current request for h. Then h can never move to any of the processes $P_1, P_2, \ldots, P_{i-1}$ during the current phase.*

Proof. The first subsequent request for the good page p_i that P_i evicts, by definition, must be the same as or must be after the first subsequent request for the farthest unmarked page u_i. So process P_i will be marked by the next time hole h is requested, by Lemma 6.3. On the other hand, the first subsequent requests of the respective farthest unmarked pages $u_1, \ldots, u_{i-1}$ appear before that of page u_i. Thus, by Lemma 6.2, the processes $P_1, P_2, \ldots, P_{i-1}$ are already marked before the next time hole h (page p_i) gets requested and will remain marked for the remainder of the phase. Hence, by the fact that only unmarked processes get chosen, hole h can never move to any of the processes $P_1, P_2, \ldots, P_{i-1}$. ☐

7. Competitive analysis of our online algorithm.

Our main result is Theorem 7.1, which states that our online algorithm for the multiapplication caching problem is roughly $2 \ln P$-competitive, assuming application processes always make good decisions (e.g., if each process knows its own future page requests). By the lower bound of Theorem 5.2, it follows that our algorithm is optimal in terms of competitive ratio up to a factor of 2.

THEOREM 7.1. *The competitive ratio of our online algorithm in section* 4 *for the multiapplication caching problem, assuming that good evictions are always made, is at most* $2H_{P-1} + 2$. *Our competitive ratio is within a factor of about* 2 *of the best possible competitive ratio for this problem.*

The rest of this section is devoted to proving Theorem 7.1. To count the number of faults generated by our algorithm in a phase, we make use of the properties of holes from the previous section. If ℓ requests are made to clean pages during a phase, there are ℓ holes that move about during the phase. We can count the number of faults generated by our algorithm during the phase as

$$\ell + \sum_{i=1}^{\ell} N_i, \tag{7.1}$$

where N_i is the number of times hole h_i is requested during the phase. Assuming good decisions are always made, we will now prove for each phase and for any hole h_i that the expected value of N_i is bounded by H_{P-1}.

Consider the first request to a hole h during the phase. Let R_h be the number of unmarked processes at that point in time. Let C_{R_h} be the random variable associated with the number of page faults due to requests to hole h during the phase.

LEMMA 7.2. *The expected number* $E(C_{R_h})$ *of page faults due to requests to hole h is at most* H_{R_h}.

Proof. We prove this by induction over R_h. We have $E(C_0) = 0$ and $E(C_1) = 1$. Suppose for $0 \leq j \leq R_h - 1$ that $E(C_j) \leq H_j$. Using the same terminology and notation as in Lemma 6.5, we let the farthest unmarked pages of the R_h unmarked processes at the time of the request for h appear in the temporal order

$$u_1, u_2, u_3, \ldots, u_{R_h}$$

in the global request sequence. We renumber the R_h unmarked processes for convenience so that page u_i is the farthest unmarked page of unmarked process P_i.

When the hole h is requested, our algorithm randomly chooses one of the R_h unmarked processes, say, process P_i, and asks process P_i to evict a suitable page. Under our assumption, the hole h moves to some good page p_i of process P_i. From Lemmas 6.4 and 6.5, if our algorithm chooses unmarked process P_i so that its good page p_i is evicted, then *at most* $R_h - i$ *processes remain unmarked* the next time h is requested. Since each of the R_h unmarked processes is chosen with a probability of $1/R_h$, we have

$$\begin{aligned} E(C_{R_h}) &\leq 1 + \frac{1}{R_h} \sum_{i=1}^{R_h} E(C_{R_h - i}) \\ &= 1 + \frac{1}{R_h} \sum_{i=0}^{R_h - 1} E(C_i) \\ &\leq 1 + \frac{1}{R_h} \sum_{i=0}^{R_h - 1} H_i \\ &= H_{R_h}. \end{aligned}$$

The last equality follows easily by induction and algebraic manipulations. □

Now let us complete the proof of Theorem 7.1. By Lemma 6.3 the maximum possible number of unmarked processes at the time a hole h is first requested is $P-1$. Lemma 7.2 implies that the average number of times any hole can be requested during a phase is bounded by H_{P-1}. By (7.1), the total number of page faults during the phase is at most $\ell(1 + H_{P-1})$. We have already shown in Lemma 5.1 that the *OPT* algorithm incurs an amortized cost of at least $\ell/2$ for the requests made in the phase. Therefore, the competitive ratio of our algorithm is bounded by $\ell(1+H_{P-1})/(\ell/2) = 2H_{P-1}+2$. Applying the lower bound of Theorem 5.2 completes the proof.

8. Application-controlled caching with fairness. In this section we analyze our algorithm's performance in the realistic scenario where application processes can make mistakes, as defined in Definition 4.2. We bound the number of page faults it incurs in a phase in terms of page faults caused by mistakes made by application processes during that phase. The main idea here is that if an application process P_i commits a mistake by evicting a certain page p and then during the same phase requests page p while process P_i is still unmarked, our algorithm makes process P_i pay for the mistake in step 2(a)(i).

On the other hand, if page p's eviction from process P_i was a mistake, but process P_i is marked when page p is later requested in the same phase, say, at time t, then process P_i's mistake is "not worth detecting" for the following reason: since evicting page p was a mistake, it must mean that at the time t_1 of p's eviction, there existed a set U of one or more unmarked pages of process P_i in cache whose subsequent requests appear *after* the next request for page p. Process P_i is marked at the time of the next request for p, implying that all pages in U were *evicted* by P_i at some times $t_2, t_3, \ldots, t_{|U|+1}$ after the mistake of evicting p. If instead at time $t_1, t_2, \ldots, t_{|U|+1}$ process P_i makes the specific good paging decisions of evicting the farthest unmarked pages, *the same set* $\{p\} \cup U$ *of pages will be out of cache at time* t. In our notion of fairness we choose to ignore all such mistakes and consider them "not worth detecting."

DEFINITION 8.1. *During an ongoing phase, any page fault corresponding to a request for a page* p *of an* unmarked *process* P_i *is called an* unfair fault *if the request for page* p *is not a clean page request. All faults during the phase that are not unfair are called* fair faults.

The unfair faults are precisely those page faults which are caused by mistakes considered "worth detecting." We state the following two lemmas that follow trivially from the definitions of mistakes, good decisions, unfair faults, and fair faults.

Lemma 8.2. *During a phase, all page requests that get processed in step* 2(a)(i) *of our algorithm are precisely the unfair faults of that phase. That is, unfair faults correspond to mistakes that get caught in step* 2(a)(i) *of our algorithm.*

Lemma 8.3. *All fair faults are precisely those requests that get processed in step* 2(b)(iii).

We now consider the behavior of holes in the current mistake-prone scenario.

Lemma 8.4. *The number of holes in a phase equals the number of clean pages requested in the phase.*

Lemma 8.5. *Consider a hole h associated with a page p of a process P_i. If a request for h is an unfair fault, process P_i is still unmarked and the hole h moves to some other page belonging to process P_i. If a request for hole h is a fair fault, then process P_i is already marked and the hole h can never move to process P_i subsequently during the phase.*

Proof. If the request for hole h is an unfair fault, then by definition process P_i is unmarked and by Lemma 8.2, h moves to some other page p' of process P_i. If the request for h is a fair fault, then by definition and the fact that the request for h is not a clean page request, process P_i is marked. Since our algorithm never chooses a marked process for eviction, it follows that h can never visit process P_i subsequently during the phase. □

During a phase, a hole h is created in some process, say, P_1, by some clean page request. It then moves around zero or more times within process P_1 on account of P_1's mistakes, until a request for hole h is a fair fault, upon which it moves to some other process P_2, never to come back to process P_1 during the phase. It behaves similarly in process P_2, and so on up to the end of the phase. Let T_h denote the total number of faults attributed to requests to hole h during a phase, of which F_h faults are fair faults and U_h faults are unfair faults. We have $T_h = F_h + U_h$.

By Lemma 8.5 and the same proof techniques as those in the proofs of Lemma 7.2 and Theorem 7.1, we can prove the following key lemma.

Lemma 8.6. *The expected number $E(F_h)$ of page requests to hole h during a phase that result in fair faults is at most H_{P-1}.*

By Lemma 8.4, our algorithm incurs at most $\ell + \sum_{i=1}^{\ell} T_{h_i}$ page faults in a phase with ℓ clean page requests. The expected value of this quantity is at most $\ell(H_{P-1} + 1) + \sum_{i=1}^{\ell} U_{h_i}$, by Lemma 8.6.

The expression $\sum_{i=1}^{\ell} U_{h_i}$ is the number of *unfair faults*, that is, the number of mistakes considered "worth detecting." Our algorithm is very efficient in that the number of unfair faults is an additive term. For any phase ϕ with ℓ clean requests, we denote $\sum_{i=1}^{\ell} U_{h_i}$ as M_ϕ.

Theorem 8.7. *The number of faults in a phase ϕ with ℓ clean page requests and M_ϕ unfair faults is bounded by $\ell(1 + H_{P-1}) + M_\phi$. At the time of each of the M_ϕ unfair faults, the application process that makes the mistake that causes the fault must evict a page from its own cache. No application process is ever asked to evict a page to service an unfair fault caused by some other application process.*

8.1. Extending fairness to the algorithm by Cao, Felten, and Li. It turns out that our notion of fairness extends, without any change, to the generalized version of the deterministic algorithm of [CFL94a] that we mentioned in section 4.2. It is easy to see that in the case of the generalized version of the algorithm of [CFL94a], a process incurs an unfair fault only if, at some time in the past, that process had the LRU page and the page it evicted was not a good page. Consequently, a result similar to Theorem 8.7 with $(1 + H_{P-1})$ replaced by $(1 + P)$ holds for the generalized version of the algorithm of [CFL94a].

9. Conclusions.

Cache management strategies are of prime importance for high performance computing. We consider the case where there are P independent processes running on the same computer system and sharing a common cache of size k. Applications often have advance knowledge of their page request sequences. In this paper we have addressed the issue of exploiting this advance knowledge to devise intelligent strategies to manage the shared cache, in a theoretical setting. We have presented a simple and elegant randomized application-controlled caching algorithm for the multiapplication caching problem that achieves a competitive ratio of $2H_{P-1}+2$. Our result is a significant improvement over the competitive ratios of $2P+2$ [CFL94a] for deterministic multiapplication caching and $\Theta(H_k)$ for classical caching, since the cache size k is often orders of magnitude greater than P. We have proven that no online algorithm for this problem can have a competitive ratio smaller than $\min\{H_{P-1}, H_k\}$, even if application processes have perfect knowledge of individual request sequences. We conjecture that an upper bound of H_{P-1} can be proven, up to second-order terms, perhaps using techniques from [MS91], although the resulting algorithm is not likely to be practical.

Using our notion of mistakes we are able to consider a more realistic setting when application processes make bad paging decisions and show that our algorithm is a fair and efficient algorithm in such a situation. No application needs to pay for some other application process's mistake, and we can bound the global caching performance of our algorithm in terms of the number of mistakes. Our notions of good page replacement decisions, mistakes, and fairness in this context are new.

One related area of possible future work is to consider alternative models to our model of worst-case interleaving. Another interesting area would be to consider caching in a situation where some applications have good knowledge of future page requests while other applications have no knowledge of future requests. We could also consider pages shared among application processes.

Appendix. The Cao, Felten, and Li algorithm.

The algorithm proposed by Cao, Felten, and Li [CFL94a] for the multiapplication caching problem amounts to evicting, at the time of a page fault, the *farthest* page from cache belonging to the process that owns the LRU page in cache.

THEOREM A.1. *The algorithm of Cao, Felten, and Li is* $(2P+2)$*-competitive. A generalized version of the algorithm of Cao, Felten, and Li, in which, at the time of a page fault, the process owning the LRU page in cache evicts any (deterministically chosen) good page, is also* $(2P+2)$*-competitive.*

Proof. Let there be ℓ clean page requests in a phase. Then there are ℓ faults due to clean page requests resulting in ℓ holes. The algorithm evicts only *good* pages from cache, so holes are associated only with such pages. By Lemma 6.4 we can conclude that each hole can result in at most one page fault per process up to the end of the phase, so that the total number of page faults in the phase is bounded by $\ell + \ell P$. Using Lemma 5.1 gives the above competitive factor. □

THEOREM A.2. *The competitive ratio of any deterministic online algorithm for the multiapplication caching problem is at least* $P+1$.

Proof. Since the algorithm is deterministic, we can construct an interleave that costs the algorithm a factor of $P+1$ times the number of faults that *OPT* will incur. For instance, consider a single clean request in each phase. On the basis of our knowledge of the deterministic choices made by the algorithm, we can easily make the resulting hole visit each process at least once so that the deterministic online algorithm incurs at least $P + 1$ faults per phase, whereas OPT incurs just one fault. $\square$

REFERENCES

[Bel66] A. L. Belady, *A study of replacement algorithms for virtual storage computers*, IBM Systems J., 5 (1966), pp. 78–101.

[BIRS91] A. Borodin, S. Irani, P. Raghavan, and B. Schieber, *Competitive paging with locality of reference*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computation, New Orleans, LA, 1991, pp. 249–259.

[CFL94a] P. Cao, E. W. Felten, and K. Li, *Application-controlled file caching policies*, in Proceedings of the Summer USENIX Conference, Boston, MA, 1994, pp. 171–182.

[CFL94b] P. Cao, E. W. Felten, and K. Li, *Implementation and performance of application-contolled file caching*, in Proceedings of the First OSDI Symposium, Monterey, CA, 1994, pp. 165–177.

[FKL$^+$91] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, *On competitive algorithms for paging problems*, J. Algorithms, 12 (1991), pp. 685–699.

[KPR92] A. R. Karlin, S. J. Phillips, and P. Raghavan, *Markov paging*, in Proceedings of the 33rd Annual IEEE Conference on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 208–217.

[MS91] L. A. McGeoch and D. D. Sleator, *A strongly competitive randomized paging algorithm*, Algorithmica, 6 (1991), pp. 816–825.

[ST85] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1304–1320

PERFECT-INFORMATION LEADER ELECTION WITH OPTIMAL RESILIENCE*

RAVI B. BOPPANA† AND BABU O. NARAYANAN‡

Abstract. This paper investigates the leader-election problem in the perfect-information model of distributed computing. It is shown that for every $\epsilon < \frac{1}{2}$, there exist leader-election protocols for n processors that tolerate ϵn faults.

Key words. distributed computing, fault tolerance, leader election, perfect information, probabilistic methods

AMS subject classifications. 68Q22, 68Q25, 60C05

PII. S0097539796307182

1. Introduction.

1.1. Statement of problem.

A distributed system of computers often needs to designate one of its processors as the "leader." Some of the processors may be faulty, but the leader should be nonfaulty with probability bounded away from 0. We follow the perfect-information model of Ben-Or and Linial [4] in which a processor communicates only by broadcasting a message to every other processor; faulty processors are permitted unlimited computational power, and hence cryptographic techniques are useless.

Our main result is that for every $\epsilon < \frac{1}{2}$, there exist leader-election protocols for n processors that tolerate ϵn faults.

1.2. Relation to other work.

Ben-Or and Linial [4] introduced the perfect-information model of distributed computing; they constructed a leader-election protocol for n processors that tolerates $O(n^{\log_3 2}/\log n)$ faults. (Actually they constructed their protocol for the weaker problem of collective coinflipping, but their protocol works for leader election also.) Saks [16] constructed an election protocol called "pass the baton" that tolerates $O(n/\log n)$ faults; Ajtai and Linial [1] analyzed the baton-passing protocol in more detail. Subsequently, Alon and Naor [2] showed the existence of a protocol that tolerates ϵn faults for every $\epsilon < \frac{1}{3}$: their protocol is nonconstructive. (They designed a constructive protocol for some small but positive constant ϵ.) In [6], we showed that the (nonconstructive) Alon–Naor protocol tolerates ϵn faults for every $\epsilon < (2\sqrt{10} - 5)/3 \approx 0.44$. Finally, in the present paper, we show that the Alon–Naor protocol actually tolerates ϵn faults for every $\epsilon < \frac{1}{2}$.

Our bound of ϵn faults for every $\epsilon < \frac{1}{2}$ is asymptotically optimal: Saks [16] observed that no election protocol for n processors can tolerate $\lceil n/2 \rceil$ faults. (As Saks did not publish his proof, we present a proof in section 3.) These two bounds leave open the case of $n/2 - o(n)$ faults. Because our upper bound seems hard to improve, we conjecture that $n/2 - o(n)$ faults are impossible to tolerate.

*Received by the editors July 24, 1996; accepted for publication (in revised form) October 19, 1998; published electronically February 18, 2000.
http://www.siam.org/journals/sicomp/29-4/30718.html

†Bloomberg Financial Markets, 499 Park Avenue, New York, NY 10022 (rboppana@bloomberg.net). The research of this author was supported in part by Presidential Young Investigator Award CCR–9196230 from the National Science Foundation.

‡Qualcomm, Inc. 6455 Lusk Boulevard, San Diego, CA 92122 (babun@qualcomm.com).

Recently, researchers have investigated the running time of protocols. In each round, each processor may broadcast a message to every other processor. Cooper and Linial [9] constructed a protocol that runs in polylogarithmic time and yet tolerates ϵn faults for some positive constant ϵ.

After our work was completed, Ostrovsky, Rajagopalan, and Vazirani [14] showed nonconstructively the existence of a protocol with $O(\log n)$ rounds that tolerates ϵn faults for every $\epsilon < \frac{1}{2}$; they designed a constructive protocol for $\epsilon < 0.0045$. Later, Zuckerman [17] designed a constructive protocol with $O(\log n)$ rounds that tolerates ϵn faults for every $\epsilon < \frac{1}{2}$. After that, Russell and Zuckerman [15] designed a constructive protocol with $\log^* n + O(1)$ rounds that tolerates ϵn faults for every $\epsilon < \frac{1}{2}$. All three papers use our protocol as a subroutine for small subsets of processors.

For two delightful surveys of the perfect-information model and related topics, see Ben-Or, Linial, and Saks [5] and Linial [13].

The most common model in distributed computing is not the perfect-information model, but rather the message-passing model in which processors may send private messages to other processors. For that model, Feldman and Micali [10] present a randomized protocol for leader election that tolerates a linear number of faults and yet runs in a constant expected number of rounds. See Chor and Dwork [8] for an excellent survey of the message-passing model.

1.3. Organization. In section 2, we present the formal definition of election protocols. In section 3, we prove Saks's impossibility result. In section 4, we present the probabilistic construction of election protocols due to Alon and Naor. In section 5, we present an overview of the proof of our main result. In sections 6, 7, and 8, we give the actual proof.

2. Protocols.

In this section, we present the formal definition of election protocols, introduced by Saks [16]. Throughout this section, let X be a finite set of processors.

DEFINITION 2.1 (election protocol). (a) An *election protocol* for X is a nonempty, strictly-binary tree with each node labeled by a processor in X.

(b) Let P be an election protocol for X and $A \subseteq X$ be a set of faulty processors. We define $\mathrm{Pr}_A(P)$ recursively as follows. If P consists of a single node (with label x), then

$$\mathrm{Pr}_A(P) = \begin{cases} 1 & \text{if } x \notin A, \\ 0 & \text{if } x \in A. \end{cases}$$

If P consists of a root node (with label x), a left subprotocol L, and a right subprotocol R, then

$$\mathrm{Pr}_A(P) = \begin{cases} \frac{1}{2}(\mathrm{Pr}_A(L) + \mathrm{Pr}_A(R)) & \text{if } x \notin A, \\ \min(\mathrm{Pr}_A(L), \mathrm{Pr}_A(R)) & \text{if } x \in A. \end{cases}$$

Informally, think of an election protocol as a biased random tree process. The processor x at the root node is supposed to choose the left or right subtree uniformly at random. (If x is faulty, that is, if $x \in A$, then it makes an adversarial choice, not a random choice.) The process continues until we reach a leaf node; its processor is selected as the leader. The number $\mathrm{Pr}_A(P)$ is the minimum probability that the protocol P elects a nonfaulty leader.

For example, suppose that the processor set X is $\{1, 2, 3\}$. Suppose that the protocol P is the 3-node binary tree in which the root is labeled by 1, the root's left child is labeled by 2, and the root's right child is labeled by 3. If $A = \emptyset$ or $A = \{1\}$, then $\Pr_A(P) = 1$. If $A = \{2\}$ or $A = \{3\}$, then $\Pr_A(P) = \frac{1}{2}$. Finally, if $A = \{1, 2\}$ or $A = \{1, 3\}$ or $A = \{2, 3\}$ or $A = \{1, 2, 3\}$, then $\Pr_A(P) = 0$.

We could generalize election protocols to allow more than two subprotocols and a nonuniform distribution on the subprotocols, but we will not need these generalizations.

Next, we define the notion of resilience.

Definition 2.2 (resilience). (a) If P is an election protocol for X, and t is a nonnegative real number, then

$$\Pr_t(P) = \min \{\Pr_A(P) : A \subseteq X \wedge |A| \leq t\}.$$

(b) An election protocol P is *t-resilient* if $\Pr_t(P)$ is bounded away from 0. (Technically, for "bounded away from 0" to make sense, we should speak of a *sequence* $P(n)$ of protocols being $t(n)$-resilient. We shall ignore this technicality.)

To illustrate, let P be the previous example protocol. Then $\Pr_0(P) = 1$, $\Pr_1(P) = \frac{1}{2}$, $\Pr_2(P) = 0$, and $\Pr_3(P) = 0$.

3. Limitations.

Saks [16] states that an elementary argument shows that an election protocol cannot tolerate $\lceil n/2 \rceil$ faults. In this section, we provide the proof.

Theorem 3.1 (dichotomy). *If A and B are finite sets, and P is an election protocol for $A \cup B$, then* $\min(\Pr_A(P), \Pr_B(P)) = 0$.

Proof. Nonnegativity is trivial, so we prove only nonpositivity. The proof is by structural induction on P.

Base case (*P has exactly one node*). Let x be the processor of the root node of P. Because $x \in A \cup B$, either $x \in A$ or $x \in B$; without loss of generality, assume that $x \in A$. By the definitions of min and Pr, we have

$$\min(\Pr_A(P), \Pr_B(P)) \leq \Pr_A(P) = 0.$$

Inductive case (*P has more than one node*). In P, let x be the root processor, let L be the left subprotocol, and let R be the right subprotocol. Because $x \in A \cup B$, either $x \in A$ or $x \in B$; without loss of generality, assume that $x \in A$.

By the definition of Pr, we have

$$\begin{aligned} \Pr_B(P) &= \begin{cases} \frac{1}{2}(\Pr_B(L) + \Pr_B(R)) & \text{if } x \notin B, \\ \min(\Pr_B(L), \Pr_B(R)) & \text{if } x \in B \end{cases} \\ &\leq \frac{1}{2}(\Pr_B(L) + \Pr_B(R)) \\ &\leq \max(\Pr_B(L), \Pr_B(R)). \end{aligned} \tag{3.1}$$

By the definition of Pr and the assumption $x \in A$, we have

$$\Pr_A(P) = \min(\Pr_A(L), \Pr_A(R)). \tag{3.2}$$

Hence, by (3.1), the distributive law of min over max, (3.2), and the inductive hypothesis, we have

$$\begin{aligned}\min(\Pr_A(P), \Pr_B(P)) &\le \min(\Pr_A(P), \max(\Pr_B(L), \Pr_B(R)))\\ &= \max(\min(\Pr_A(P), \Pr_B(L)), \min(\Pr_A(P), \Pr_B(R)))\\ &\le \max(\min(\Pr_A(L), \Pr_B(L)), \min(\Pr_A(R), \Pr_B(R)))\\ &= \max(0, 0)\\ &= 0.\end{aligned}$$

This inequality completes the induction. □

The proof of the preceding theorem resembles the proof of a classical result in game theory: In every two-player, perfect-information game, one of the two players has a winning strategy. The resemblance is no accident: we could have derived our theorem as a consequence of the game-theoretic result.

We now derive Saks's impossibility result.

COROLLARY 3.2 (Saks). *If P is an election protocol for X, and $t \ge \lceil |X|/2 \rceil$, then $\Pr_t(P) = 0$.*

Proof. Nonnegativity is trivial, so we prove only nonpositivity. Let A and B be subsets of X such that $A \cup B = X$ and $|A| \le \lceil |X|/2 \rceil$ and $|B| \le \lceil |X|/2 \rceil$. Then by the definition of $\Pr_t$, and by Theorem 3.1, we have

$$\Pr_t(P) \le \min(\Pr_A(P), \Pr_B(P)) = 0.$$

This inequality completes the proof. □

4. Random protocols.

In this section we present the probabilistic construction of election protocols due to Alon and Naor [2]. For the remainder of this paper, let ϵ be a constant such that $0 \le \epsilon < \frac{1}{2}$. Let X be a nonempty finite set.

DEFINITION 4.1 (uniform distribution). The *uniform distribution* $\mathrm{Unif}(X)$ is the probability distribution on X that assigns each element of X the probability $1/|X|$.

DEFINITION 4.2 (function of distributions). Let f be a function on $A \times B$, and let C and D be probability distributions on A and B, respectively. Then $\widetilde{f}(C, D)$ is defined to be the probability distribution of the random variable $f(c, d)$, where c and d are independent random variables with distributions C and D, respectively. Although we presented this definition for a function with two arguments, we will generalize the definition in the obvious way to functions with fewer or more arguments.

DEFINITION 4.3 (random protocols). (a) If x is an element of X, and L and R are election protocols for X, then $f(x, L, R)$ is the election protocol for X whose root node is labeled x, whose left subprotocol is L, and whose right subprotocol is R.

(b) If D is a probability distribution on election protocols for X, then $F(D)$ is the probability distribution on election protocols for X defined by

$$F(D) = \widetilde{f}(\mathrm{Unif}(X), D, D).$$

(Here we are using Definition 4.2 to extend f to $\widetilde{f}$.) Informally, $F(D)$ has a random processor at its root and has independent copies of D as its left and right subprotocols.

(c) If i is a nonnegative integer, then $\mathcal{F}_i$ is the probability distribution on election protocols for X defined recursively as follows. If $i = 0$, then $\mathcal{F}_i$ ranges over trees with just one node, with that node labeled uniformly at random from X. If $i > 0$, then $\mathcal{F}_i = F(\mathcal{F}_{i-1})$. Informally, $\mathcal{F}_i$ is a full binary tree of height i, in which every node is labeled by a random processor.

We need to analyze the probability that the random protocol $\mathcal{F}_i$ elects a nonfaulty leader. It turns out, as Alon and Naor [2] observed, that this probability can be captured by a random number process that does not refer to protocols themselves. We now define this random process.

Definition 4.4 (common sets). Define $\mathbf{R}$ to be the set of real numbers. Define $\mathbf{R}^+$ to be the set of nonnegative real numbers. Define $\mathbf{B}$ to be the set $\{\text{true}, \text{false}\}$ of Boolean values.

Definition 4.5 (common distributions). Let p be a number between 0 and 1. Define the *Boolean distribution* $\text{Bool}(p)$ to be the distribution on $\mathbf{B}$ that assigns true the probability p and assigns false the probability $1-p$. Define the *Bernoulli distribution* $\text{Bern}(p)$ to be the distribution on $\{0,1\}$ that assigns 1 the probability p and assigns 0 the probability $1-p$.

Definition 4.6 (random numbers). (a) If b is a Boolean value, and x and y are [nonnegative] real numbers, then $g(b,x,y)$ is the [nonnegative] real number defined by

$$g(b,x,y) = \begin{cases} \min(x,y) & \text{if } b \text{ is true,} \\ (x+y)/2 & \text{otherwise.} \end{cases}$$

(b) If D is a probability distribution on $\mathbf{R}$ (or $\mathbf{R}^+$), then $G(D)$ is the probability distribution on $\mathbf{R}$ ($\mathbf{R}^+$) defined by

$$G(D) = \widetilde{g}(\text{Bool}(\epsilon), D, D).$$

Informally, given two independent arguments from D, we take their minimum with probability ϵ and their average with probability $1-\epsilon$.

(c) If i is a nonnegative integer, then $\mathcal{G}_i$ is the probability distribution on $\mathbf{R}^+$ defined recursively by

$$\mathcal{G}_i = \begin{cases} \text{Bern}(1-\epsilon) & \text{if } i = 0, \\ G(\mathcal{G}_{i-1}) & \text{if } i > 0. \end{cases}$$

Informally, $\mathcal{G}_i$ corresponds to a full binary tree of height i, in which each node is labeled randomly by a min or an average, with a bias toward averaging.

The following lemma shows that the distribution $\mathcal{G}_i$ of numbers is related to the distribution $\mathcal{F}_i$ of protocols.

Lemma 4.7 (protocols versus numbers). (a) *If A is a subset of X, and x is an element of X, and L and R are election protocols for X, then*

$$\Pr_A(f(x,L,R)) = g(x \in A, \Pr_A(L), \Pr_A(R)).$$

(b) *If A is a subset of X of size $\epsilon\,|X|$, and D is a probability distribution on election protocols for X, then*

$$\widetilde{\Pr_A}(F(D)) = G(\widetilde{\Pr_A}(D)).$$

(c) *If A is a subset of X of size $\epsilon\,|X|$, and i is a nonnegative integer, then*

$$\widetilde{\Pr_A}(\mathcal{F}_i) = \mathcal{G}_i.$$

Proof. (a) By the definitions of f and $\Pr_A$ and g, we have

$$\begin{aligned}\Pr_A(f(x,L,R)) &= \begin{cases} \min(\Pr_A(L), \Pr_A(R)) & \text{if } x \in A, \\ (\Pr_A(L) + \Pr_A(R))/2 & \text{otherwise} \end{cases} \\ &= g(x \in A, \Pr_A(L), \Pr_A(R)).\end{aligned}$$

(b) Because A has size $\epsilon\,|X|$, if x is uniformly distributed on X, then $x \in A$ has distribution $\mathrm{Bool}(\epsilon)$. By the definition of F, part (a), and the definition of G, we have

$$\begin{aligned}\widetilde{\mathrm{Pr}_A}(F(D)) &= \widetilde{\mathrm{Pr}_A}(\widetilde{f}(\mathrm{Unif}(X), D, D))\\ &= \widetilde{g}(\mathrm{Bool}(\epsilon), \widetilde{\mathrm{Pr}_A}(D), \widetilde{\mathrm{Pr}_A}(D))\\ &= G(\widetilde{\mathrm{Pr}_A}(D)).\end{aligned}$$

(c) The proof is by induction on i.

Base case $i = 0$. Recall that if x is uniformly distributed on X, then $x \in A$ has distribution $\mathrm{Bool}(\epsilon)$, and so $x \notin A$ has distribution $\mathrm{Bool}(1-\epsilon)$. Hence $\widetilde{\mathrm{Pr}_A}(\mathcal{F}_0)$ is the distribution $\mathrm{Bern}(1-\epsilon)$, which is $\mathcal{G}_0$.

Inductive case $i > 0$. By the definition of $\mathcal{F}_i$, part (b), the inductive hypothesis, and the definition of $\mathcal{G}_i$, we have

$$\widetilde{\mathrm{Pr}_A}(\mathcal{F}_i) = \widetilde{\mathrm{Pr}_A}(F(\mathcal{F}_{i-1})) = G(\widetilde{\mathrm{Pr}_A}(\mathcal{F}_{i-1})) = G(\mathcal{G}_{i-1}) = \mathcal{G}_i.$$

Hence all three parts of the lemma have been proved. ∎

5. Proof overview. In this section, we present an overview of our proof that the Alon–Naor random protocols tolerate ϵn faults (for every fixed $\epsilon < \frac{1}{2}$).

Throughout, let X be a finite nonempty set, of size n. We may assume that ϵn is an integer; otherwise we could use the smaller value $\lfloor \epsilon n \rfloor / n$ for ϵ.

Let A be a fixed subset of X of size ϵn. We will find a constant $\delta > 0$ (independent of n) such that for sufficiently large i, we obtain the tail bound

$$\mathrm{Pr}[\mathcal{G}_i \le \delta] < 1 \bigg/ \binom{n}{\epsilon n}. \tag{5.1}$$

By the definition of Pr_t, Boole's inequality, Lemma 4.7(c), and the tail bound (5.1), it would follow that

$$\begin{aligned}\mathrm{Pr}[\widetilde{\mathrm{Pr}_{\epsilon n}}(\mathcal{F}_i) \le \delta] &= \mathrm{Pr}\left[\bigvee_{|A|=\epsilon n} \widetilde{\mathrm{Pr}_A}(\mathcal{F}_i) \le \delta\right]\\ &\le \sum_{|A|=\epsilon n} \mathrm{Pr}[\widetilde{\mathrm{Pr}_A}(\mathcal{F}_i) \le \delta]\\ &= \sum_{|A|=\epsilon n} \mathrm{Pr}[\mathcal{G}_i \le \delta]\\ &< \sum_{|A|=\epsilon n} 1 \bigg/ \binom{n}{\epsilon n}\\ &= \binom{n}{\epsilon n} \bigg/ \binom{n}{\epsilon n}\\ &= 1.\end{aligned}$$

Hence there would be an election protocol P in the support of $\mathcal{F}_i$ such that $\mathrm{Pr}_{\epsilon n}(P) > \delta$; in other words, P would be ϵn-resilient, and we would be done.

To prove the tail bound (5.1) for $\epsilon < \frac{1}{3}$, Alon and Naor [2] used estimates on the expectation and variance of $\mathcal{G}_i$. To handle larger values of ϵ, we need to analyze higher moments of $\mathcal{G}_i$.

Definition 5.1 (expected value). If x is a random variable on $\mathbf{R}$, then $\mathrm{E}(x)$ denotes its expected value. If D is a probability distribution on $\mathbf{R}$, then $\mathrm{E}(D)$ denotes $\mathrm{E}(x)$, where x is a random variable with distribution D.

Definition 5.2 (moment). If D is a probability distribution on $\mathbf{R}$ and d is a nonnegative real number, then the dth *moment* of D is

$$\mathrm{M}_d(D) = \mathrm{E}(|x-y|^d),$$

where x and y are independent random variables with distribution D.

Our proof of the tail bound (5.1) divides into three steps:

(i) Our key step is to show that the higher moments of $\mathcal{G}_i$ are geometrically contracting in i; loosely speaking, each of these distributions is "concentrated" near one value.

(ii) Once we have this contraction result, we will show that the expectation of $\mathcal{G}_i$ is bounded away from 0; for small ϵ, this step is easy, but for large ϵ, we need to use a majorization argument that relates our random process to another process with smaller moments.

(iii) Once we know that the expectation of $\mathcal{G}_i$ is bounded away from 0, a simple argument using Chebyshev's inequality will give the tail bound (5.1).

In [7] we used the same outline of steps to analyze a biased-coin process, extending a result of Alon and Rabin [3]. The key difference between [7] and the present paper, besides the difference in problems, is in step (i): The contraction argument here is much more complicated than the one in [7], perhaps unavoidably so. Steps (ii) and (iii) resemble the corresponding steps in [7].

In section 6, we prove the contraction result. In section 7, we develop the majorization argument. Finally, in section 8, we prove the tail bound (5.1).

6. Contraction.

In this section, we prove that the higher moments of the distribution $\mathcal{G}_i$ geometrically contract with i. Let d be a nonnegative real number, the moment that we will focus on.

Definition 6.1 (Δ function). Let $x = (x_1, x_2, x_3, x_4)$ be a vector in $\mathbf{R}^4$. The function $\Delta\colon \mathbf{R}^4 \to \mathbf{R}$ is defined by

$$\Delta(x) = \Delta_1(x) + \Delta_2(x) + \Delta_3(x) + \Delta_4(x),$$

where

$$\Delta_1(x) = (1-\epsilon)^2 \left|\frac{x_1+x_2}{2} - \frac{x_3+x_4}{2}\right|^d,$$

$$\Delta_2(x) = \epsilon(1-\epsilon) \left|\frac{x_1+x_2}{2} - \min(x_3,x_4)\right|^d,$$

$$\Delta_3(x) = \epsilon(1-\epsilon) \left|\min(x_1,x_2) - \frac{x_3+x_4}{2}\right|^d,$$

and

$$\Delta_4(x) = \epsilon^2 \left|\min(x_1,x_2) - \min(x_3,x_4)\right|^d.$$

We defined Δ this way to get the following identity:

$$\mathrm{E}(|\widetilde{g}(\mathrm{Bool}(\epsilon), x_1, x_2) - \widetilde{g}(\mathrm{Bool}(\epsilon), x_3, x_4)|^d) = \Delta(x_1, x_2, x_3, x_4).$$

(Strictly speaking, for this identity to make sense, we should view g, and hence $\widetilde{g}$, as a function of just one argument, treating its last two arguments as constants. The identity follows from the definitions of g and Δ.)

Now let D be a probability distribution on $\mathbf{R}$. Let x be a random vector in $\mathbf{R}^4$ such that x_1, x_2, x_3, and x_4 are independent random variables with distribution D. Taking the expected value of the previous equation, and using the definition of G, we get the identity

$$\mathrm{M}_d(G(D)) = \mathrm{E}(\widetilde{\Delta}(D,D,D,D)). \tag{6.1}$$

The following lemma, an upper bound on Δ, is the key tool in proving our contraction result.

Lemma 6.2 (deterministic contraction). *If x is a vector in $\mathbf{R}^4$, then*

$$\begin{aligned}\Delta(x_1,x_2,x_3,x_4) + \Delta(x_3,x_2,x_1,x_4) + \Delta(x_1,x_3,x_2,x_4)&\\ \le \frac{1}{2}c \sum_{1\le i<j\le 4} \left|x_i - x_j\right|^d,&\end{aligned} \tag{6.2}$$

where $c = \max(2\epsilon, \frac{1+\epsilon}{2}) + 6(\frac{3}{4})^d$.

Proof. Throughout the proof we rely, without explicit mention, on the inequalities

$$\left|\frac{p+q}{2}\right|^d \le \frac{1}{2}\left[|p|^d + |q|^d\right]$$

and

$$\left|\frac{p+q+r+s}{4}\right|^d \le \frac{1}{4}\left[|p|^d + |q|^d + |r|^d + |s|^d\right],$$

which are special cases of the power mean inequality [12, Theorem 16].

Both sides of the inequality (6.2) are symmetric functions of x. Hence we may assume that $x_1 \le x_2 \le x_3 \le x_4$. The proof divides into two cases, depending on the value of x_2.

Case 1 ($x_2 \ge (x_1+x_3)/2$). Our first three steps are to develop an upper bound on each of the three Δ terms in (6.2); our last step is to add these three upper bounds to establish (6.2).

Step 1.1. We bound the four terms of $\Delta(x_1,x_2,x_3,x_4)$:

$$\begin{aligned}\Delta_1(x_1,x_2,x_3,x_4) &= (1-\epsilon)^2\left|\frac{x_3+x_4}{2} - \frac{x_1+x_2}{2}\right|^d\\ &\le (1-\epsilon)^2\left|\frac{x_3+x_4}{2} - \frac{x_1+(x_1+x_3)/2}{2}\right|^d\\ &= (1-\epsilon)^2\left|\frac{x_3+2x_4-3x_1}{4}\right|^d \le (1-\epsilon)^2\left(\frac{3}{4}\right)^d |x_4-x_1|^d\\ &\le \left(\frac{3}{4}\right)^d |x_4-x_1|^d;\end{aligned}$$

$$\begin{aligned}\Delta_2(x_1,x_2,x_3,x_4) &= \epsilon(1-\epsilon)\left|x_3 - \frac{x_1+x_2}{2}\right|^d \le \epsilon(1-\epsilon)\left|x_3 - \frac{x_1+(x_1+x_3)/2}{2}\right|^d\\ &= \epsilon(1-\epsilon)\left(\frac{3}{4}\right)^d |x_3-x_1|^d \le \left(\frac{3}{4}\right)^d |x_3-x_1|^d;\end{aligned}$$

$$\begin{aligned}\Delta_3(x_1,x_2,x_3,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_3+x_4}{2}-x_1\right|^d = \epsilon(1-\epsilon)\left|\frac{x_3-x_1}{2}+\frac{x_4-x_1}{2}\right|^d \\ &\le \frac{1}{2}\epsilon(1-\epsilon)\left[|x_3-x_1|^d+|x_4-x_1|^d\right]; \\ \Delta_4(x_1,x_2,x_3,x_4) &= \epsilon^2|x_3-x_1|^d.\end{aligned}$$

Adding these four inequalities gives the bound

$$\begin{aligned}\Delta(x_1,x_2,x_3,x_4) \le{}& \frac{1}{2}\epsilon(1+\epsilon)|x_3-x_1|^d+\frac{1}{2}\epsilon(1-\epsilon)|x_4-x_1|^d \\ &+\left(\frac{3}{4}\right)^d|x_3-x_1|^d+\left(\frac{3}{4}\right)^d|x_4-x_1|^d.\end{aligned}\tag{6.3}$$

Step 1.2. We bound the four terms of $\Delta(x_3,x_2,x_1,x_4)$:

$$\begin{aligned}\Delta_1(x_3,x_2,x_1,x_4) &= (1-\epsilon)^2\left|\frac{x_1+x_4}{2}-\frac{x_3+x_2}{2}\right|^d \le (1-\epsilon)^2\left|\frac{x_4-x_1}{2}\right|^d \\ &= (1-\epsilon)^2\left(\frac{1}{2}\right)^d|x_4-x_1|^d \le (1-\epsilon)\left(\frac{3}{4}\right)^d|x_4-x_1|^d; \\ \Delta_2(x_3,x_2,x_1,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_3+x_2}{2}-x_1\right|^d \le \frac{1}{2}\epsilon(1-\epsilon)\left[|x_2-x_1|^d+|x_3-x_1|^d\right]; \\ \Delta_3(x_3,x_2,x_1,x_4) &= \epsilon(1-\epsilon)\left|x_2-\frac{x_1+x_4}{2}\right|^d \le \epsilon(1-\epsilon)\left|\frac{x_4-x_1}{2}\right|^d \\ &= \epsilon(1-\epsilon)\left(\frac{1}{2}\right)^d|x_4-x_1|^d \le \epsilon\left(\frac{3}{4}\right)^d|x_4-x_1|^d; \\ \Delta_4(x_3,x_2,x_1,x_4) &= \epsilon^2|x_2-x_1|^d.\end{aligned}$$

Adding these four inequalities gives

$$\begin{aligned}\Delta(x_3,x_2,x_1,x_4) \le{}& \frac{1}{2}\epsilon(1+\epsilon)|x_2-x_1|^d+\frac{1}{2}\epsilon(1-\epsilon)|x_3-x_1|^d \\ &+\left(\frac{3}{4}\right)^d|x_4-x_1|^d.\end{aligned}\tag{6.4}$$

Step 1.3. We bound the four terms of $\Delta(x_1,x_3,x_2,x_4)$:

$$\begin{aligned}\Delta_1(x_1,x_3,x_2,x_4) &= (1-\epsilon)^2\left|\frac{x_1+x_3}{2}-\frac{x_2+x_4}{2}\right|^d \le (1-\epsilon)^2\left|\frac{x_4-x_1}{2}\right|^d \\ &= (1-\epsilon)^2\left(\frac{1}{2}\right)^d|x_4-x_1|^d \le \left(\frac{3}{4}\right)^d|x_4-x_1|^d; \\ \Delta_2(x_1,x_3,x_2,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_1+x_3}{2}-x_2\right|^d \le \epsilon(1-\epsilon)\left|\frac{x_3-x_1}{2}\right|^d \\ &= \epsilon(1-\epsilon)\left(\frac{1}{2}\right)^d|x_3-x_1|^d \le \left(\frac{3}{4}\right)^d|x_3-x_1|^d; \\ \Delta_3(x_1,x_3,x_2,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_4+x_2}{2}-x_1\right|^d \le \frac{1}{2}\epsilon(1-\epsilon)\left[|x_2-x_1|^d+|x_4-x_1|^d\right];\end{aligned}$$

$$\Delta_4(x_1, x_3, x_2, x_4) = \epsilon^2 |x_2 - x_1|^d \leq \epsilon^2 |x_4 - x_1|^d .$$

Adding these four inequalities gives

$$\begin{aligned}\Delta(x_1, x_3, x_2, x_4) \leq {} & \frac{1}{2}\epsilon(1-\epsilon)|x_2 - x_1|^d + \frac{1}{2}\epsilon(1+\epsilon)|x_4 - x_1|^d \\ & + \left(\frac{3}{4}\right)^d |x_3 - x_1|^d + \left(\frac{3}{4}\right)^d |x_4 - x_1|^d .\end{aligned} \tag{6.5}$$

Step 1.4. Adding the three bounds (6.3), (6.4), and (6.5), and using the definition of c, we obtain

$$\begin{aligned}&\Delta(x_1, x_2, x_3, x_4) + \Delta(x_3, x_2, x_1, x_4) + \Delta(x_1, x_3, x_2, x_4) \\ &\quad \leq \epsilon |x_2 - x_1|^d + \epsilon |x_3 - x_1|^d + \epsilon |x_4 - x_1|^d + 2\left(\frac{3}{4}\right)^d |x_3 - x_1|^d + 3\left(\frac{3}{4}\right)^d |x_4 - x_1|^d \\ &\quad \leq \left[\epsilon + 3\left(\frac{3}{4}\right)^d\right] \sum_{i<j} |x_i - x_j|^d \leq \frac{1}{2} c \sum_{i<j} |x_i - x_j|^d .\end{aligned}$$

This inequality is (6.2), which settles Case 1.

Case 2 ($x_2 \leq (x_1 + x_3)/2$). Again, our first three steps are to develop an upper bound on each of the three Δ terms in the inequality (6.2); our last step is to add these three upper bounds to establish (6.2).

Step 2.1 We bound the four terms of $\Delta(x_1, x_2, x_3, x_4)$:

$$\begin{aligned}\Delta_1(x_1, x_2, x_3, x_4) &= (1-\epsilon)^2 \left|\frac{x_1 + x_2}{2} - \frac{x_3 + x_4}{2}\right|^d \\ &= (1-\epsilon)^2 \left|\frac{x_4 - x_1}{4} + \frac{x_4 - x_2}{4} + \frac{x_3 - x_1}{4} + \frac{x_3 - x_2}{4}\right|^d \\ &\leq \frac{1}{4}(1-\epsilon)^2 \left[|x_4 - x_1|^d + |x_4 - x_2|^d + |x_3 - x_1|^d + |x_3 - x_2|^d\right]; \\ \Delta_2(x_1, x_2, x_3, x_4) &= \epsilon(1-\epsilon)\left|\frac{x_1 + x_2}{2} - x_3\right|^d \leq \frac{1}{2}\epsilon(1-\epsilon)\left[|x_3 - x_1|^d + |x_3 - x_2|^d\right]; \\ \Delta_3(x_1, x_2, x_3, x_4) &= \epsilon(1-\epsilon)\left|x_1 - \frac{x_3 + x_4}{2}\right|^d \leq \frac{1}{2}\epsilon(1-\epsilon)\left[|x_3 - x_1|^d + |x_4 - x_1|^d\right]; \\ \Delta_4(x_1, x_2, x_3, x_4) &= \epsilon^2 |x_3 - x_1|^d .\end{aligned}$$

Adding these four inequalities and using $|x_3 - x_1| \leq |x_4 - x_1|$ gives the bound

$$\begin{aligned}\Delta(x_1, x_2, x_3, x_4) &\leq \frac{1}{4}(1+\epsilon)^2 |x_3 - x_1|^d + \frac{1}{4}(1-\epsilon^2)|x_4 - x_1|^d \\ &\qquad + \frac{1}{4}(1-\epsilon^2)|x_3 - x_2|^d + \frac{1}{4}(1-\epsilon)^2 |x_4 - x_2|^d \\ &\leq \frac{1+\epsilon}{4}|x_3 - x_1|^d + \frac{1+\epsilon}{4}|x_4 - x_1|^d \\ &\qquad + \frac{1}{4}(1-\epsilon^2)|x_3 - x_2|^d + \frac{1}{4}(1-\epsilon)^2 |x_4 - x_2|^d \\ &\leq \frac{1+\epsilon}{4}\sum_{i<j}|x_i - x_j|^d .\end{aligned} \tag{6.6}$$

Step 2.2. We bound the four terms of $\Delta(x_3, x_2, x_1, x_4)$:

$$\begin{aligned}
\Delta_1(x_3,x_2,x_1,x_4) &= (1-\epsilon)^2 \left|\frac{x_3+x_2}{2} - \frac{x_1+x_4}{2}\right|^d \le (1-\epsilon)^2\left(\frac{1}{2}\right)^d |x_4-x_1|^d \\
&\le (1-\epsilon)\left(\frac{3}{4}\right)^d |x_4-x_1|^d; \\
\Delta_2(x_3,x_2,x_1,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_3+x_2}{2} - x_1\right|^d = \epsilon(1-\epsilon)\left|\frac{x_3+(x_1+x_3)/2}{2} - x_1\right|^d \\
&= \epsilon(1-\epsilon)\left(\frac{3}{4}\right)^d |x_3-x_1|^d \le (1-\epsilon)\left(\frac{3}{4}\right)^d |x_3-x_1|^d; \\
\Delta_3(x_3,x_2,x_1,x_4) &= \epsilon(1-\epsilon)\left|x_2 - \frac{x_1+x_4}{2}\right|^d \le \epsilon(1-\epsilon)\left(\frac{1}{2}\right)^d |x_4-x_1|^d \\
&\le \epsilon\left(\frac{3}{4}\right)^d |x_4-x_1|^d; \\
\Delta_4(x_3,x_2,x_1,x_4) &= \epsilon^2 |x_2-x_1|^d \le \epsilon^2\left|\frac{x_1+x_3}{2} - x_1\right|^d \\
&\le \epsilon^2\left(\frac{1}{2}\right)^d |x_3-x_1|^d \le \epsilon\left(\frac{3}{4}\right)^d |x_3-x_1|^d.
\end{aligned}$$

Adding these four inequalities gives

$$\Delta(x_3,x_2,x_1,x_4) \le \left(\frac{3}{4}\right)^d |x_3-x_1|^d + \left(\frac{3}{4}\right)^d |x_4-x_1|^d. \tag{6.7}$$

Step 2.3. We bound the four terms of $\Delta(x_1, x_3, x_2, x_4)$:

$$\begin{aligned}
\Delta_1(x_1,x_3,x_2,x_4) &= (1-\epsilon)^2 \left|\frac{x_1+x_3}{2} - \frac{x_2-x_4}{2}\right|^d \le (1-\epsilon)^2\left(\frac{1}{2}\right)^d |x_4-x_1|^d \\
&\le (1-\epsilon)\left(\frac{3}{4}\right)^d |x_4-x_1|^d; \\
\Delta_2(x_1,x_3,x_2,x_4) &= \epsilon(1-\epsilon)\left|\frac{x_1+x_3}{2} - x_2\right|^d \le \epsilon(1-\epsilon)\left(\frac{1}{2}\right)^d |x_3-x_1|^d \\
&\le (1-\epsilon)\left(\frac{3}{4}\right)^d |x_3-x_1|^d; \\
\Delta_3(x_1,x_3,x_2,x_4) &= \epsilon(1-\epsilon)\left|x_1 - \frac{x_2+x_4}{2}\right|^d \le \epsilon(1-\epsilon)\left|x_1 - \frac{(x_1+x_3)/2+x_4}{2}\right|^d \\
&= \epsilon(1-\epsilon)\left|\frac{2x_4-x_3-3x_1}{4}\right|^d \le \epsilon(1-\epsilon)\left(\frac{3}{4}\right)^d |x_4-x_1|^d \\
&\le \epsilon\left(\frac{3}{4}\right)^d |x_4-x_1|^d;
\end{aligned}$$

$$\begin{aligned}\Delta_4(x_1, x_3, x_2, x_4) &= \epsilon^2 \left|x_2 - x_1\right|^d \le \epsilon^2 \left|\frac{x_1 + x_3}{2} - x_1\right|^d \\ &= \epsilon^2 \left(\frac{1}{2}\right)^d \left|x_3 - x_1\right|^d \le \epsilon \left(\frac{3}{4}\right)^d \left|x_3 - x_1\right|^d.\end{aligned}$$

Adding these four inequalities gives

$$\Delta(x_1, x_3, x_2, x_4) \le \left(\frac{3}{4}\right)^d \left|x_3 - x_1\right|^d + \left(\frac{3}{4}\right)^d \left|x_4 - x_1\right|^d. \tag{6.8}$$

Step 2.4. Adding the three bounds (6.6), (6.7), and (6.8), and using the definition of c, we obtain

$$\begin{aligned}&\Delta(x_1, x_2, x_3, x_4) + \Delta(x_3, x_2, x_1, x_4) + \Delta(x_1, x_3, x_2, x_4) \\ &\quad\le \frac{1}{4}(1+\epsilon)\sum_{i<j}\left|x_i - x_j\right|^d + 2\left(\frac{3}{4}\right)^d \left|x_3 - x_1\right|^d + 2\left(\frac{3}{4}\right)^d \left|x_4 - x_1\right|^d \\ &\quad\le \left[\frac{1+\epsilon}{4} + 2\left(\frac{3}{4}\right)^d\right]\sum_{i<j}\left|x_i - x_j\right|^d \le \frac{1}{2}c\sum_{i<j}\left|x_i - x_j\right|^d.\end{aligned}$$

This inequality is (6.2), which settles Case 2, which settles the lemma. ∎

Using the preceding lemma, we can prove the contraction result that we desire.

LEMMA 6.3 (contraction lemma). *If D is a probability distribution on $\mathbf{R}$ and d is a nonnegative real number, then*

$$\mathrm{M}_d(G(D)) \le c\,\mathrm{M}_d(D),$$

where $c = \max(2\epsilon, \frac{1+\epsilon}{2}) + 6(\frac{3}{4})^d$.

Proof. Let x be a random vector in $\mathbf{R}^4$ such that x_1, x_2, x_3, and x_4 are independent random variables with distribution D. Note that the three tuples (x_1, x_2, x_3, x_4), (x_3, x_2, x_1, x_4), and (x_1, x_3, x_2, x_4) are identically distributed. In particular, we have

$$\begin{aligned}\mathrm{E}(\widetilde{\Delta}(D, D, D, D)) &= \mathrm{E}(\Delta(x_1, x_2, x_3, x_4)) \\ &= \mathrm{E}(\Delta(x_3, x_2, x_1, x_4)) \\ &= \mathrm{E}(\Delta(x_1, x_3, x_2, x_4)).\end{aligned}$$

Hence, from the identity (6.1) and Lemma 6.2, we get

$$\begin{aligned}\mathrm{M}_d(G(D)) &= \mathrm{E}(\widetilde{\Delta}(D, D, D, D)) \\ &= \frac{1}{3}\left[\mathrm{E}(\Delta(x_1, x_2, x_3, x_4)) + \mathrm{E}(\Delta(x_3, x_2, x_1, x_4)) + \mathrm{E}(\Delta(x_1, x_3, x_2, x_4))\right] \\ &\le \frac{1}{6}c\sum_{1\le i<j\le 4}\mathrm{E}(\left|x_i - x_j\right|^d) \\ &= \frac{1}{6}c\sum_{1\le i<j\le 4}\mathrm{M}_d(D) \\ &= c\,\mathrm{M}_d(D).\end{aligned}$$

This inequality completes the proof. ∎

This lemma is a contraction result: The constant c can be made less than 1. Because $\epsilon < \frac{1}{2}$, every sufficiently large d makes $c < 1$. In fact, the choice $d = 8\ln\frac{6}{1-2\epsilon}$ works.

7. Majorization.

7. Majorization. In this section we relate our random process to another random process that is easier to analyze, using a concept called majorization. This argument is similar to the majorizing argument in Boppana and Narayanan [7].

DEFINITION 7.1 (majorization). Let C and D be probability distributions on $\mathbf{R}$. Say that C *majorizes* D, written $C \succeq D$, if there are random variables x and y such that

(i) the random variable x has distribution C,
(ii) the random variable y has distribution D, and
(iii) the inequality $x \geq y$ always holds.

Say that C is *majorized by* D, written $C \preceq D$, if D majorizes C.

Note that majorization is transitive. Also, if $C \succeq D$, then $\mathrm{E}(C) \geq \mathrm{E}(D)$.

We now show that the transformation G (from Definition 4.6) is "stochastically increasing."

LEMMA 7.2 (increasing). *If C and D are distributions on $\mathbf{R}$ such that $C \preceq D$, then $G(C) \preceq G(D)$.*

Proof. Note that $g(b, x, y)$ is increasing in x and y. By the definition of G and the hypothesis $C \preceq D$, it follows that

$$G(C) = \widetilde{g}(\mathrm{Bool}(\epsilon), C, C) \preceq \widetilde{g}(\mathrm{Bool}(\epsilon), D, D) = G(D).$$

That inequality completes the proof. □

We next define a random process $\mathcal{G}'_i$ that is majorized by our original process $\mathcal{G}_i$ but which can be analyzed exactly. (Compare the next definition with Definition 4.6.)

DEFINITION 7.3 (smaller random numbers). (a) If b is a Boolean value, and x and y are (nonnegative) real numbers, then $g'(b, x, y)$ is the (nonnegative) real number defined by

$$g'(b, x, y) = \begin{cases} \min(x, y)/2 & \text{if } b \text{ is true,} \\ \max(x, y)/2 & \text{otherwise.} \end{cases}$$

(b) If D is a probability distribution on $\mathbf{R}$ (or $\mathbf{R}^+$), then $G'(D)$ is the probability distribution on $\mathbf{R}$ ($\mathbf{R}^+$) defined by

$$G'(D) = \widetilde{g'}(\mathrm{Bool}(\epsilon), D, D).$$

(c) If i is a nonnegative integer, then $\mathcal{G}'_i$ is the probability distribution on $\mathbf{R}^+$ defined recursively by

$$\mathcal{G}'_i = \begin{cases} \mathrm{Bern}(1-\epsilon) & \text{if } i = 0, \\ G'(\mathcal{G}'_{i-1}) & \text{if } i > 0. \end{cases}$$

We now compare the unprimed distributions $\mathcal{G}_i$ to the primed distributions $\mathcal{G}'_i$.

LEMMA 7.4 (majorization). (a) *If b is a Boolean value, and x and y are nonnegative real numbers, then $g(b, x, y) \geq g'(b, x, y)$.*

(b) *If D is a distribution on $\mathbf{R}^+$, then $G(D) \succeq G'(D)$.*

(c) *If i is a nonnegative integer, then $\mathcal{G}_i \succeq \mathcal{G}'_i$.*

Proof.

(1) By the definitions of g and g', and the nonnegativity of x and y, we have

$$g(b, x, y) = g'(b, x, y) + \frac{1}{2}\min(x, y) \geq g'(b, x, y).$$

(2) Let b, x, and y be independent random variables such that b has distribution $\mathrm{Bool}(\epsilon)$, and x and y have distribution D. By the definition of G [or G'], the random variable $g(b,x,y)$ [$g'(b,x,y)$] has distribution $G(D)$ [$G'(D)$]. But $g(b,x,y) \geq g'(b,x,y)$ by part (a). The definition of majorization implies that $G(D) \succeq G'(D)$.

(3) The proof is by induction on i.

Base case ($i = 0$). By the definitions of $\mathcal{G}_0$ and $\mathcal{G}'_0$, we have

$$\mathcal{G}_0 = \mathrm{Bern}(1-\epsilon) = \mathcal{G}'_0.$$

Inductive case ($i > 0$). By the definition of $\mathcal{G}_i$, Lemma 7.2 and the inductive hypothesis, part (b), and the definition of $\mathcal{G}'_i$, we have

$$\mathcal{G}_i = G(\mathcal{G}_{i-1}) \succeq G(\mathcal{G}'_{i-1}) \succeq G'(\mathcal{G}'_{i-1}) = \mathcal{G}'_i.$$

Hence all three parts of the lemma have been proved. $\square$

We next show that the primed distributions $\mathcal{G}'_i$ are simple two-point distributions.

DEFINITION 7.5 (probability recurrence). If i is a nonnegative integer, then p_i is the nonnegative real number defined recursively by

$$p_i = \begin{cases} \epsilon & \text{if } i = 0; \\ 2\epsilon p_{i-1} + (1-2\epsilon)p_{i-1}^2 & \text{if } i > 0. \end{cases}$$

LEMMA 7.6 (two-point distribution). *If i is a nonnegative integer, then the distribution $\mathcal{G}'_i$ maps $1/2^i$ to $1 - p_i$ and maps 0 to p_i.*

Proof. The proof is by induction on i.

Base case ($i = 0$). By the definitions of $\mathcal{G}'_0$ and p_0, we have

$$\mathcal{G}'_0 = \mathrm{Bern}(1-\epsilon) = \mathrm{Bern}(1-p_0),$$

which completes the base case.

Inductive case ($i > 0$). Let b, x, and y be independent random variables such that b has distribution $\mathrm{Bool}(\epsilon)$, and x and y have distribution $\mathcal{G}'_{i-1}$. By the definitions of $\mathcal{G}'_i$ and G' and g', by independence, by the inductive hypothesis, and finally by the definition of p_i, we have

$$\begin{aligned}
\mathcal{G}'_i(0) &= G'(\mathcal{G}'_{i-1})(0) \\
&= \widetilde{g'}(\mathrm{Bool}(\epsilon), \mathcal{G}'_{i-1}, \mathcal{G}'_{i-1})(0) \\
&= \Pr[\, g'(b,x,y) = 0 \,] \\
&= \Pr[\, b \wedge \min(x,y) = 0 \,] + \Pr[\, \bar{b} \wedge \max(x,y) = 0 \,] \\
&= \Pr[\, b \,] \cdot \Pr[\, \min(x,y) = 0 \,] + \Pr[\, \bar{b} \,] \cdot \Pr[\, \max(x,y) = 0 \,] \\
&= \Pr[\, b \,] \cdot \Pr[\, x = 0 \vee y = 0 \,] + \Pr[\, \bar{b} \,] \cdot \Pr[\, x = 0 \wedge y = 0 \,] \\
&= \epsilon[1 - (1-p_{i-1})^2] + (1-\epsilon)p_{i-1}^2 \\
&= 2\epsilon p_{i-1} + (1-2\epsilon)p_{i-1}^2 \\
&= p_i.
\end{aligned}$$

Similarly, we can calculate that $\mathcal{G}'_i(1/2^i) = 1 - p_i$. This calculation completes the induction. $\square$

Next we show that the probabilities p_i are approaching 0.

LEMMA 7.7 (vanishing probability). *If i is a nonnegative integer, then $p_i \leq \epsilon(3\epsilon - 2\epsilon^2)^i$.*

Proof. The proof is by induction on i.

Base case ($i = 0$). This case holds (with equality) from the definition of p_0.

Inductive case ($i > 0$). By the definition of p_i, and by the inductive hypothesis, and because $3\epsilon - 2\epsilon^2 < 1$, we have

$$\begin{aligned} p_i &= 2\epsilon p_{i-1} + (1-2\epsilon)p_{i-1}^2 \\ &\leq 2\epsilon^2(3\epsilon - 2\epsilon^2)^{i-1} + (1-2\epsilon)\epsilon^2(3\epsilon - 2\epsilon^2)^{2(i-1)} \\ &\leq 2\epsilon^2(3\epsilon - 2\epsilon^2)^{i-1} + (1-2\epsilon)\epsilon^2(3\epsilon - 2\epsilon^2)^{(i-1)} \\ &= \epsilon(3\epsilon - 2\epsilon^2)^i. \end{aligned}$$

This inequality completes the induction. ☐

Because $3\epsilon - 2\epsilon^2 < 1$, the preceding lemma implies that $\lim_{i\to\infty} p_i = 0$.

As i approaches infinity, the good news is that $\mathcal{G}'_i$ is mostly concentrated on one value, so that $\mathcal{G}'_i$ has small moments (even relative to its expectation). The bad news is that the expectation of $\mathcal{G}'_i$ approaches 0. Therefore we introduce a hybrid of $\mathcal{G}_i$ and $\mathcal{G}'_i$ that preserves the good news but avoids the bad news.

Choose a nonnegative integer j so that $p_j \leq \frac{1}{2}[(1 - c^{1/d})/(2\epsilon)]^d$. Such a j exists because $\lim_{i\to\infty} p_i = 0$.

DEFINITION 7.8 (hybrid distribution). If i is a nonnegative integer, then the probability distribution $\mathcal{H}_i$ on $\mathbf{R}^+$ is defined recursively by

$$\mathcal{H}_i = \begin{cases} \mathcal{G}'_i & \text{if } i \leq j, \\ G(\mathcal{H}_{i-1}) & \text{if } i > j. \end{cases}$$

As $\mathcal{G}_i \succeq \mathcal{G}'_i$ (Lemma 7.4), and as G is stochastically increasing (Lemma 7.2), it follows (by induction on i) that $\mathcal{G}_i \succeq \mathcal{H}_i$.

8. Tail bound. In this section, we finally prove the tail bound (5.1), completing the proof of our main result on the existence of ϵn-resilient election protocols (for every fixed $\epsilon < \frac{1}{2}$).

It is easy to see, from the definition of g, that

$$g(b, x, y) = \frac{1}{2}(x + y) - \frac{1}{2}[b]\,|x - y|\,,$$

where $[b]$ is the *indicator* of b—namely 1 if b is true and 0 otherwise. From the definition of G, it follows that if D is a distribution on $\mathbf{R}$, then

$$\mathrm{E}(G(D)) = \mathrm{E}(D) - \frac{1}{2}\epsilon\,\mathrm{M}_1(D). \qquad (8.1)$$

(Recall that the expected value of a distribution is just the expected value of a random variable with that distribution.)

We need the following relation between moments. The power mean inequality [12, Theorem 16] implies the inequality

$$\mathrm{M}_\alpha(D)^{1/\alpha} \leq \mathrm{M}_\beta(D)^{1/\beta},$$

whenever $0 < \alpha \leq \beta$.

By Lemmas 7.6 and 7.7, we have

$$\mathrm{E}(\mathcal{G}_i') = (1-p_i)/2^i \geq (1-\epsilon)/2^i \geq 1/2^{i+1} \tag{8.2}$$

and

$$\mathrm{M}_d(\mathcal{G}_i') = 2p_i(1-p_i)/2^{id} \leq 2p_i/2^{id}. \tag{8.3}$$

Next, we show that the expectations of $\mathcal{G}_i$ are bounded away from 0. By replacing $\mathcal{G}_k$ with $\mathcal{H}_k$, iterating the identity (8.1), applying the power mean inequality and the contraction lemma, summing the geometric series, applying the bounds (8.2) and (8.3), and finally plugging in the definition of j, we obtain

$$\begin{aligned}
\mathrm{E}(\mathcal{G}_k) &\geq \mathrm{E}(\mathcal{H}_k)\\
&= \mathrm{E}(\mathcal{H}_j) - \frac{1}{2}\epsilon \sum_{i=j}^{k-1} \mathrm{M}_1(\mathcal{H}_i)\\
&\geq \mathrm{E}(\mathcal{H}_j) - \frac{1}{2}\epsilon \sum_{i=j}^{k-1} \mathrm{M}_d(\mathcal{H}_i)^{1/d}\\
&\geq \mathrm{E}(\mathcal{H}_j) - \frac{1}{2}\epsilon \sum_{i=j}^{k-1} \left(c^{i-j}\,\mathrm{M}_d(\mathcal{H}_j)\right)^{1/d}\\
&\geq \mathrm{E}(\mathcal{H}_j) - \frac{1}{2}\epsilon \frac{\mathrm{M}_d(\mathcal{H}_j)^{1/d}}{1-c^{1/d}}\\
&= \mathrm{E}(\mathcal{G}_j') - \frac{1}{2}\epsilon \frac{\mathrm{M}_d(\mathcal{G}_j')^{1/d}}{1-c^{1/d}}\\
&\geq \frac{1}{2^{j+1}} - \frac{1}{2}\epsilon \frac{(2p_j)^{1/d}/2^j}{1-c^{1/d}}\\
&= \frac{1}{2^{j+1}}\left[1 - \epsilon \frac{(2p_j)^{1/d}}{1-c^{1/d}}\right]\\
&\geq \frac{1}{2^{j+2}}.
\end{aligned} \tag{8.4}$$

Because j is fixed, the expectation is indeed bounded away from 0.

Finally we prove the tail bound (5.1). By the expectation bound (8.4), Chebyshev's inequality [11, section IX.6], the power mean inequality, and the contraction lemma, we have

$$\begin{aligned}
\Pr[\mathcal{G}_k \leq 1/2^{j+3}] &\leq \Pr[\mathcal{G}_k \leq \mathrm{E}(\mathcal{G}_k)/2]\\
&\leq \Pr[|\mathcal{G}_k - \mathrm{E}(\mathcal{G}_k)| \geq \mathrm{E}(\mathcal{G}_k)/2]\\
&\leq \mathrm{Var}(\mathcal{G}_k)/(\mathrm{E}(\mathcal{G}_k)/2)^2\\
&= 2\,\mathrm{M}_2(\mathcal{G}_k)/\,\mathrm{E}(\mathcal{G}_k)^2\\
&\leq 2\,\mathrm{M}_d(\mathcal{G}_k)^{2/d}/\,\mathrm{E}(\mathcal{G}_k)^2\\
&\leq 2c^{2k/d}\,\mathrm{M}_d(\mathcal{G}_0)^{2/d}/\,\mathrm{E}(\mathcal{G}_k)^2\\
&= 2c^{2k/d}[2\epsilon(1-\epsilon)]^{2/d}/\,\mathrm{E}(\mathcal{G}_k)^2\\
&\leq 2c^{2k/d}[2\epsilon(1-\epsilon)]^{2/d}2^{2(j+2)}.
\end{aligned}$$

As k gets large, the right side approaches 0, and in particular becomes less than $1/\binom{n}{\epsilon n}$ for k on the order of n. We thus obtain the tail bound (5.1). As discussed in section 5, this bound implies the existence of an ϵn-resilient election protocol, which proves our main result.

Acknowledgments. We thank Noga Alon for suggesting this problem to us, Nati Linial for some valuable discussions, and the two referees for useful comments.

REFERENCES

[1] M. Ajtai and N. Linial, *The influence of large coalitions*, Combinatorica, 13 (1993), pp. 129–145.

[2] N. Alon and M. Naor, *Coin-flipping games immune against linear-sized coalitions*, SIAM J. Comput., 22 (1993), pp. 403–417.

[3] N. Alon and M. O. Rabin, *Biased coins and randomized algorithms*, in Advances in Computing Research 5: Randomness and Computation, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 499–507.

[4] M. Ben-Or and N. Linial, *Collective coin flipping*, in Advances in Computing Research 5: Randomness and Computation, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 91–115.

[5] M. Ben-Or, N. Linial, and M. E. Saks, *Collective coin flipping and other models of imperfect randomness*, in Proceedings of the 7th Hungarian Colloquium on Combinatorics, Colloquia Mathematica Societatis Janós Bolyai 52, A. Hajnal, L. Lovász, and V. T. Sós, eds., North-Holland, Amsterdam, 1988, pp. 75–112.

[6] R. B. Boppana and B. O. Narayanan, *The biased coin problem*, in Proceedings of the 25th Annual ACM Symposium on the Theory of Computing, 1993, pp. 252–257.

[7] R. B. Boppana and B. O. Narayanan, *The biased coin problem*, SIAM J. Discrete Math., 9 (1996), pp. 29–36.

[8] B. Chor and C. Dwork, *Randomization in Byzantine agreement*, Adv. Comput. Res., 5 (1989), pp. 443–498.

[9] J. Cooper and N. Linial, *Fast perfect-information leader-election protocols with linear immunity*, Combinatorica, 15 (1995), pp. 319–332.

[10] P. Feldman and S. Micali, *An optimal probabilistic protocol for synchronous Byzantine agreement*, SIAM J. Comput., 26 (1997), pp. 873–933.

[11] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1, 3rd ed., John Wiley and Sons, New York, 1968.

[12] G. H. Hardy, J. E. Littlewood, and G. Pólya, *Inequalities*, 2nd ed., Cambridge University Press, Cambridge, UK, 1952.

[13] N. Linial, *Game-theoretic aspects of computing*, in Handbook of Game Theory with Economic Applications, vol. II, R. J. Aumann and S. Hart, eds., North-Holland, Amsterdam, 1994, pp. 1339–1395.

[14] R. Ostrovsky, S. Rajagopalan, and U. V. Vazirani, *Simple and efficient leader election in the full information model*, in Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, 1994, pp. 234–242.

[15] A. Russell and D. Zuckerman, *Perfect information leader election in* $\log^* n + O(1)$ *rounds*, in Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, 1998.

[16] M. E. Saks, *A robust noncryptographic protocol for collective coin flipping*, SIAM J. Discrete Math., 2 (1989), pp. 240–244.

[17] D. Zuckerman, *Randomness-optimal sampling, extractors, and constructive leader election*, in Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, 1996, pp. 286–295.

SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1321–1333

NODE-DISJOINT PATHS ON THE MESH AND A NEW TRADE-OFF IN VLSI LAYOUT[*]

ALOK AGGARWAL[†], JON KLEINBERG[‡], AND DAVID P. WILLIAMSON[†]

Abstract. A number of basic models for VLSI layout are based on the construction of node-disjoint paths between terminals on a multilayer grid. In this setting, one is interested in minimizing both the number of *layers* required and the *area* of the underlying grid. Building on work of Cutler and Shiloach [*Networks*, 8 (1978), pp. 253–278], Aggarwal et al. [*Proc. 26th IEEE Symposium on Foundations of Computer Science*, Portland, OR, 1985; *Algorithmica*, 6 (1991), pp. 241–255], and Aggarwal, Klawe, and Shor [*Algorithmica*, 6 (1991), pp. 129–151], we prove an upper-bound trade-off between these two quantities in a general multilayer grid model. As a special case of our main result, we obtain significantly improved bounds for the problem of routing a full permutation on the mesh using node-disjoint paths; our new bound here is within polylogarithmic factors of the bisection bound. Our algorithms involve some new techniques for analyzing the structure of node-disjoint paths in planar graphs and indicate some respects in which this problem, at least in the planar case, is fundamentally different from its edge-disjoint counterpart.

Key words. combinatorial optimization, VLSI layout, disjoint paths problem

AMS subject classifications. 05C85, 68R10, 68Q20

PII. S0097539796312733

1. Introduction.

The basic node-disjoint paths problem is as follows. We are given a graph G and pairs of vertices in G. The two natural objectives we consider are to

(1) simultaneously route as many pairs as possible on node-disjoint paths in G;
(1′) route all pairs in as few "rounds of communication" as possible, where all paths routed in a single round must be node-disjoint.

Finding node-disjoint paths between designated pairs of vertices in a planar graph is fundamental to a number of routing and embedding problems. In particular, in many models of VLSI layout, effective layout strategies rely on heuristics for finding node-disjoint paths; many current routing software systems rely extensively on such an approach (see also [13]). Node-disjointness constraints can arise as well in the context of virtual circuit routing in networks, when one requires paths to be node-disjoint for the sake of fault tolerance or because of bottlenecks involving switches rather than links.

At the same time, the basic node-disjoint paths problem is NP-hard even for the two-dimensional mesh [10], and very little is known in the way of reasonable approximation algorithms for the problem. Given the basic nature of the problem,

[*]Received by the editors November 27, 1996; accepted for publication (in revised form) November 24, 1997; published electronically February 23, 2000. A preliminary version of this paper appeared in *Proc. 27th ACM Symposium on Theory of Computing,* pp. 585–594.

http://www.siam.org/journals/sicomp/29-4/31273.html

[†]I.B.M. T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (aggarwa@watson.ibm.com, dpw@watson.ibm.com).

[‡]Department of Computer Science, Cornell University, Ithaca, NY 14853 (kleinber@cs.cornell.edu). The research of this author was performed in part while the author was visiting the IBM T. J. Watson Research Center and in part while the author was at the MIT Laboratory for Computer Science MIT, supported by an ONR Graduate Fellowship. The research of this author was also supported in part by an Alfred P. Sloan Research Fellowship and by NSF Faculty Early Career Development Award CCR-9701399.

we are interested in looking for techniques with provable performance guarantees for constructing node-disjoint paths in planar graphs, as such techniques may help in answering questions of the form (1) and (1′) as they arise in practice.

The study of this problem is also interesting to us precisely because its structure, in the case of planar graphs, is very different from the analogous edge-disjoint paths problem. This is true even in a technical sense; for example, work of Kleinberg and Tardos [9] has shown that for many planar graphs including the mesh, a natural *cut condition* is sufficient to within polylogarithmic factors for the feasibility of edge-disjoint paths. But this is far from true in the node-disjoint case, where one encounters a variety of "topological" obstructions to the existence of the paths. See section 1.1 for a discussion of some of these issues.

Let us discuss a very basic problem that arises in this setting. We say a set of terminal pairs in a graph is a *permutation* if each vertex of the graph appears in exactly one pair. If G denotes the $n \times n$ two-dimensional mesh graph, and we are given a permutation in which every pair needs to cross the center column, then the standard bisection bound says that at most n terminal pairs can be routed simultaneously, and so at least $\frac{1}{2}n$ rounds, in the sense of (1′), are required.

The lack of understanding of this area is such that the following has remained open; again, let G denote the mesh:

> (2) For every permutation on G, does there exist a set of $\Omega(n^{1-\varepsilon})$ pairs that can be simultaneously routed on node-disjoint paths for every $\varepsilon > 0$?
>
> (2′) Can every permutation on G be routed in $O(n^{1+\varepsilon})$ rounds for every $\varepsilon > 0$?

Again, by way of contrast, these questions are straightforward to resolve in the affirmative when one requires only edge-disjoint paths.

These latter questions turn out to be closely related to open problems of Aggarwal et al. [1] in the setting of *multilayer* VLSI layout. In multilayer VLSI layouts, typically a *pin* (a *node*, in our terminology) goes through several layers at the same location, and the routing among pins at each layer is planar. If a routing wire needs to change a layer, then a *via* (or *contact cut*) has to be made between two layers, and this usually affects the yield of the output chip (as well as affecting the general quality of the circuit). Consequently, it is practically useful to route as many connections on a single layer as possible (in order to reduce the vias); for more details see [1, 6, 15]. These constraints thus result directly in problems of the type (1) and (1′), with layers now playing the role of "rounds."

One typically reduces the number of layers by introducing spacing between the terminals. In the model of [1, 6], for a parameter d, consider a $dn \times dn$ mesh G_d, with an $n \times n$ grid of terminals at a *uniform spacing* of d. By this we mean that terminals are placed at the set of nodes of G_d whose row and column numbers are multiples of d; thus, $d-1$ paths can now be routed between adjacent terminals. The goal in this model is to find the tightest trade-off possible between the spacing d and the number of layers required in the worst case.

Cutler and Shiloach [6] showed that spacing $d = O(n^2)$ suffices to route all pairs in a single layer; and [1, 2, 8] showed that this value of d was tight up to constant factors. In terms of a general trade-off between d and the minimum number of layers required, the techniques of [6, 1] are sufficient only to show that, for spacing d, $O((\frac{n}{\sqrt{d}})^{4/3})$ layers are sufficient. However, guided by (2′), a reasonable question is the following:

> (3′) With spacing d, are $O((\frac{n}{\sqrt{d}})^{1+\varepsilon})$ layers sufficient to route any

permutation on the mesh, for every $\varepsilon > 0$?

This contains the result of [6] in the case $d = \Theta(n^2)$ and question ($2'$) in the case $d = 1$.

The main results of this paper provide affirmative answers to questions (2), ($2'$), and ($3'$). Specifically, we prove the following theorems.

THEOREM 1.1. *Every permutation on the mesh contains a set of* $\Omega(n/\log n)$ *terminal pairs that can be simultaneously routed on node-disjoint paths. Moreover, every permutation on the mesh can be routed in* $O(n \log^2 n)$ *rounds.*

THEOREM 1.2. *With spacing d, every permutation on the mesh contains a set of* $\Omega(n\sqrt{d}/\log n)$ *pairs that can be routed in one layer, and the entire permutation can be routed in* $O(nd^{-\frac{1}{2}} \log^2 n)$ *layers.*

This strengthens the bounds of [6, 1] on the number of layers required for routing a permutation. Within the context of VLSI, where the number of layers is typically a fixed small number, the "dual" maximization versions of these theorems may in fact be of greater interest. Namely, these are the first algorithms that can provably extract from any permutation on the mesh a routable set of terminal pairs whose size is within polylogarithmic factors of the bisection bound. Moreover, the approach we develop is fundamentally different from that of Cutler and Shiloach [6], which was essentially the only previous method applicable in this setting. Given the basic importance of the problem, we feel that the introduction of new techniques for constructing node-disjoint paths with improved performance guarantees could be of value in suggesting additional heuristic approaches to these problems.

We mentioned above that questions (2) and ($2'$) are easy in the case of edge-disjoint paths; in fact, one can construct a set of paths meeting the bisection bound using only one bend in each path. On the other hand, the paths in the Cutler–Shiloach construction, as well as our constructions in Theorems 1.1 and 1.2, use many bends. Thus it is natural to ask how many rounds are required with a routing that uses only a constant number of bends. Getting $o(n^2)$ rounds is not immediate, and it appears that no previous construction in the literature achieves it. We address this in the following theorem.

THEOREM 1.3. *Every permutation on the mesh can be routed in* $O(n^{7/5})$ *rounds, with each path in the routing using only six bends.*

1.1. The node-disjoint paths problem. For the sake of concreteness, we give some definitions. Throughout this paper, let $G = (V, E)$ denote the $n \times n$ two-dimensional mesh. Let $\mathcal{T} = \{(s_1, t_1), \ldots, (s_m, t_m)\}$ denote a set of *terminal pairs*, where each $s_i, t_i \in V$. A permutation is a set $\mathcal{T}$ of terminal pairs that partitions the vertex set; a *partial permutation* is a set of terminal pairs in which each node appears at most once. These are the only types of sets $\mathcal{T}$ we will be considering.

We say that a subset $\mathcal{T}'$ of $\mathcal{T}$ is *realizable* in G if all pairs in $\mathcal{T}'$ can be simultaneously connected by node-disjoint paths, and we define $\chi(\mathcal{T})$ to be the minimum size of a partition of $\mathcal{T}$ into realizable sets. This corresponds to the minimum number of rounds of communication—or the minimum number of layers—required, in the sense of ($1'$).

In this language, Theorem 1.1 simply says that for any permutation $\mathcal{T}$, we have $\chi(\mathcal{T}) = O(n \log^2 n)$. For the model of [6, 1], with a $dn \times dn$ mesh and terminals evenly spaced at distance d, we use $\chi_d(\mathcal{T})$ to denote the minimum size of a partition of $\mathcal{T}$ into routable sets. Thus our second result is that $\chi_d(\mathcal{T}) = O(nd^{-\frac{1}{2}} \log^2 n)$ for any permutation $\mathcal{T}$.

In addition to connections with VLSI and network routing, we mention that in the context of edge-disjoint paths the quantity $\chi(\mathcal{T})$ can also be viewed as the minimum number of *wavelengths* required to route $\mathcal{T}$ under an optical routing model in which two paths can share an edge as long as they are assigned different wavelengths. Bounds for $\chi(\mathcal{T})$ for permutations $\mathcal{T}$ on a d-dimensional mesh are a result of Raghavan and Upfal [14], and our work here can be seen to provide node-disjoint analogues of some of these bounds. However, optical routing is not, strictly speaking, a motivation of this work, since we do not know of proposals for architectures based on optical routing in which paths sharing a wavelength must remain node-disjoint.

When the number of terminal pairs is restricted to be a fixed constant, and the graph is planar, a polynomial-time algorithm for (1) and (1′) was first given by Robertson and Seymour in [16]. A more practically efficient algorithm for this case was subsequently given by Schrijver [17], who also gave a polynomial-time algorithm for the case in which all terminals lie on the boundaries of a designated set of faces, and the homotopies of the paths with respect to these faces is fixed. This extends work by Cole and Siegel [4] and Leiserson and Maley [12] for the mesh. This approach based on homotopy appears not to be directly applicable to our problem, for although we could test any fixed choice of homotopies for our paths in polynomial time, there are exponentially many choices of homotopies to try.

Let us also indicate some ways in which the node-disjoint paths problem on the mesh differs fundamentally from its edge-disjoint counterpart. First of all, letting $v_{i,j}$ denote the node at row i and column j of the mesh, consider the set $\mathcal{T}$ of consisting of all pairs of the form $(v_{i,1}, v_{n,i})$, $i = 1, \ldots, n$. This set of terminal pairs satisfies the usual cut condition, stating that there is no way to delete k nodes and separate $k+1$ pairs of terminals, and in fact the set $\mathcal{T}$ is routable in one round on edge-disjoint paths. However, since any two paths in a routing of $\mathcal{T}$ must cross, $\mathcal{T}$ requires n rounds when paths must be node-disjoint.

This example actually does not rely on the terminals being positioned near the boundary of the mesh. Consider an infinite grid graph; let $p = m^2$ for some parameter m and let $\mathcal{T}$ denote the set of all pairs of the form $(v_{im+j,0}, v_{jm+i,p})$, with $1 \leq i, j \leq m$. Then it is not hard to show by a crossing number argument (using a lemma of [11]) that $\chi(\mathcal{T}) = \Theta(\sqrt{p})$, although $\mathcal{T}$ is again routable in a single round using edge-disjoint paths.

As a final example, we show that while Theorem 1.3 implies that six bends are sufficient to route any permutation in $O(n^{7/5})$ rounds, there exist partial permutations that require $\Omega(n^2)$ rounds in any one-bend routing.

PROPOSITION 1.4. *There exists a partial permutation $\mathcal{T}$ on G for which there is no one-bend routing using fewer than $\frac{1}{8}n^2$ rounds.*

Proof. Assume for simplicity that n is even; so $n = 2m$. Let $\mathcal{T}$ consists of all pairs of the form $(v_{i,j}, v_{m+j,m+i})$, where $1 \leq i, j \leq m$. Let

$$X = \{v_{i,j} : 1 \leq i \leq m < j \leq 2m\},$$

$$Y = \{v_{i,j} : 1 \leq j \leq m < i \leq 2m\}.$$

Then in any one-bend routing of $\mathcal{T}$, at least $\frac{1}{8}n^2$ of the paths pass though either X or Y, and every pair of such paths must cross at some node.

1.2. Sketch of the algorithm.

We will show that it is possible, with a constant-factor loss in the bound, to reduce the problem for an arbitrary set of terminals to one in which there are two $k \times k$ subsquares of G, denoted A and B, which are spaced $4k$ apart from one another; also, every terminal pair has one end in A and the other in B. If there are p terminal pairs, we will show how to route $p/(k \log k)$ of them simultaneously; applying this process greedily gives the bound of Theorem 1.1.

The approach of Cutler and Shiloach [6] requires, in order to route q terminals in a single round, that the interterminal spacing be at least q. Using this, one can therefore be sure only of routing a set of size $p^{1/3}$, resulting in an overall bound of $n^{4/3}$ for $\chi(\mathcal{T})$.

We start by considering the following simple notion: Suppose we tried choosing at most one terminal from each row of A and at most one from each row of B, and then join as many of these terminals as possible in the region between A and B. There are a number of difficulties with this; in particular, there may be no way to choose terminals in this way so that nearly p/k can be joined in node-disjoint fashion. Let us call this above approach *row-selection*—one tries to pick a unique terminal from each row so that the selected terminals in A can be routed out of A on their rows and then joined with their selected counterparts in B, in such a way that no paths cross.

Our main technique is the following generalization of row selection: Although there may be no good way to choose unique terminals from the rows of A and B, we show that there exists a "rotation" of A and a "rotation" of B, so that the resulting set of rows allows for such a good selection. Now, meshes are discrete objects, so the notion of rotation has to be defined carefully—we use monotone random walks of varying biases as a basic way of "rotating" A and B.

Having chosen a candidate pair of rotations, we decompose A and B into *pseudorows*. The task of *pseudorow selection*—choosing a unique terminal from each pseudorow—is based on analyzing the bipartite graph H on the set of pseudorows, in which a pseudorow of A is joined to a pseudorow of B if there is some terminal pair with one end in each. We show that one can choose rotations of A and B so that this bipartite graph contains a large *monotone matching*—one in which no edges cross. The pairs corresponding to the edges of this matching can be routed in a single round.

With interterminal spacing of $d > 1$, a very similar approach is applicable. The main difference is that we can now afford to choose edge sets of the bipartite graph H which are not necessarily monotone matchings; we can handle a limited amount of nonmonotonicity using the technique of Cutler–Shiloach within each pseudorow.

2. Preliminaries.

In this section we develop some basic lemmas that will be useful in analyzing our algorithms. In section 3 we consider the basic problem of a permutation on the mesh, and we prove Theorem 1.1. In section 4 we give our algorithm for routing with interterminal spacing in the model of [6, 1]. Finally, in section 5 we discuss algorithms for routing with only a constant number of bends per path.

LEMMA 2.1. *Let $x_1, \ldots, x_d$ be positive integers for which $\sum_i x_i = a$ and $\sum_i x_i^2 = b$. Then $d \geq a^2 b^{-1}$.*

Proof. $a^2 = (\sum_i x_i)^2 \leq d \sum_i x_i^2 = db$. □

The following is easily proved by induction on b.

LEMMA 2.2. *For nonnegative integers a and b, one has*

$$\int_0^1 x^a (1-x)^b \, dx = (a+b+1)^{-1} \binom{a+b}{b}^{-1}.$$

LEMMA 2.3. *Consider the following random experiment: We first choose a bias α for a coin uniformly from the interval $[0, 1]$ and then flip the coin n times. The probability that the number of heads is k is equal to $\frac{1}{n+1}$.*

Proof. The probability is equal to

$$\binom{n}{k} \int_0^1 \alpha^k (1-\alpha)^{n-k} \, d\alpha,$$

which by Lemma 2.2 is equal to $\frac{1}{n+1}$. ∎

There is also a purely combinatorial proof of this, which we leave as an exercise for the reader.

2.1. Monotone matchings. We make use of a notion that we call a monotone matching. Here we define and develop some basic facts about monotone matchings at a general level.

In this section, let H denote a bipartite graph with bipartition (U, V), $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, and let e denote the number of edges of H. As usual, by a *matching* in H we mean a set of edges that have no endpoints in common, and we let $\mu(H)$ denote the size of the largest matching in H.

DEFINITION 2.4. *A matching in H is said to* monotone *if it consists of edges $(u_{i_1}, v_{j_1}), \ldots, (u_{i_k}, v_{j_k})$ such that the sequence $i_1, \ldots, i_k$ is increasing and the sequence $j_1, \ldots, j_k$ is either increasing or decreasing. We say the matching is* increasing *in the former case and* decreasing *in the latter.*

We give two lower bounds on the size of the largest monotone matching in H.

LEMMA 2.5. *H has a monotone matching of size at least $\sqrt{\mu(H)}$.*

Proof. Let M be a matching of maximum size in H, and order its edges so that their endpoints in U have increasing indices. Consider the resulting sequence of indices in V. By a lemma due to Erdös and Szekeres [7], this sequence must have either an increasing sequence or a decreasing sequence of length at least $\sqrt{|M|} = \sqrt{\mu(H)}$. The edges corresponding to this sequence constitute a monotone matching. ∎

LEMMA 2.6. *H has an increasing monotone matching of size at least $e/2n$.*

Proof. Let M_k denote the set of all edges (u_i, v_j) for which $j = i + k \bmod n$. Let M_k' denote the subset of M_k in which $j \geq i$ and M_k'' the subset of M_k in which $j < i$. Now, each M_k' and M_k'' is a monotone matching, and since they are all disjoint one has size at least $e/2n$. ∎

An example by Coppersmith [5] shows that for every $d \leq n$ there exist d-regular bipartite graphs on $2n$ nodes with no monotone matching of size greater than $O(\max(d, \sqrt{n}))$.

Finally, it is also useful to establish "covering" versions of these lemmas. The edges of a bipartite graph of maximum degree Δ can be partitioned into Δ matchings; from the proof of Lemma 2.5, it follows that any matching can be partitioned into $O(\sqrt{n})$ monotone matchings. Thus we have the following lemma.

LEMMA 2.7. *The edges of H can be covered by $O(\Delta\sqrt{n})$ monotone matchings, where Δ is the maximum degree of H.*

The monotone matchings constructed in the proof of Lemma 2.6 cover the edges of H; thus we have the following.

LEMMA 2.8. *The edges of H can be covered by $2n$ increasing monotone matchings.*

3. The bound for $\chi(\mathcal{T})$.

For vertices $x, y \in G$, let $d(x, y)$ denote the shortest-path distance between them. As indicated above, we will first consider the following special case: There are $k \times k$ subsquares A and B of G, with $d(A, B) \geq 4k$, each terminal pair has one end in A and the other in B, and we must use paths that stay

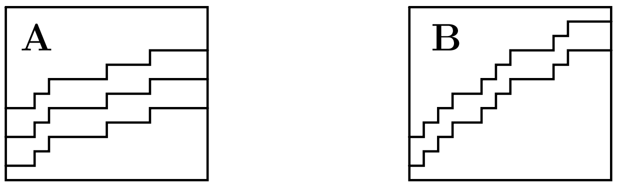


FIG. 3.1. *Decomposing into pseudorows.*

within $d(A,B)$ of $A \cup B$. Below we show how a bound for this special case can be used to prove Theorem 1.1.

We represent A as a union of pseudorows, as follows. Let v_1 denote the lower left vertex of A. We choose $\alpha \in [0,1]$ uniformly at random and perform the following random walk for k time steps starting from v_1:

(i) With probability $1-\alpha$, move one step to the right.

(ii) With probability α, move one step up and then one step to the right.

Let L_1^α denote the path traversed by this walk; note that it is contained in A. For $-k \le i \le k$, define L_i^α to be a copy of this path translated $(i-1)$ steps upward and then intersected with A. The sets L_i^α will be called pseudorows.

See Figure 3.1 for an example of such pseudorows.

For $x \in A$, let $R_x^\alpha = \{i : x \in L_i^\alpha\}$. The following is immediate from the construction but worth noting.

LEMMA 3.1. *If $v \in A$ belongs to both L_i^α and L_j^α, then $|j-i| = 1$. Thus $|R_v^\alpha| \le 2$ (v belongs to at most two of these sets) and, in particular, $E|R_v^\alpha| \le 2$.*

We say that $x, y \in A$ are *collinear* if there is some pseudorow that contains them both; in this case we write $x \sim y$.

For a vertex $x \in G$, let $a(x)$ denote its row number and $b(x)$ denote its column number. For two vertices x and y, let $d_\infty(x,y)$ denote the L_∞ distance between them; that is,

$$d_\infty(x,y) = \max\left(|a(x)-a(y)|, |b(x)-b(y)|\right).$$

LEMMA 3.2. *The probability that $x, y \in A$ are collinear is at most $\frac{2}{d_\infty(x,y)}$.*

Proof. Suppose without loss of generality that $b(x) \le b(y)$. We have

$$\begin{aligned}
Pr(x \sim y) &= \int_0^1 Pr(x \sim y \mid \alpha = t)\, dt \\
&\le \int_0^1 \sum_i Pr(x, y \in L_i^t)\, dt \\
&= \int_0^1 \sum_i Pr(x \in L_i^t) \cdot Pr(y \in L_i^t \mid x \in L_i^t)\, dt \\
&= \int_0^1 Pr(y \in L_i^t \mid x \in L_i^t) \cdot \sum_i Pr(x \in L_i^t)\, dt \\
&= \int_0^1 Pr(y \in L_i^t \mid x \in L_i^t) \cdot E|R_x^t|\, dt \\
&\le 2\int_0^1 Pr(y \in L_i^t \mid x \in L_i^t)\, dt.
\end{aligned}$$

If it is not the case that $b(y) - b(x) \geq a(x) - a(y) \geq 0$, then this last integral is 0. Otherwise, it is simply the probability, over a uniformly distributed bias, that a given number of heads will come up when a coin with that bias is flipped $d_\infty(x, y)$ times. By Lemma 2.3, this is $\frac{1}{1+d_\infty(x,y)}$. The lemma follows. $\square$

LEMMA 3.3. *There is an absolute constant c' so that if there are p terminal pairs with ends in A and B, then there is a realizable subset of size at least $p/(c'k \log k)$.*

Proof. We first choose a random α and decompose A into pseudorows. We do the same for B, using a random β. Now, by Lemma 3.1 we can choose half the pseudorows of A and half the pseudorows of B so that all pseudorows are disjoint and at least $q = p/4$ of the terminal pairs have both ends in these chosen pseudorows. Let $\mathcal{T}'$ denote a set of q such pairs.

Define a *collineation* to be a pair i, j such that s_i and s_j are collinear in A and t_i and t_j are collinear in B. We wish to bound the expected number of collineations among the pairs in $\mathcal{T}'$; this is simply the sum

$$\sum_{i,j} Pr(s_i \sim s_j) \cdot Pr(t_i \sim t_j).$$

By the Cauchy–Schwarz inequality, this dot product will be maximized if $Pr(s_i \sim s_j) = Pr(t_i \sim t_j)$ for all pairs i, j.

Thus the expected number of collineations is bounded by the largest possible sum

$$\sum_{x,y \in S} Pr(x \sim y)^2,$$

where S is a subset of A of size q. We can write this sum as

$$\frac{1}{2} \sum_{x \in S} \sum_{y \in S \setminus \{x\}} Pr(x \sim y)^2 \leq \frac{1}{2} \sum_{x \in S} \sum_{y \in S \setminus \{x\}} \frac{4}{d_\infty(x,y)^2},$$

with the inequality following from Lemma 3.2. This inner sum is maximized by having the vertices y fill in an L_∞ ball centered at x, in which case its value is bounded by

$$\sum_{i=1}^{\frac{1}{2}\sqrt{q}} \frac{32i}{i^2} \leq 32 + 16 \ln q \leq c \ln q$$

for an absolute constant c. Thus, summing over all x, we see that the expected number of collineations in $\mathcal{T}'$ is at most $\frac{1}{2} cq \ln q$.

Thus there exist α, β and decompositions of A and B into pseudorows for which there is a subset $\mathcal{T}'$ of q terminal pairs whose number of collineations is at most $\frac{1}{2} cq \ln q$. For this set $\mathcal{T}'$, let us build the following bipartite multigraph H. There is one vertex u_i on the left for each chosen pseudorow A and one vertex v_j on the right for each chosen pseudorow B. We now put w_{ij} parallel edges from u_i to v_j if there are w_{ij} terminals with one end in the pseudorow u_i and the other end in v_j. From H, we build a simple graph H' which contains an edge (u_i, v_j) iff $w_{ij} > 0$.

We claim that the number of edges of H' is at least $q/(2c \ln q)$. To see this, note that the number of collineations in $\mathcal{T}'$ is precisely

$$\sum_{i,j} \binom{w_{ij}}{2},$$

whence we have $\sum_{i,j} w_{ij}^2 \leq 2cq \ln q$. Since $q = |T'| = \sum_{i,j} w_{ij}$, the claim follows from Lemma 2.1.

Finally, the number of vertices in each half of the bipartition of H' is at most k, so by Lemma 2.6 H' has a monotone matching of size at least

$$\frac{q}{4ck \ln q} \geq \frac{p}{16ck \ln q} \geq \frac{p}{32ck \log k}.$$

We can route one terminal for each edge of the monotone matching, which implies the lemma with $c' = 32c$. Note that we need $d(A, B) = \Omega(k)$ in order to ensure that we correctly align the matched pseudorows of A with the corresponding pseudorows of B. ∎

LEMMA 3.4. *All terminal pairs with ends in A and B can be routed in $O(k \log^2 k)$ rounds.*

Proof. We produce the partition into realizable sets greedily, using Lemma 3.3. The bound follows since the number of remaining terminals goes down by a $(1 - \frac{c'}{k \log k})$ fraction for each realizable set we produce. ∎

Proof of Theorem 1.1. For an integer τ, let $\mathcal{T}_\tau$ denote the set of terminal pairs (s_i, t_i) for which $\frac{1}{2}\tau \leq d(s_i, t_i) \leq \tau$. We prove that $\mathcal{T}_\tau$ can be routed in $O(\tau \log^2 \tau)$ rounds; the overall bound on the number of rounds then follows since, for an absolute constant c'',

$$\begin{aligned} \chi(\mathcal{T}) &\leq \sum_{i=0}^{\lceil \log n \rceil} \chi(\mathcal{T}_{2^i}) \\ &\leq \sum_{i=0}^{\lceil \log n \rceil} c'' 2^i \log^2(2^i) \\ &\leq 4c'' n \log^2 n. \end{aligned}$$

To prove that $\chi(\mathcal{T}_\tau) = O(\tau \log^2 \tau)$, we show that $\mathcal{T}_\tau$ can be partitioned into a constant number of sets, and each of these sets can be partitioned into pairs of well-separated squares; we then invoke the algorithm of Lemma 3.4 on all the pairs in a single set simultaneously.

Choose $\tau' = \frac{1}{16}\tau$ and define S_{ij} to be the $\tau' \times \tau'$ subsquare of G whose upper-left corner lies in row $i\tau'$ and column $j\tau'$. Each pair of squares $(S_{ij}, S_{\ell m})$ defines a subset of $\mathcal{T}_\tau$, consisting of those terminal pairs with one end in S_{ij} and the other in $S_{\ell m}$. We call such a pair *active* if it contains a terminal pair in $\mathcal{T}_\tau$. By definition, any active pair $(S_{ij}, S_{\ell m})$ satisfies

$$6 \leq |\ell - i| + |m - j| \leq 16.$$

Thus, by Lemma 3.4, we can route all the terminal pairs in the set defined by any active pair in $O(\tau \log^2 \tau)$ rounds.

We say that active pairs $(S_{ij}, S_{\ell m})$ and $(S_{i'j'}, S_{\ell' m'})$ *interfere* with one another if there is some vertex within distance τ of both $S_{ij} \cup S_{\ell m}$ and $S_{i'j'} \cup S_{\ell' m'}$. Let us build a graph $\mathcal{K}$ on the set of active pairs, joining two by an edge if they interfere with one another. This graph has constant degree and hence can be colored with a constant number of colors. But we can use the same set of rounds for routing *all* the active pairs in a single color class, and thus $\chi(\mathcal{T}_\tau) = O(\tau \log^2 \tau)$.

The proof of the other statement of Theorem 1.1, that every permutation contains a routable set of size $\Omega(n/\log n)$, is strictly analogous. We first argue, using the interference graph $\mathcal{K}$ and Lemma 3.3 instead of Lemma 3.4, that $\mathcal{T}_\tau$ contains a routable set of size $\Omega(|\mathcal{T}_\tau|/\tau \log \tau)$. The bound now follows since for some $\tau \in \{2^i : 0 \le i \le \lceil \log n \rceil\}$ we have $|\mathcal{T}_\tau| = \Omega(n\tau)$. □

4. Terminals with spacing. The proof of Theorem 1.2 is essentially that of Theorem 1.1, with one additional step. As before, we can specialize to the case of two $kd \times kd$ squares A and B that contain p terminal pairs. To construct pseudorows, we first partition A and B into disjoint $d \times d$ subsquares, contract these subsquares, and then use the previous construction.

LEMMA 4.1. *There is an absolute constant c' so that if there are p terminals with ends in A and B, then there is a realizable subset of size at least $p\sqrt{d}/(c'k \log k)$.*

Proof. We follow the proof of Lemma 3.3 up through the construction of the bipartite graph H' with at most k vertices in each half of its bipartition, and with at least $q/(c \ln q)$ edges. Now define a *slice* in H' to be a set of the form $U' \cup V'$, where U' is a set of $\sqrt{d}$ consecutive vertices on the left side of H' and V' is a set of $\sqrt{d}$ consecutive vertices on the right side. The crucial point is that there are only d edges of H' with both ends in a given slice, and so using the technique of [1, 6], we can route one terminal pair for each of these edges in a single round.

We now construct a graph H'' by identifying blocks of $\sqrt{d}$ consecutive vertices of H' on the left and on the right. (We can essentially think of edges of H'' as corresponding to particular slices of H'.) Let us give each edge (u^*, v^*) in H'' a weight equal to the number of edges of H' that have ends in the supernodes u^* and v^*; thus these weights are integers between 0 and d. By the weighted analogue of Lemma 2.6, H'' has a monotone matching of weight at least

$$\frac{\text{weight}(H'')}{2|V(H'')|} = \frac{|E(H')|}{2|V(H'')|} \ge \frac{q\sqrt{d}\log k}{4ck \ln q} \ge \frac{p\sqrt{d}}{32ck \log k}.$$

We can route a number of terminals equal to the weight of any monotone matching in H'', as argued above; this implies the lemma. □

Proof of Theorem 1.2. Again reducing to the case of $kd \times kd$ squares A and B, we use a greedy approach based on Lemma 4.1. Each time, we reduce the number of remaining terminals by a factor of at least $(1 - \frac{c'\sqrt{d}}{k \log k})$; thus we route all terminal pairs in $O(kd^{-\frac{1}{2}} \log^2 k)$ rounds. □

5. Constant-bend routing. As in the previous sections, let us first consider the case in which all terminal pairs have ends in $k \times k$ squares A and B, with $d(A, B) \ge 4k$. Here, we additionally assume that A and B are each at a distance of at least $\Omega(k^{7/10})$ from the boundary of the mesh. Let us suppose that the lowest column number of a node in A is less than or equal to the lowest column number of a node in B.

The main difficulty we encounter here is this: Since we wish to use only a constant number of bends, the approach based on pseudorows appears to be inherently inapplicable, as it involves partitioning A and B into paths with many bends. Thus, in this case, we build a graph H with a vertex u_i for each row of A, and a vertex v_j for each row of B. We give the edge (u_i, v_j) a weight w_{ij} equal to the number of terminal pairs with one end in each row.

To give a sense of the algorithm, we first mention a very easy algorithm for routing in $O(k^{3/2})$ rounds. This is as follows. First replace each edge (u_i, v_j) of H with w_{ij} parallel edges. Now by Lemma 2.7 the edges of H can be covered by $O(k^{3/2})$ monotone matchings. Moreover, we may assume that each matching in the cover contains at most $O(k^{1/2})$ edges. We claim that all the terminal pairs corresponding to a single one

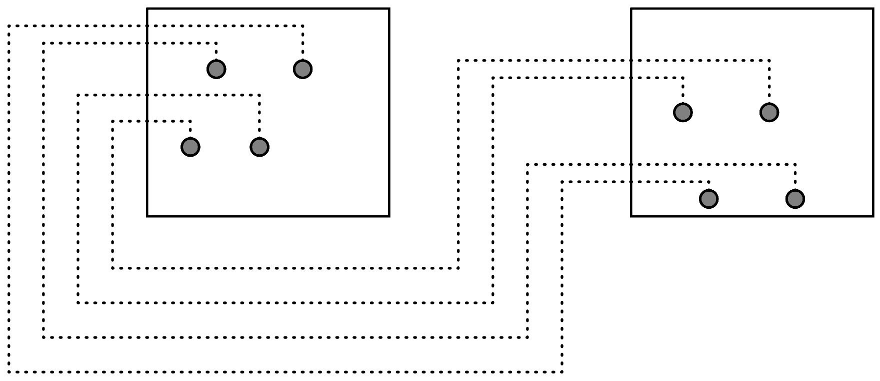


Fig. 5.1. *Routing with a constant number of bends.*

of these matchings can be routed in a single round. If the matching is increasing, then the pairs can be routed by having each terminal leave A along its row through the east wall, and enter B along the matching row through the west wall; if the matching is decreasing, then the pairs can be routed by having each terminal leave A along its row through the west wall and enter B along the matching row through the west wall. (See Figure 5.1 for a version of the latter case.) Moreover, since A and B are at least $\Omega(k^{1/2})$ distance from the boundary of the mesh, there is room for all the paths corresponding to a single matching to be routed in the region outside $A \cup B$.

To get a bound of $O(k^{7/5})$, we need a slight strengthening of this technique. We divide the edges into $\log k$ classes, by assigning each edge to the smallest power of 2 greater than its weight. Let H_w denote the subgraph of H consisting only of edges that are assigned the weight w, and let $\mathcal{T}_w$ denote the set of terminals corresponding to edges of H_w.

Lemma 5.1. *$\mathcal{T}_w$ can be routed in at most $2kw$ rounds, using at most four bends.*

Proof. Using Lemma 2.8, we can partition the edges of H_w into $2k$ (increasing) monotone matchings. Each edge of H_w corresponds to at most w terminals, so all terminals can be routed in $2kw$ rounds. It is easy to construct paths for these terminals using at most four bends. □

Lemma 5.2. *$\mathcal{T}_w$ can be routed in at most*

$$\max[O(k^{3/2}/w^{1/4}), O(k^{13/10})]$$

rounds, using at most six bends.

Proof. Since each edge of H_w corresponds to at least $w/2$ terminals, the maximum degree of a vertex in H_w is at most $2k/w$; thus we can partition the edges of H_w into $2k/w$ matchings. Let us say that a matching M is *well spaced* if for edges (u_i, v_j) and (u_k, v_ℓ) of M the difference between i and k is at least $\sqrt{w}$ and the distance between j and ℓ is at least $\sqrt{w}$. By an easy coloring argument, we can partition any matching in H_w into $O(\sqrt{w})$ well-spaced matchings, and thus we can partition the edges of H_w into $O(k/\sqrt{w})$ well-spaced matchings. Finally, each of these well-spaced matchings is only on $O(k/\sqrt{w})$ vertices, so by Lemma 2.7 we can partition each one into $O(\sqrt{k}/w^{1/4})$ monotone matchings.

Thus we now have a total of $O(k^{3/2}/w^{3/4})$ monotone matchings covering the edges of H_w, and each has the property that the endpoints of its edges are mutually at distance $\sqrt{w}$. We use a different set of rounds for routing each monotone matching.

Now, how many rounds are needed for a single monotone matching M? (Recall that each edge of M represents up to w terminals.) Let A_i (resp., B_i) denote the set of vertices in the ith row of A (resp., B). Since the edges of M are anchored in rows of A and B that are mutually $\sqrt{w}$ apart, we can assign to each edge $e = (u_i, v_j)$ a "band" of rows of width $\Omega(\sqrt{w})$ around row A_i and a similar band around row B_j. The pairs corresponding to an edge e are themselves a matching on the vertex set $A_i \cup B_j$. Thus these pairs can be partitioned, again by Lemma 2.7, into $O(\sqrt{w})$ monotone matchings, and each of *these* monotone matchings can be routed in a single round in the bands surrounding rows A_i and B_j.

Finally, we can do this simultaneously (i.e., using the same set of rounds) for all edges of a single well-spaced monotone matching, provided there is sufficient room between the squares A and B and the boundary of the mesh for routing all the paths. If there is sufficient room, then the number of rounds used is $O(k^{3/2}/w^{1/4})$. Otherwise, we must break up the rounds that use too many paths into subrounds that route only $O(k^{7/10})$ terminals at a time; the resulting bound is $O(k^{13/10})$ rounds. ∎

LEMMA 5.3. *The set of terminals with ends in A and B can be routed in $O(k^{7/5})$ rounds, using at most six bends.*

Proof. Let $r(\mathcal{T})$ denote the total number of rounds required for routing $\mathcal{T}$ with six bends. Then

$$\begin{aligned} r(\mathcal{T}) &\leq \sum_{w=2^i} r(\mathcal{T}_w) \\ &\leq \sum_{w=2^i \leq k^{2/5}} r(\mathcal{T}_w) + \sum_{k^{2/5} \leq w=2^i \leq k^{4/5}} r(\mathcal{T}_w) + \sum_{w=2^i \geq k^{4/5}} r(\mathcal{T}_w) \\ &\leq O(k) \sum_{w=2^i \leq k^{2/5}} w + O(k^{3/2}) \sum_{k^{2/5} \leq w=2^i \leq k^{4/5}} w^{-1/4} + O(k^{13/10}) \sum_{w=2^i \geq k^{4/5}} 1. \end{aligned}$$

Since all three of the terms in this last sum are easily seen to be $O(k^{7/5})$, the bound follows. ∎

Proof of Theorem 1.3. We use the notion of an interference graph $\mathcal{K}$ on pairs of subsquares as in the proof of Theorem 1.1. The only change is that we can now work only with $k \times k$ squares that have either no row or no column within $O(k^{7/10})$ of the boundary. (It is enough to have this guarantee for either rows *or* columns, since we just as well could have built our graph H on the set of columns of one of the squares.) To handle this, we define G' to be the subgraph of G induced by all nodes within distance $O(n^{2/5})$ of the boundary, or within distance $O(n^{7/10})$ of a corner of the mesh. Since $|G'| = O(n^{7/5})$, we can use a different round for each terminal pair with an end in G'; we then run the algorithm above (using the interference graph $\mathcal{K}$) in $G \setminus G'$. ∎

6. Conclusion.

There are a number of natural questions suggested by this work. First, the algorithm of Theorem 1.2 routes a permutation with spacing d in $O(nd^{-1/2} \log^2 n)$ rounds and thereby gives an upper-bound trade-off between inter-terminal spacing and the number of rounds required. However, we do not know how close to optimal this trade-off is for all values of d. At the extreme ends of the range $0 \leq d \leq n^2$, the number of rounds used by our algorithm is tight to within polylogarithmic factors—this follows from the bisection bound when $d = \Theta(1)$ and from [1] when $d = \Theta(n^2)$. While it is possible that the trade-off is tight to within polylogarithmic factors for all values of d, we are not able to prove this.

For constant-bend routing, we do not know whether there is an absolute constant b such that every permutation on the mesh can be routed in $O(n \log^{O(1)} n)$ rounds using at most b bends per path. It would also be interesting to understand the quantitative trade-off between the number of bends and the number of rounds needed.

In a somewhat different direction, the algorithm of Theorem 1.1 gives a routing for a set $\mathcal{T}$ of terminal pairs for which the number of rounds used is close to the best possible in the worst case. However, it is not an *approximation algorithm*, since we do not have a result relating the number of rounds we require for a given set $\mathcal{T}$ to the optimal value $\chi(\mathcal{T})$. It would be very interesting to obtain such an algorithm, since it would appear to require new techniques for analyzing $\chi(\mathcal{T})$ in cases when the cut condition gives no information.

Finally, it would be interesting to decide whether $\chi(\mathcal{T}) = O(n)$ for every permutation on the mesh.

Acknowledgments. We would like to thank Don Coppersmith, Tom Leighton, Baruch Schieber, and Madhu Sudan for valuable discussions on these problems.

REFERENCES

[1] A. Aggarwal, M. Klawe, D. Lichtenstein, N. Linial, and A. Wigderson, *Multi-layer grid embeddings*, in Proc. 26th IEEE Symposium on Foundations of Computer Science, Portland, OR, 1985.

[2] A. Aggarwal, M. Klawe, D. Lichtenstein, N. Linial, and A. Wigderson, *A lower bound on the area of permutation layouts*, Algorithmica, 6 (1991), pp. 241–255.

[3] A. Aggarwal, M. Klawe, and P. Shor, *Multi-layer grid embeddings for VLSI*, Algorithmica, 6 (1991), pp. 129–151.

[4] R. Cole and A. Siegel, *River routing every which way but loose*, in Proc. 25th IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984.

[5] D. Coppersmith, *private communication*, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1995.

[6] M. Cutler and Y. Shiloach, *Permutation layout*, Networks, 8 (1978), pp. 253–278.

[7] P. Erdös and G. Szekeres, *A combinatorial problem in geometry*, Compositio Math., 2 (1935), pp. 463–470.

[8] M. Klawe and T. Leighton, *A tight lower bound on the size of planar permutation networks*, SIAM J. Discrete Math., 5 (1992), pp. 558–563.

[9] J. Kleinberg and É. Tardos, *Disjoint paths in densely embedded graphs*, in Proc. 36th IEEE Symposium on Foundations of Computer Science, Milwaukee, WI, 1995.

[10] M. E. Kramer and J. van Leeuwen, *The complexity of wire routing and finding the minimum area layouts for arbitrary VLSI circuits*, in Advances in Computing Research 2: VLSI Theory, F. P. Preparata, ed., JAI Press, London, 1984, pp. 129–146.

[11] F. T. Leighton, *Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI*, Ph.D. thesis, MIT Math. Dept., Cambridge, MA, 1981.

[12] C. Leiserson and F. M. Maley, *Algorithms for routing and testing routability of planar VLSI layouts*, in Proc. 17th ACM Symposium on Theory of Computing, Providence, RI, 1985.

[13] W. R. Pulleyblank, *Two Steiner tree packing problems*, invited talk at ACM Symposium on Theory of Computing, Las Vegas, NV, 1995.

[14] P. Raghavan and E. Upfal, *Efficient all-optical routing*, in Proc. 26th ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, 1994.

[15] R. Rivest, *The placement and interconnect system*, in Proc. 19th Design and Automation Conference, Las Vegas, NV, 1982.

[16] N. Robertson and P. D. Seymour, *Graph minors* VII*, Disjoint paths on a surface*, J. Combin. Theory Ser. B, 45 (1988), pp. 212–254.

[17] A. Schrijver, *Disjoint homotopic paths and trees in a planar graph*, Discrete Comput. Geom., 6 (1991), pp. 527–574.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1334–1370

QUALITY MESH GENERATION IN HIGHER DIMENSIONS*

SCOTT A. MITCHELL† AND STEPHEN A. VAVASIS‡

Abstract. We consider the problem of triangulating a d-dimensional region. Our mesh generation algorithm, called QMG, is a quadtree-based algorithm that can triangulate any polyhedral region including nonconvex regions with holes. Furthermore, our algorithm guarantees a bounded aspect ratio triangulation provided that the input domain itself has no sharp angles. Finally, our algorithm is guaranteed never to overrefine the domain, in the sense that the number of simplices produced by QMG is bounded above by a factor times the number produced by any competing algorithm, where the factor depends on the aspect ratio bound satisfied by the competing algorithm. The QMG algorithm has been implemented in C++ and is used as a mesh generator for the finite element method.

Key words. triangulation, aspect ratio, tetrahedra, mesh generation, polyhedron

AMS subject classifications. 65N50, 65Y25

PII. S0097539796314124

1. Introduction.

The *finite element method* refers to a family of numerical methods for solving boundary value problems and is used extensively in electromagnetics, thermodynamics, structural analysis, acoustics, chemistry, and astronomy. A crucial preprocessing step is mesh generation. A *mesh generator* is an algorithm for subdividing a finite subset of $\mathbf{R}^2$ or $\mathbf{R}^3$ into small convex cells, typically triangles or quadrilaterals in two dimensions and tetrahedra or hexahedra (brick shapes) in three dimensions.

We propose a mesh generation algorithm called QMG for nonconvex polyhedral regions in any dimension. QMG takes as input a representation of a polyhedral region in $\mathbf{R}^d$ and produces as output a simplicial complex that is a subdivision of the input region. QMG uses a quadtree technique: the domain is covered with a large d-dimensional cube, and then cubes are recursively split into 2^d subcubes until each subcube is triangulated.

For good accuracy bounds in the finite element method, it is necessary that the tetrahedra have bounded *aspect ratio.* The aspect ratio of a simplex is defined as its maximum side length divided by its minimum altitude. For an analysis of the accuracy of the finite element method, see Johnson [9].

The mesh produced by QMG is guaranteed to have good aspect ratio. Let ρ_{QMG} be the worst aspect ratio among all simplices in the QMG triangulation of a particular input polyhedron P. Let ρ_S be the worst aspect ratio among all simplices in any other

*Received by the editors December 27, 1996; accepted for publication (in revised form) May 28, 1999; published electronically February 29, 2000. This work supported in part by an NSF PYI award with matching funds from AT&T, Sun Microsystems, Tektronix, and Xerox. Support was also received from the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under contract W-31-109-Eng-38 through Argonne National Laboratory. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Copyright is owned by SIAM to the extent not limited by these rights.

http://www.siam.org/journals/sicomp/29-4/31412.html

†Computational Mechanics and Visualization Department, Sandia National Laboratories, Albuquerque, NM 87185 (samitch@sandia.gov).

‡Department of Computer Science, Cornell University, Ithaca, NY 14853 (vavasis@cs.cornell.edu). Part of this work was done while the author was visiting Argonne National Laboratory.

triangulation $\mathcal{S}$ of P, where $\mathcal{S}$ is produced by some other competing algorithm. Then Theorem 15.2 says that $\rho_{\mathrm{QMG}} \leq c\rho_{\mathcal{S}}$, where c is a universal constant, in the case $d = 2$ or $d = 3$. The technique used to prove this theorem is as follows. First, a lower bound is proved stating that any triangulation $\mathcal{S}$ of P must have at least one simplex with aspect ratio at least as large as $c/\theta(P)$, where $\theta(P)$ denotes the sharpest angle of P and c is some other constant. Then we prove that QMG's aspect ratio is bounded above by $c/\theta(P)$. In the case $d > 3$, a weaker version of this result is proved.

Our second main theorem is that the number of simplices generated by QMG is the smallest possible; that is, the mesh is as coarse as possible, in the following sense. Let n_{QMG} be the number of simplices produced by QMG when applied to a particular polyhedral domain P, and let $n_{\mathcal{S}}$ be the number of simplices in some other triangulation $\mathcal{S}$ of P. Then $n_{\mathrm{QMG}} \leq f(d, \rho_{\mathcal{S}}) \cdot n_{\mathcal{S}}$, where f is some function of d, the dimension, and of $\rho_{\mathcal{S}}$, the aspect ratio bound satisfied by the competing triangulation. In other words, n_{QMG} is much larger than $n_{\mathcal{S}}$ only in the case when $\mathcal{S}$ has simplices with poor aspect ratio. The precise values of the constants present in these two main results are not worked out explicitly in this article but are expected to be quite large.

The importance of bounding the number of tetrahedra is as follows. The running time of the finite element method is a function of the number of nodes and elements in the triangulation. In particular, if n is the number of nodes (or elements—for bounded aspect ratio triangulations, the number of nodes and elements are within a constant factor of each other), then the running time of the finite element method is $O(n^{\alpha})$, where α is at least 1 and depends on the method used for solving the sparse linear equations. Thus, there is a significant penalty for meshes with too many elements. On the other hand, small elements are necessary for high accuracy with the finite element method. Practitioners usually address this tradeoff by using meshes with varying degrees of refinement: such a mesh has small elements in the part of the domain of interest where high accuracy is desired, and larger elements are used elsewhere. Because QMG generates the coarsest mesh possible (up to the multiplicative factor $f(d, \rho_{\mathcal{S}})$), it can be used as the starting point for further refinement. Indeed, the implementation of QMG allows a user-specified refinement function to control the degree of refinement.

Our work is closely related to earlier work by Bern, Eppstein, and Gilbert [4], who solved the corresponding problem for two-dimensional (2D) polygonal domains. These authors also used a quadtree approach, but the extension of their technique to higher dimensions is far from straightforward. Our QMG algorithm differs in many ways from that earlier paper.

Other work on triangulation problems with optimality guarantees is the result of Baker, Grosse, and Rafferty [1], whose algorithm triangulates 2D polygons with nonobtuse angles and Chew's [7] triangulation of 2D polygons with guaranteed aspect ratio using a Delaunay approach. Chew's work was extended by Ruppert [14] to handle varying degrees of refinement (and thus establishing the Bern et al. optimality properties), and later by Chew also [8] to curved surfaces.

In three dimensions, no work previous to ours guaranteed bounded aspect ratio triangulations, although Chazelle and Palios [6] developed an algorithm with the best possible bound (up to a constant factor) on the cardinality of the triangulation in terms of reflex angles.

Our triangulation uses *Steiner points*, meaning that it introduces new vertices into the domain not present in the original input. Indeed, as shown by Schönhardt [15], Steiner points are necessary for triangulating nonconvex polyhedra in three dimensions and higher. For additional background on optimal triangulation, we refer the reader to the excellent surveys of Bern and Eppstein [3] and Bern and Plassmann [5]. Note that, because of the importance of mesh generation, there is a vast body of literature on mesh generation algorithms. We do not attempt to survey this literature here because the majority of these papers are not concerned with mathematical quality guarantees.

The remainder of this article is organized as follows. In section 2 we describe the class of allowable input domains for QMG. In sections 3 and 4 we present a high-level description of the QMG algorithm. In sections 5–7 we provide more details about the algorithm. In sections 8 and 9 we formally define aspect ratio and sharp angles and establish some results about them. In sections 10–17 we provide the analysis of QMG, including the proofs of the two main optimality properties mentioned above. In section 18, we consider the asymptotic running time of QMG, and in section 19 we briefly describe the implementation.

This article has a companion paper [12] that describes how to triangulate a grid of uniform boxes cut by a k-affine space. The method in that paper is used as a subroutine here, and we need some of the results of the analysis in that other paper for the analysis in section 10.

Besides the QMG algorithm and its analysis, the other main contribution of this paper is a series of new bounds that apply to any possible triangulation of a polyhedral domain (see section 9) and other results that apply to any possible bounded aspect ratio triangulation of a polyhedral domain (see section 16). The results in these sections act as lower bounds for proving QMG's optimality, but they would be useful for the analysis of other triangulation algorithms.

This article, along with the companion [12], supersedes our earlier work [13]. We briefly summarize the difference between this article and the earlier work. First, this work applies to d-dimensional regions for any d, whereas the earlier work was limited to three dimensions. A consequence of this generalization is that we have discarded the case-based proofs used in [13] in favor of more uniform treatment here. The notion of enforcing a "balance" condition in the quadtree has been dropped. Further, the idea of "warping" has been replaced by the approach in the companion paper, together with the "alignment" procedure described here.

2. Nonconvex polyhedra.

Recall that the input to our algorithm is a nonconvex polyhedron P in R^d. Mathematically, a nonconvex polyhedron is the set resulting from a finite number of union and intersection operations applied to halfspaces. We assume P is compact. We assume that P is presented via a *boundary representation*; in fact, from now on, we refer to polyhedra as "b-reps." The b-rep of P consists of a lattice of faces: zero-dimensional faces are called vertices, one-dimensional (1D) faces are called edges, and the d-dimensional face is P itself. Each face of dimension 1 or higher has boundaries that are faces of one lower dimension. Thus, a b-rep is stored as a layered directed acyclic graph with one node for each face, and with arcs to indicate the "is-a-boundary-of" relation. Nodes at level 0 (vertices) have coordinates stored with them.

Finally, to simplify our presentation, we assume that P is a d-manifold with boundary, although the implementation of QMG allows many nonmanifold features, such as internal boundaries.

3. Boxes.

The main data structure of QMG is a box. A *box* is a d-dimensional cube embedded in an axis-parallel manner in $\mathbf{R}^d$. Our algorithm is a *quadtree*-based algorithm, meaning that it starts with a single d-cube and then subdivides it into 2^d equal-size smaller cubes. The subdivision continues recursively.

Boxes of dimension less than d occur as separate data items. These lower-dimensional boxes are discussed in more detail in section 7. We ignore the existence of these lower-dimensional boxes until section 7 to allow a simplified presentation of QMG's quadtree generation in the next three sections.

Initially, there is one large d-dimensional box, called the *top box*, which contains all of P and also a neighborhood around P. This box is considered *active*. Other boxes are generated from the top box by applying one of three operations recursively. First, an active box may be *split*, meaning that it is replaced by 2^d smaller boxes each of equal size, as mentioned above. Second, a box may be *duplicated*, meaning that it is replaced by two or more boxes with the same size and position as the original box. Third, an active box may be *protected*, in which case it is no longer active and no longer available for splitting or duplicating. The collection of boxes is called a *quadtree*.

The data items stored with a box are as follows. QMG stores its position and size. Because of the dyadic nature of the quadtree, the position and size are both represented exactly (as integers). As mentioned in the last paragraph, boxes are either *active* or *protected*. An active box B has stored with it its *content*, which is denoted $\mathrm{co}(B)$. The definition of content is as follows. Let $\mathrm{ex}(B)$ denote a cube in $\mathbf{R}^d$ that is concentric with B but has a diameter larger by a constant factor $1+\gamma$, where γ is defined below. Note that $P \cap \mathrm{ex}(B)$ is a polyhedral region. If $P \cap \mathrm{ex}(B)$ is connected (in the topological sense), then we define $\mathrm{co}(B) = P \cap \mathrm{ex}(B)$. If $P \cap \mathrm{ex}(B)$ is not connected, then QMG makes duplicates of B, one for each component of $P \cap \mathrm{ex}(B)$, and assigns one component to each duplicate. Thus, $\mathrm{co}(B)$ is always a connected polyhedral region. More details are given in section 5.

A protected box is always associated with a particular face F of P, and F must meet $\mathrm{ex}(B)$. Thus, a protected box has stored with it a reference to F and also a close point. The *close point* is a point in $\mathbf{R}^d$ lying in $F \cap \mathrm{ex}(B)$. The coordinates of the close points are stored in an auxiliary table, and the protected box stores an index into this table. (This is because several protected boxes can share the same close point.) The collection of close points makes up the vertices of the final triangulation.

4. High-level description of the quadtree generation.

The mesh generation algorithm has two parts: quadtree generation and triangulation. See Figures 4.1–4.2 for the high-level outline of quadtree generation. Triangulation is described in section 7. Not all the terms in these figures have been defined yet.

Quadtree generation is divided into $d+1$ *phases* numbered $0, \ldots, d$. We use k throughout the article to denote the current phase. Phase k works primarily with the k-dimensional faces of P. (Thus, in phase d we look at P itself.) Each phase is subdivided into two *stages*, the separation stage and the alignment stage. During the separation stage, active boxes are split. There is also splitting of active boxes during the alignment stage. The alignment stage also turns some active boxes into protected boxes.

5. Separation stage.

In this section we describe the separation stage of phase k in more detail. At the start of the phase there is a list of active boxes I_k and the (initially empty) idle list I_{k+1}. During the phase we repeatedly remove one active box, say, B, from I_k and test it for crowdedness (defined below). If B is crowded, it is split. Let us use the term *children* to denote the boxes on the next deeper level resulting from the split. All the children with a nonempty content are inserted back

```
/* **Quadtree generation** */.
Initialize $I_0 := \{\text{top\_box}\}$.
Initialize $J := \{\}$.
for $k := 0, \ldots, d$ do
    Initialize $I_{k+1} := \{\}$.
    Initialize $O_F := \{\}$ for each $k$-dimensional $P$-face $F$.
    /* **Phase $k$ separation stage.** */
    while $I_k$ is nonempty do
        Remove an active box $B$ from $I_k$.
        if $B$ is crowded or too big for the size function then
            Split $B$ into $B_1, \ldots, B_{2^d}$; duplicate as necessary.
            Delete $B_i$'s with empty content.
            Put remaining $B_i$'s into $I_k$.
        elseif co($B$) contains a (necessarily unique) $k$-face $F$ of $P$ then
            $O_F := O_F \cup \{B\}$.
        else
            $I_{k+1} := I_{k+1} \cup \{B\}$.
        end if
    end while
```

FIG. 4.1. *High-level description of QMG's quadtree generation (continued in Figure* 4.2*).*

into I_k. Box B itself is deleted, and the children with empty content are deleted. On the other hand, if B is not crowded, we check whether it has a P-face of dimension k, say, F, in its content. If so, the box is transferred to orbit O_F. If not, the box is transferred to the idle list. In this manner I_k is eventually emptied.

We now explain the terms "content" and "crowdedness." First, we define ex(B) for an active box B to be a d-dimensional cube in $\mathbf{R}^d$ concentric with B but expanded in each dimension by a multiplicative factor $1+\gamma$. Parameter γ must satisfy $\gamma \geq \epsilon_{0,F}$ for each P-face F, where $\epsilon_{0,F}$ is the tolerance for alignment described in section 6. For instance, $\gamma = 0.5$ is acceptable.

The *content* of an active box B, denoted co(B), is a b-rep and is typically $P \cap$ ex(B). However, if $P \cap \text{ex}(B)$ has more than one connected component, we identify the components of $P \cap \text{ex}(B)$, say, $C_1, \ldots, C_p$, and we replace B with p copies of itself, say, $B_1, \ldots, B_p$. Then we define $\text{co}(B_i) = C_i$ for $i = 1, \ldots, p$.

Say B is split, and say B' is one of the child boxes. We compute co(B') by intersecting co(B) with ex(B'). (Notice that our definition of ex(B) guarantees that ex(B') is a proper subset of ex(B).) In particular, we do not compute co(B') by intersecting the original b-rep P with ex(B'). This is because this latter approach could reintroduce connected components that were duplicated into a box different from B at some earlier level of splitting.

We say that a box B is *crowded* if either

- co(B) meets any P-face of dimension $k-1$ or less, or
- co(B) meets a P-face F of dimension k, and co(B) meets another P-face G that is not a superface of F.

Thus, a box B is not crowded during phase k if either co(B) does not meet any P-faces of dimension k or lower, or co(B) meets exactly one P-face F of dimension k, no P-faces of dimension less than k, and every P-face of dimension higher than k

```
        /* **Phase $k$ alignment stage.** */
        for each $k$-dimensional $P$-face $F$ do
            while $O_F$ is nonempty do
                Remove the highest-precedence box $B$ from $O_F$.
                Find the highest-priority subface $B'$ of $B$ that is close to $F$.
                if $B$ has no such close subface then
                    $I_{k+1} := I_{k+1} \cup \{B\}$.
                elseif the alignment condition is satisfied for $B$ then
                    Protect $B$; its associated $P$-face is $F$.
                    Find the close point on $F$ for $B$ (near $B'$).
                    $J := J \cup \{B\}$.
                else
                    Split $B$ into $B_1, \ldots, B_{2^d}$; duplicate as necessary.
                    Delete the $B_i$'s with no content.
                    Put remaining $B_i$'s into $O_F$.
                end if
            end while
        end for
    end for /* **end of $k$ loop** */
```

FIG. 4.2. *High-level description of QMG's quadtree generation (continued from Figure* 4.1*).*

in co(B) is a superface of F. A box that is not crowded is transferred either to I_{k+1} in the first case (i.e., when co(B) does not meet any P-faces of dimension k or lower) or to O_F in the second case. Some examples of crowdedness are given in Figure 5.1.

Another rule used in the separation stage is that we split boxes if their side length is greater than the user-specified mesh refinement function that was mentioned in the introduction. We do not say any more about this here, since the analysis in subsequent sections does not involve a user-specified mesh refinement function.

The reader may notice that there appears to be a potential infinite loop: if an active box B in phase k has a P-face of dimension $k-1$ or less in its interior, this will cause an infinite recursion of splitting because there will always be a crowded subbox. Fortunately, this situation can never occur. The reason is that a box whose interior meets a P-face of dimension $k-1$ or less would have had a close subface identified in an earlier phase and would have become protected or would have been crowded in an earlier phase. (See the next section for a description of close subfaces.) Therefore, it could never end up in I_k. It is possible, however, for a box in I_k to have a face of dimension $k-1$ or less inside ex(B) but outside B. This can happen because, in the previous phase, a $(k-1)$-face F could lie in the content of B and yet not be close enough to come close to a subface of B. In this case the box will be split until the d-cubes ex(B) have shrunk enough that they do not meet the low-dimensional P-face. The number of times that a box can be split is analyzed in subsequent sections.

The computation of co(B) (that is, computing the geometric intersection co(B')$\cap$ ex(B), where B' is the parent of B, and then checking whether this intersection is connected) is among the most computationally intensive tasks of QMG. We carry out the search for connected components with a ray-shooting algorithm that we do not describe here. The worst-case running time of this ray-shooting algorithm is $O(n^2)$, where n is the total combinatorial complexity of co(B) (i.e., the total number of

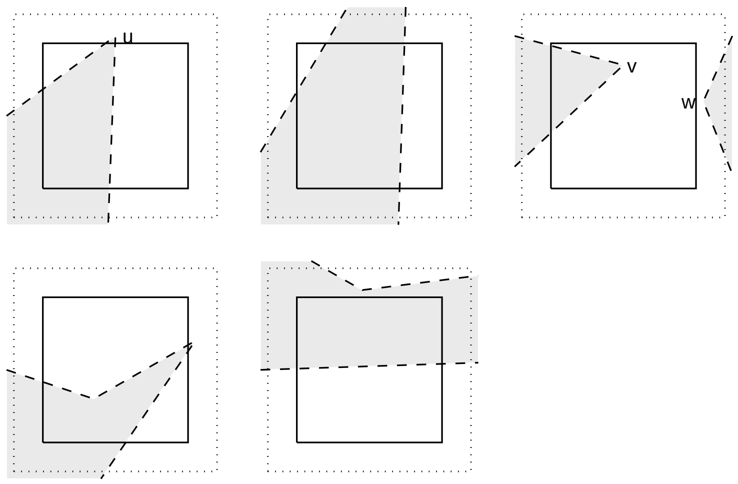


Fig. 5.1. *Examples of crowdedness: solid lines indicate boxes, dotted lines indicate* ex(B) *for these boxes, dashed lines indicate the boundary of P, and shading represents the interior of P. Suppose we are in the separation stage of the phase 0 in the case $d = 2$. All boxes in the top row are uncrowded. The first box would be placed into $O_{\boldsymbol{u}}$. The second box would be placed into I_1. The third box in the top row must be duplicated, and then one duplicate would go into $O_{\boldsymbol{v}}$ and the other into $O_{\boldsymbol{w}}$. Both boxes in the bottom row are crowded and must be split.*

boundary faces), but in practice the running time will usually be closer to $O(n)$.

In the case $d = 2$, it is possible to find connected components of co(B) via a plane sweep in $O(n \log n)$ operations. In the case $d = 3$, an $O(n \log n)$ plane sweep can also be used provided that P is preprocessed with $O(N^2)$ preprocessing steps, where N is the combinatorial complexity of the input polyhedron P. This efficient algorithm for $d = 3$ is described in our earlier paper [13]. We have not implemented a plane-sweep procedure for either $d = 2$ or $d = 3$.

6. Alignment.

In this section we describe the alignment stage. Recall that the alignment stage processes each orbit independently. For this section, assume we are in phase k and are processing orbit O_F of P-face F whose dimension is k.

First, a sequence of parameters

$$0 = \epsilon_{d,F} = \epsilon_{d-1,F} = \cdots = \epsilon_{d-k,F} < \epsilon_{d-k-1,F} < \epsilon_{d-k-2,F} < \cdots < \epsilon_{0,F} < 0.5$$

is chosen for F. The method for choosing positive scalars $\epsilon_{d-k-1,F}, \epsilon_{d-k-2,F}, \ldots, \epsilon_{0,F}$ is described in [12], which must be slightly modified to take into account the containment relationship between P-faces of different dimensions. These parameters have positive upper and lower bounds depending only on d and k.

We now process boxes in O_F in the order described below. Let B be the high-precedence box in the orbit. Let B' be any subface of B. We construct the ∞-norm neighborhood of radius $\epsilon_{r,F}$ around B', denoted $N(B')$, where r stands for the dimension of B'. Thus, this neighborhood is an axis-parallel parallelepiped (which could be degenerate if $\epsilon_{r,F} = 0$). If F meets $N(B')$, then F is said to be *close* to B'. The *close subface* of B is the box subface of lowest dimension that is close to F. If there is a tie (i.e., there are several faces of the same lowest dimension all close to F), then we break the tie with a priority rule, which is described below. A box with no close subface is transferred to I_{k+1}.

Because $\gamma \geq \epsilon_{0,F}$, if F is close to B (i.e., if B has a close subface), then F must meet $\mathrm{ex}(B)$. Thus, we can check whether B has a subface close to F by examining $\mathrm{co}(B)$. Indeed, it is important that we query $\mathrm{co}(B)$ rather than the original P, because it might be difficult to determine from queries on P whether the P-face in question is associated with B or with a duplicate of B.

Next we claim a partial converse: if F meets $\overline{\mathrm{ex}}(B)$, then B has a subface close to F. We define $\overline{\mathrm{ex}}(B)$ to be a cube concentric with B and expanded by $\epsilon_{d-k-1,F}$ in each dimension. Because $0 < \epsilon_{d-k-1,F} < \gamma$, it follows that $B \subset \overline{\mathrm{ex}}(B) \subset \mathrm{ex}(B)$. Furthermore, it follows from Lemmas 1 and 2 of [12] that if any P-face F meets $\overline{\mathrm{ex}}(B)$, then B has a subface close to F. The cube $\overline{\mathrm{ex}}(B)$ is not used in our algorithm, but it plays a role in the analysis below.

Once every box in the orbit has chosen its close subface, we now test the *alignment condition*. The alignment condition is as follows. Define the *extended orbit* of F to be O_F united with protected boxes from earlier phases that are associated with proper subfaces of F. For every active box B in the orbit, the close subface of B must be completely covered by boxes in the extended orbit (either active or protected) that are the same size or larger. For an example of the alignment condition, see Figure 6.1. We provide motivation for the alignment condition in section 7.

Let us now comment further on the alignment condition. First, we have to explain what is meant by "completely covered." We say that a box subface B' is *completely covered* by some collection of boxes $\{B_1, \ldots, B_n\}$ provided that for any point $\boldsymbol{p}$ in the relative interior of B', there exists an open neighborhood N of $\boldsymbol{p}$ such that $N \subset B_1 \cup \cdots \cup B_n$. (Note that if B' is a 0-dimensional box subface, i.e., a vertex, then its relative interior is B' itself.)

With these definitions of "extended orbit" and "completely covered," we can now state the priority rule for choosing a close subface. Recall that the close subface of B is the box subface of lowest dimension close to F. Let l be the dimension of this subface. If there is a tie (i.e., there is more than one face of B of the dimension l close to F), then we favor the subfaces that are completely covered by boxes the same size or larger in the extended orbit (i.e., those close subfaces of dimension l for which the alignment condition holds). If there is still a tie, we use a lexicographic tie-breaking rule.

Recall that boxes can get duplicated during the separation stage, and thus several active boxes can cover the same geometric region in $\mathbf{R}^d$. We claim that two boxes with overlapping geometric regions in $\mathbf{R}^d$ cannot end up in the same orbit. (This fact simplifies the sorting necessary to check the alignment condition.) The reason is as follows. Suppose B and B' are two boxes whose interiors have a common point in $\mathbf{R}^d$, and suppose both $\mathrm{co}(B)$ and $\mathrm{co}(B')$ contain a point of P-face F. By the tree-nature of the quadtree, two boxes that share a common interior point must have the property that one is contained in the other.

We claim that $\mathrm{co}(B)$ and $\mathrm{co}(B')$ both must meet a proper subface of F. If not, then $\mathrm{co}(B)$ would have to contain the intersection of $\mathrm{ex}(B)$ with the entire affine hull of F (because no boundaries of F are in $\mathrm{co}(B)$). Similarly, $\mathrm{co}(B')$ also would contain the intersection $\mathrm{ex}(B')$ with the hull of F. (See section 8 for definition of "affine hull" and other mathematical terminology.) Since one box contains the other, this means that one box contains points from F that the other box also contains. But then there could not be two distinct connected components of P in $\mathrm{co}(B)$ and $\mathrm{co}(B')$, so duplication would not have taken place.

Thus, $\mathrm{co}(B), \mathrm{co}(B')$ each contain proper subfaces of F. But in this case, the boxes

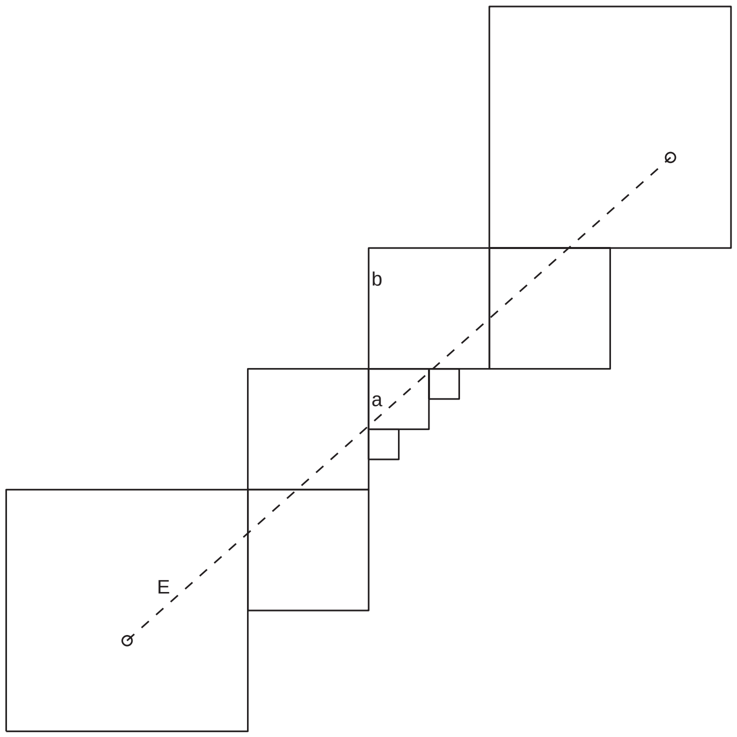


Fig. 6.1. *The alignment condition in the case $d = 2, k = 1$: the boxes in this figure are the extended orbit of a P-edge E, which is the dashed line. The two large boxes at the ends are protected boxes for the endpoints of E, protected from phase 0. In this figure, box a must be split because the alignment condition does not hold for this box. Its close subface, which could be either its lower left-hand corner or upper right-hand corner, is contained by another box smaller than a. All other boxes satisfy the alignment condition. For example, box b does not have to be split; its close subface could be either its bottom edge or right edge. The right edge will have higher priority, since the alignment condition holds for that edge.*

could not end up in O_F (i.e., if they were still active in phase $\dim(F)$, they would be crowded).

As mentioned earlier, a box is protected if the alignment condition holds for its close subface. We make the following claim: if the alignment condition holds for B at the time it is protected, then the condition continues to hold for the remainder of the algorithm. In other words, the following situation cannot occur: a box B with close subface B' is deemed to satisfy the alignment condition and becomes protected. Later a neighboring box $\bar{B}$ also containing B' as a subface gets split because the alignment condition does not hold for $\bar{B}$, thus causing the alignment condition to be violated for B.

To prove the claim in the last paragraph, we must describe the order in which QMG processes the boxes in an orbit O_F. "Process" means that QMG determines whether the box satisfies the alignment condition; if so, then protect it, and if not, then split it. The correct order is to start with the largest boxes in the orbit, working down to the smallest. Within the set of boxes of the same size, we process those with the lowest-dimensional close subfaces first, working toward highest-dimensional close subfaces.

We claim that this order assures that if the alignment condition holds for a box B at the time it is processed, then the alignment condition holds for B for the remainder of the algorithm. Suppose we are at the step when B is processed and the alignment condition is satisfied. Let B' be the close subface of B, and let l be the dimension of B'. Let $B_1, \ldots, B_n$ be the boxes in the extended orbit that cover B'. Some of $B_1, \ldots, B_n$ will be larger than B' and hence already protected. Protected boxes are not split again, so they will continue to cover B' for the rest of the algorithm. Consider a box B_i that is the same size as B. If B_i has a close face of dimension less than l, then B_i is already protected (because we process boxes with lower-dimensional close faces first). The dimension of the close face of B_i cannot be greater than l, because B' is a subface of B_i and has higher priority than any subface of B_i of dimension $l+1$ or more. Therefore, the only remaining possibility is that B_i is the same size as B and that the close subface of B_i has dimension exactly l. But then this subface, if it is not B', must also be completely covered by boxes in the orbit because otherwise B' would have higher priority. (Recall that faces that are completely covered have higher priority.) So we see that B_i will become protected as well and cannot be split.

When a box B is protected, as mentioned above, we have identified a close subface B' of B. This subface has the property that F meets an ∞-norm neighborhood of B'. We now select a point lying on F in this neighborhood (see [12] for more details on selecting the close point). The rule used for choosing the close point has the property that any other box B'' that is the same size as B and also has B' as its close subface will choose the same close point. Thus, several adjacent boxes that are in the same orbit and are the same size might share a close point.

The alignment stage continues until there are no boxes left to process; every box is either protected or has been moved to the idle list. When O_F is empty, the alignment moves onto a different orbit. Once the orbits of all dimension-k faces of P are empty, the phase is over.

7. Triangulation.

After phase d of quadtree generation, QMG triangulates the quadtree. In the triangulation procedure, the collection of protected boxes is triangulated into a simplicial complex.

To describe the triangulation procedure, we must first bring lower-dimensional boxes into the picture. In this section, we revisit some of the concepts from earlier sections and revise some of the algorithm steps to take into account lower-dimensional boxes. The lower-dimensional boxes serve two purposes: first, they simplify the data structures needed for checking the alignment condition, and second, they serve as the basis for generating the final triangulation.

In QMG, boxes can have dimension 0 up to d. Initially, there is only one active box of dimension d, namely, the top box. Lower-dimensional boxes get created each time a box is protected by QMG. At the moment an i-dimensional active box B is changed from active to protected during the alignment stage for orbit O_F, all its faces of dimension $i-1$ are launched as new active boxes (there are $2i$ such faces). Let B' be one of these new active boxes. It is dealt with in a manner analogous to the way QMG handles subboxes after a split. We compute $\mathrm{co}(B')$ as the intersection $\mathrm{co}(B) \cap \mathrm{ex}(B')$. We determine the disposition of B' using the same rules as before: if $\mathrm{co}(B')$ is empty, then we delete B'. If $\mathrm{co}(B) \cap \mathrm{ex}(B')$ has more than one connected component, then we duplicate B'. If $\mathrm{co}(B')$ does not meet F (and hence meets only P-faces of dimension $k+1$ and higher), then we place B' in I_{k+1}. If $\mathrm{co}(B')$ meets F, then we place B' in O_F.

It is possible that B' will also become immediately protected; this happens, for instance, when the close subface of B is also a subface of B'.

The same operations are performed on an i-dimensional box as are performed on a full-dimensional box: such a box can be tested for crowdedness, split for separation, protected, and so on. When an i-dimensional box is split, 2^i new subboxes are created. If B is i-dimensional, it is said to *extend over* i of the possible d coordinate axes, and it is *flat* over the remaining $d - i$ coordinate axes.

The definition of $\mathrm{ex}(B)$ for a box of dimension less than d is as follows. Every box B has associated with it a number called its *size*, which we denote $\mathrm{size}(B)$ and which is the side length of B in a dimension over which it extends. The size of every box is equal to the size of the top box multiplied by a factor 2^{-p}, where p is the number of times the top box was split to reach this box. If B is a box, then $\mathrm{ex}(B)$ is an axis-parallel full-dimensional parallelepiped in $\mathbf{R}^d$ concentric with B, with side lengths $(1 + \gamma)\,\mathrm{size}(B)$ for axes over which B extends, and side length $\gamma\,\mathrm{size}(B)$ for the axes in which B is flat. This choice ensures that all properties of $\mathrm{ex}(B)$ asserted earlier are still valid, namely, if B has a subface close to F, then the subface meets $\mathrm{ex}(B)$. Also, if B is split, then $\mathrm{ex}(B')$ for each subbox B' is contained in $\mathrm{ex}(B)$. Finally, if subfaces of B are launched as new active boxes when B is protected, then each new subface B' also satisfies $\mathrm{ex}(B') \subset \mathrm{ex}(B)$. Note that for consistency, even zero-dimensional boxes must have a size. When a zero-dimensional box B is split, there is only one child, but splitting still has significance because the size is halved, which diminishes $\mathrm{ex}(B)$ and therefore $\mathrm{co}(B)$.

Earlier, when describing the alignment condition, we introduced the terms "extended orbit" and "completely cover." Recall that we defined extended orbit to be the union of the orbit O_F of a face F united with the protected boxes for all proper subfaces of F from previous phases. In fact, QMG never forms extended orbits; instead, the lower-dimensional active box faces of protected boxes act as proxies for the protected boxes. A system of weights is used to determine the complete coverage condition. In particular, every active box in QMG stores a weight associated with each of its subfaces. Thus, an i-dimensional active box has 3^i weights stored with it. Each weight is a number between 0 and 1 that indicates what fraction of the subface is "owned" by that active box. Initially, the top box owns all of its subfaces. When a box is split, the weights are divided up among children. We omit the details of how the weights get split up, but the upshot is that QMG can test whether a box subface is completely covered by boxes in its orbit by adding up the weights associated with that subface contributed by all the boxes containing it; complete coverage is indicated by a weight sum of 1.0.

Although many details are omitted, we do mention one key point that reduces the amount of searching and sorting in QMG. When testing the complete coverage rule, it is necessary to look only at boxes of a single size. This means that the complete coverage condition can be tested with a simple hash table. Consider the example in Figure 7.1.

It can be shown that these more complicated rules introduced in this section are equivalent to the definitions in the preceding sections in the following sense. For a given input P, the quadtree generation procedure produces the same sequence of full-dimensional boxes whether we follow the rules of this section or preceding sections.

The protected boxes are linked together by pointers; in particular, a protected box of dimension i has pointers to all the protected boxes of dimension $i + 1$ of which it is a subface. This data structure serves as the basis for triangulation.

The triangulation algorithm is based on our other paper [12] and is as follows.

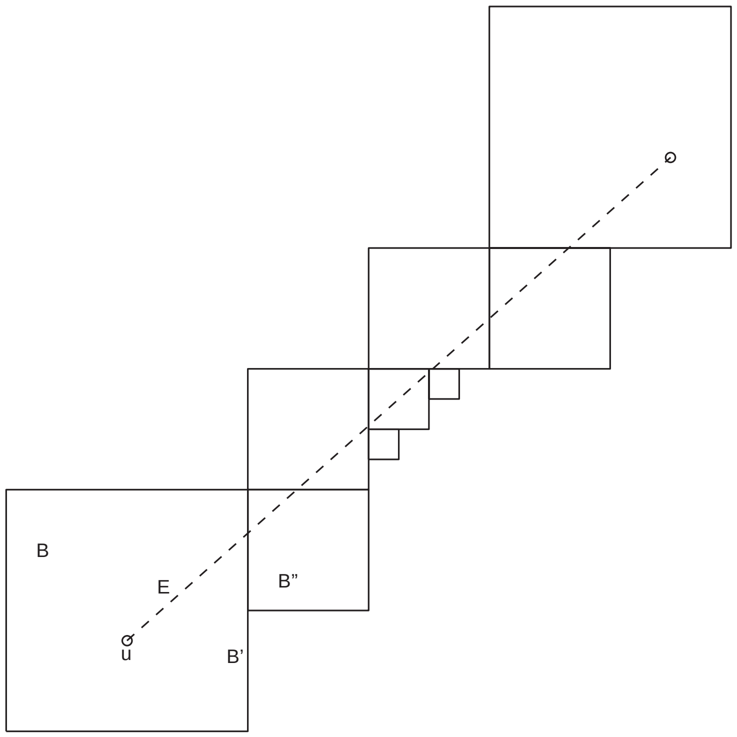


FIG. 7.1. *The alignment condition taking into account lower-dimensional boxes. The lower left-hand box B is protected in phase 0 for the vertex u of P in its interior. The right edge of this box—call it B'—becomes a new active box. It is uncrowded; its associated P-face is E (E is the dashed segment); and its close subface is the whole box B'. During the alignment phase 1, this 1D box is split because the weight of B' associated with B' itself is only 0.5. Once B' is split in half, its lower half no longer meets E and hence is placed into I_2. Its upper half, together with the full-dimensional active box labeled B'', completely covers the upper half of B' so the alignment condition is satisfied.*

Let a *chain* be a sequence of nested boxes $B_0, \ldots, B_d$ such that the dimension of B_i is i. “Nested” means that, for each i, B_i is a face or subset of a face of B_{i+1}. Let $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$ be the close points of $B_0, \ldots, B_d$. Then the simplex whose vertices are $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$ is put in the triangulation. Thus, the triangulation has one simplex for each chain. The only exception is when a close point is repeated in this chain; in this case, the simplex is said to be *null* and is not included in the triangulation. QMG enumerates all possible chains with a stack-based search algorithm. An example of the triangulation algorithm is presented in Figure 7.2.

We can now explain the importance of the alignment condition in Figure 7.3. As is seen from the figure, if the alignment condition were not enforced, then the triangulation algorithm described in the preceding paragraph would be invalid.

8. Aspect ratio.

In our analysis of QMG, which begins in section 10, we demonstrate two optimality properties: the triangulation generated by QMG has optimal aspect ratio, up to a certain factor, and also optimal cardinality (compared with all other bounded aspect ratio triangulations), up to a certain factor. Before demonstrating these properties, we must provide definitions for aspect ratio, sharp angle, and so on. This mathematical background is the topic of this section and the next

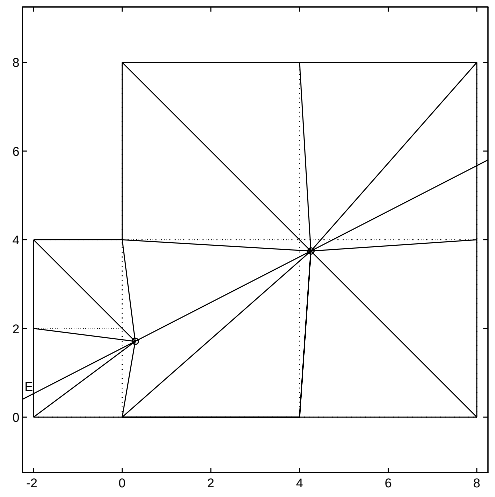


FIG. 7.2. *An example of the triangulation procedure in the case* $d = 2$. *All six full-dimensional protected boxes in this figure are associated with* P*-edge* E. *The dotted lines are boundaries of boxes that are not present in the final triangulation. The solid segments are all part of the triangulation. The close point for the two small boxes on the left is the point near* $(0, 2)$ *marked in the figure. An example of a nonnull chain in this figure would be starting from the vertex* $(0, 0)$ *(whose close point is at* $(0, 0)$ *and is associated with the* $2D$ *face* P *itself), then the edge* $\{0\} \times [0, 2]$ *containing it, whose close point is the marked point on* E *near* $(0, 2)$*, and finally the box* $[0, 4] \times [0, 4]$*, whose close point is the marked point near* $(4, 4)$ *on* E. *An example of a null chain would be the vertex at* $(4, 0)$ *(whose close point is at* $(4, 0)$*), the edge* $\{4\} \times [0, 4]$ *(whose close point is on* E *near* $(4, 4)$*), and finally the box* $[4, 8] \times [0, 4]$*, which has the same close point near* $(4, 4)$. *This figure shows a triangulation on both sides of* E *for illustrative purposes, although if* E *were a boundary edge, in fact only one side of* E *would be triangulated. As mentioned in the introduction, however, the actual implementation of QMG allows internal boundaries, so the above situation of triangulating both sides of an edge does occur with QMG.*

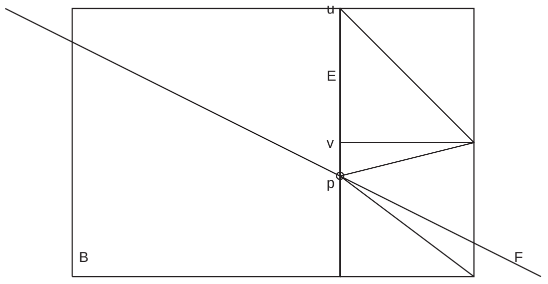


FIG. 7.3. *An inconsistency that would arise during triangulation if the alignment condition were not enforced. In particular, the close point of box* B *is point* $\boldsymbol{p}$*, which lies on* P*-face* F *and also on the right edge of box* B. *Thus, the close face for* B *is this edge. The alignment condition does not hold; that is, the close face of* B *is covered by smaller boxes on the right. If the triangulation algorithm were applied despite the violation of the alignment condition, then an illegal triangulation would result. For instance, the close point of edge* E *is one of its endpoints, say, endpoint* $\boldsymbol{v}$. *Then the chain consisting of* $\boldsymbol{u}$*, then* E *(whose close point is* $\boldsymbol{v}$*), and then* B *(whose close point is* $\boldsymbol{p}$*) is a flat simplex, that is, a triangle with collinear vertices. Such a simplex has infinite aspect ratio and must not be allowed in a triangulation. Note that a degenerate simplex like this is not a null simplex as defined in section 7: this degenerate simplex does not have any repeated vertex, and hence it cannot be legally dropped from the triangulation.*

section. In this section, all norms are assumed to be Euclidean norms (i.e., 2-norms), although most of the bounds are valid for other standard norms as well (since the bounds include a multiplicative factor depending on d).

First, we provide some standard definitions from linear algebra. An *affine set* X is the solution to a system of linear equations, that is, $X = \{\boldsymbol{x} \in \mathbf{R}^d : A\boldsymbol{x} = \boldsymbol{b}\}$ for some $m \times d$ matrix A with linearly independent rows and some m-vector $\boldsymbol{b}$. The *dimension* of this affine set is $d - m$. Let Y be any subset of $\mathbf{R}^d$. The *affine hull* of Y is defined to be the lowest-dimensional affine set that contains Y and is denoted $\operatorname{aff}(Y)$. It can be shown that $\operatorname{aff}(Y)$ is uniquely determined by this definition. In particular, it can be shown that $\operatorname{aff}(Y)$ is the set of all points that can be written in the form $\alpha_1\boldsymbol{y}_1 + \cdots + \alpha_s\boldsymbol{y}_s$, where s is an arbitrary positive integer, $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_s \in Y$, and $\alpha_1, \ldots, \alpha_s$ is an arbitrary sequence of real numbers that add up to 1. Let F be a face of P. Since P is polyhedral, $\operatorname{aff}(F)$ and F have the same dimension.

We now define aspect ratio.

DEFINITION 8.1. *Let T be a d-simplex in $\mathbf{R}^d$ with vertices $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$. Then the* altitude *of T at $\boldsymbol{v}_i$ is defined to be* $\operatorname{dist}(\boldsymbol{v}_i, \operatorname{aff}(\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{i-1}, \boldsymbol{v}_{i+1}, \ldots, \boldsymbol{v}_d))$. *The* minimum altitude *of T, denoted* $\operatorname{minalt}(T)$, *is the minimum altitude over all choices of $\boldsymbol{v}_i$ for $i = 0, \ldots, d$.*

DEFINITION 8.2. *For a d-simplex T in $\mathbf{R}^d$, the* maximum side length *of T, denoted* $\operatorname{maxside}(T)$, *is defined to be*

$$\max\{\|\boldsymbol{v}_i - \boldsymbol{v}_j\| : i, j = 0, \ldots, d\},$$

where $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$ are the vertices of T.

DEFINITION 8.3. *The* aspect ratio *of a simplex T is defined to be*

$$\operatorname{asp}(T) = \operatorname{maxside}(T)/\operatorname{minalt}(T).$$

Thus, the aspect ratio is always at least 1, and large aspect ratios indicate poor-quality elements.

In the remainder of this section, we characterize aspect ratio in terms of matrix norms. Given a d-simplex T, we define its associated matrix M_T to be the $d \times d$ matrix whose ith column, $i = 1, \ldots, d$, is $\boldsymbol{v}_i - \boldsymbol{v}_0$. Thus, M_T depends on the numbering of the vertices, and in particular, $\boldsymbol{v}_0$ plays a distinguished role. However, we note the following: if we define M_T' according to a different numbering of the vertices, the columns of M_T' can be obtained from the columns of M_T by subtracting pairs of columns in M_T and then permuting. In linear algebra terms, there exists a $d \times d$ matrix L all of whose entries are 0's except for possibly one 1 and one -1 in each column such that $M_T' = M_T L$, and such that L^{-1} has the same properties (all 0's except for possibly one 1 and one -1 per column).

The following two results hold for any numbering. These lemmas use the following well-known linear algebra fact. The norm of a $d \times d$ matrix is bounded above and below by constant multiples (where the constant depends on d) of the maximum norm among its columns, and also above and below by constant multiples of the maximum norm among its rows. In the remainder of this article, c_d denotes a constant depending only on d, which may change from formula to formula.

LEMMA 8.4. *Let σ denote* $\operatorname{maxside}(T)$. *Then*

$$c_d\sigma \le \|M_T\| \le C_d\sigma,$$

where c_d, C_d are two constants depending only on d.

Proof. There are two cases, depending on whether σ is the length of a side adjacent to $\boldsymbol{v}_0$. In the first case, say that $\sigma = \|\boldsymbol{v}_1 - \boldsymbol{v}_0\|$. Then both inequalities are easy because σ is the norm of the first column of M_T, and all the other columns of M_T have norms bounded above by σ.

The other case is that σ is the length of a side not adjacent to $\boldsymbol{v}_0$. We can reduce to the first case by renumbering the vertices and noting that the norms of the transformation matrices mentioned above, $\|L\|$ and $\|L^{-1}\|$, are bounded above by constants depending only on d. □

LEMMA 8.5. *Let* $\mu = \operatorname{minalt}(T)$. *Then*

$$c_d/\mu \leq \|M_T^{-1}\| \leq C_d/\mu,$$

where c_d, C_d *are two constants depending only on* d *(not necessarily the same constants as in the previous lemma).*

Proof. Let $\boldsymbol{u}^{\mathrm{T}}$ be the ith row of M_T^{-1}. Then $M_T^{\mathrm{T}}\boldsymbol{u}$ is a column of the identity matrix. (The superscript T denotes transpose. The subscript T indicates the association of M with simplex T.) Geometrically, this means that $\boldsymbol{u}$ is orthogonal to $d-1$ columns of M_T; in particular, $\boldsymbol{u}$ is orthogonal to the plane $\operatorname{aff}(\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{i-1}, \boldsymbol{v}_{i+1}, \ldots, \boldsymbol{v}_d)$. Thus, $\boldsymbol{u}$ is parallel to the altitude from vertex i. Its length is chosen so that its inner product with $\boldsymbol{v}_i - \boldsymbol{v}_0$ is 1, which implies that its inner product with the true altitude vector is 1. Thus, the ith row of M_T^{-1} is parallel to the altitude vector from $\boldsymbol{v}_i$ but is scaled so that its length is the reciprocal of the altitude.

Then we see that the rows of M_T^{-1} have lengths equal to reciprocals of altitudes from $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$, with the shortest altitude being the reciprocal of the norm of the largest row. This proves the lemma, provided that the minimum altitude is not adjacent to $\boldsymbol{v}_0$. The case when the minimum altitude is adjacent to $\boldsymbol{v}_0$ is handled by renumbering as in the previous proof. □

We conclude from these two lemmas that $\operatorname{asp}(T)$ is within a constant factor of $\|M_T\| \cdot \|M_T^{-1}\|$, that is, the condition number $\kappa(M_T)$. Combining these lemmas with the Hadamard inequality yields the following well-known result:

$$c_d \operatorname{minalt}(T)^d \leq \operatorname{vol}(T) \leq C_d \operatorname{maxside}(T)^d. \tag{8.1}$$

9. Angles and PL paths. In the preceding section, we defined "aspect ratio." It turns out that we can show that QMG produces triangulations whose aspect ratio is bounded above in terms of the sharpest angle of the input domain P. In this section, we provide the definition of "sharpest angle" and a theorem stating that *any possible* triangulation of P has aspect ratio bounded below in terms of the sharpest angle. Thus, the theorem in this section is the lower bound necessary to prove that the QMG triangulation is optimal.

Let $\boldsymbol{x}, \boldsymbol{y}$ be two points in P. A *piecewise linear* (PL) path Π from $\boldsymbol{x}$ to $\boldsymbol{y}$ is a path composed of a finite number of line segments. The endpoints of the segments are the *breakpoints* of Π. The *length* of this path, denoted $\operatorname{lth}(\Pi)$, is the sum of the lengths of the individual segments. Suppose that $\boldsymbol{x} \in F$ and $\boldsymbol{y} \in G$, where F and G are two faces of P. We will say that Π is *contractible* if there exists a point $\boldsymbol{z}$ such that the segment $\boldsymbol{xz}$ lies in F, the segment $\boldsymbol{yz}$ lies in G, and for all $\boldsymbol{v} \in \Pi$, the segment $\boldsymbol{vz}$ lies in P. Note that this definition forces $\boldsymbol{z}$ to lie in both F and G. Thus, a necessary condition for contractibility is that F and G have a nonempty common subface.

Note that we should really apply this term "contractible" to a triplet (Π, F, G), since the definition depends on the specification of F and G as well as on the path Π.

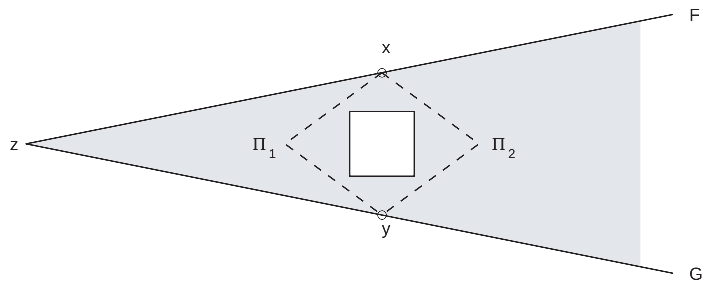


FIG. 9.1. *In the $d = 2$ case, F and G are two boundary segments meeting at boundary vertex $\boldsymbol{z}$. The polygon has a square hole in it: the interior of P is shaded. The PL path Π_1 connects a point $\boldsymbol{x} \in F$ to $\boldsymbol{y} \in G$. This path is contractible to $\boldsymbol{z}$. On the other hand, path Π_2 is incontractible.*

When we use the term, the choice of F and G will be understood from context. The opposite of contractible is *incontractible.*

Let $\mathcal{T}$ be an arbitrary triangulation of P (not necessarily the triangulation produced by QMG). If a path Π is contractible, we can obtain a lower bound on the aspect ratios of simplices of $\mathcal{T}$ that meet Π. On the other hand, if Π is incontractible, we can obtain an upper bound on the minimum altitude of simplices that meet Π. The remainder of this section is devoted to stating and proving these two results.

We start with the definition of the "angle" between two P-faces F and G, which is defined by contractible paths.

DEFINITION 9.1. *Let F, G be two faces of P, and suppose $\boldsymbol{x} \in F$ and $\boldsymbol{y} \in G$. Let Π be a contractible PL path from $\boldsymbol{x}$ to $\boldsymbol{y}$. Let A be the affine set* $\mathrm{aff}(F) \cap \mathrm{aff}(G)$. *The* angle *determined by* $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ *is*

$$\theta = \frac{\mathrm{lth}(\Pi)}{\min(\mathrm{dist}(\boldsymbol{x}, A), \mathrm{dist}(\boldsymbol{y}, A))}. \tag{9.1}$$

We say that the sharpest angle *formed by F and G is the infimum of* (9.1) *over all contractible paths from F to G (assuming that at least one contractible path from F to G exists). Finally, we say that the sharpest angle in P is the infimum over all angles.*

In the denominator of (9.1), "dist" denotes ordinary Euclidean distance. Note that $\boldsymbol{z}$, the base of the contraction, does not appear in (9.1). An example of a contractible path is given in Figure 9.1. We are concerned with the case when this angle is small, so large angles have no significance. Thus, the degenerate case when the denominator of (9.1) is 0 (which happens, for instance, if F is a superface of G or vice versa) is not relevant for our analysis.

In the case $d = 2$, the previous definition is within a constant factor of the ordinary notion of an angle between two edges of a polygon, since only the case $\dim(F) = \dim(G) = 1$ matters when $d = 2$.

Now for the first main result of this section: we show that if there is a contractible path Π forming an angle θ, then θ^{-1} is a lower bound on the aspect ratio of at least one simplex in every possible triangulation of P.

THEOREM 9.2. *Let F, G be two P-faces, and assume there is a contractible path from F to G. Let θ be the value of the sharpest angle between F and G. Let $\mathcal{T}$ be an arbitrary triangulation of P. Then there is a simplex T in the triangulation with a vertex lying on $F \cap G$ such that* $\mathrm{asp}(T) \geq c_d/\theta$.

Proof. Let $(\boldsymbol{x}, \boldsymbol{y}, \Pi)$ be the triple defining the sharpest angle θ between F and G, and let $\boldsymbol{z}$ be the point in $F \cap G$ to which we can contract Π. (Stating this more carefully, since sharpest angle is defined as an infimum, we should say that $(\boldsymbol{x}, \boldsymbol{y}, \Pi)$ defines an angle of size $(1+\epsilon)\theta$, where $\epsilon > 0$ is arbitrarily small. But the $1+\epsilon$ factor can be absorbed by the c_d factor.) Let H be the P-face contained in $F \cap G$ that contains $\boldsymbol{z}$. (If there is more than one P-face H satisfying $\boldsymbol{z} \in H \subset F \cap G$, choose any such H.) Let $A = \mathrm{aff}(F) \cap \mathrm{aff}(G)$. Note that $H \subset A$, since $H \subset F \cap G$.

In the triangulation of P, restrict attention to simplices of $\mathcal{T}$ that have at least one vertex on H. Call this collection of simplices $\mathcal{T}'$. Since $\mathcal{T}'$ is a finite set, there is an $\epsilon > 0$ such that every point in $\boldsymbol{v} \in P$ satisfying $\mathrm{dist}(\boldsymbol{v}, \boldsymbol{z}) \le \epsilon$ and $\boldsymbol{v}\boldsymbol{z} \subset P$ is contained in a simplex from $\mathcal{T}'$. Then we can contract $(\boldsymbol{x}, \boldsymbol{y}, \Pi)$ toward $\boldsymbol{z}$ (i.e., replace $\boldsymbol{x}$ by $(1-\lambda)\boldsymbol{z} + \lambda\boldsymbol{x}$, $\boldsymbol{y}$ by $(1-\lambda)\boldsymbol{z} + \lambda\boldsymbol{y}$, and each point $\boldsymbol{v} \in \Pi$ by $(1-\lambda)\boldsymbol{z} + \lambda\boldsymbol{v}$ for some fixed $\lambda \in (0, 1]$) so that, without loss of generality, all of Π is covered by simplices in $\mathcal{T}'$. Note that the contraction operation does not affect the value of θ because the numerator and denominator of (9.1) scale by the same amount when we contract toward $\boldsymbol{z}$.

Without loss of generality, $\mathrm{dist}(\boldsymbol{x}, A) \ge \mathrm{dist}(\boldsymbol{y}, A)$; define $\alpha = \mathrm{dist}(\boldsymbol{y}, A)$. Define $\beta = \mathrm{lth}(\Pi)$. Thus, $\theta = \beta/\alpha$ is the sharpest angle. Now define a continuous piecewise linear function $f : P \to \mathbf{R}$ as follows. We first define f on the vertices of $\mathcal{T}$ as follows. For each $\mathcal{T}$-vertex $\boldsymbol{v} \in G$ we define $f(\boldsymbol{v}) = \mathrm{dist}(\boldsymbol{v}, A)$ where distance is measured in the ordinary Euclidean sense. Since $H \subset A$, this fixes $f(\boldsymbol{v}) = 0$ for vertices $\boldsymbol{v} \in H$. For all other vertices $\boldsymbol{v}$ of $\mathcal{T}$ we define $f(\boldsymbol{v}) = 0$. Notice that all vertices $\boldsymbol{v}$ of F have $f(\boldsymbol{v}) = 0$ because the intersection of F and G is contained in A. Now extend f to all of P by linearly interpolating over each simplex. This yields a uniquely determined piecewise linear function $f : P \to \mathbf{R}$. Notice that f is identically 0 on F.

Next, we claim that $f(\boldsymbol{y}) \ge \alpha$. Notice that for points on G, f is a linear interpolation of the function $\boldsymbol{u} \mapsto \mathrm{dist}(\boldsymbol{u}, A)$. This latter function is a convex function because A is convex. A linear interpolant of a convex function is always greater than or equal to the function value itself; thus $f(\boldsymbol{y}) \ge \mathrm{dist}(\boldsymbol{y}, A) = \alpha$.

On the other hand, $f(\boldsymbol{x}) = 0$ because $\boldsymbol{x} \in F$. Let $f|_\Pi$ be the restriction of f to Π; then $f|_\Pi$ is PL and continuous and increases by α. The length of Π is β. Therefore, there is a point $\boldsymbol{u}$ where the directional derivative of f at $\boldsymbol{u}$ parallel to Π is at least α/β in magnitude. Let T be the simplex of $\mathcal{T}$ containing $\boldsymbol{u}$ (if there is more than one such T, choose arbitrarily). Note that $T \in \mathcal{T}'$ because we are assuming Π is covered by $\mathcal{T}'$. On this simplex T, since the gradient is constant, $\|\nabla f\| \ge \alpha/\beta$.

There is an analytic expression for ∇f on T as follows. Let us number the vertices of T with $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$ so that $\boldsymbol{v}_0$ is a vertex on H. (Recall that every simplex in $\mathcal{T}'$ has at least one vertex on H.) Let $r_i = f(\boldsymbol{v}_i)$ for $i = 0, \ldots, d$. Thus, $r_0 = 0$ because f is zero on H, as noted above. Then one checks that $f(\boldsymbol{u})$ on T is given by the linear mapping $f(\boldsymbol{u}) = \boldsymbol{r}^{\mathrm{T}} M_T^{-1}(\boldsymbol{u} - \boldsymbol{v}_0)$, where M_T was defined earlier, and $\boldsymbol{r}^{\mathrm{T}}$ denotes $(r_1, \ldots, r_d)$. Then we see that ∇f on T is given by $(M_T^{\mathrm{T}})^{-1}\boldsymbol{r}$, so

$$\|\nabla f\| \le \|M_T^{-1}\| \cdot \|\boldsymbol{r}\| \le c_d \cdot (\max |r_i|)/\operatorname{minalt}(T).$$

Notice that $\max |r_i|$ is the maximum distance of a vertex of T from A, but this is at most $\operatorname{maxside}(T)$, since T has an edge from each of its vertices to $\boldsymbol{v}_0$, which lies on A. Thus,

$$\begin{aligned} \|\nabla f\| &\le c_d \operatorname{maxside}(T)/\operatorname{minalt}(T) \\ &= c_d \operatorname{asp}(T). \end{aligned}$$

On the other hand, we showed in the previous paragraph that $\|\nabla f\| \geq \alpha/\beta$, which is the reciprocal of θ. Thus, we have proved that the aspect ratio of T is bounded below by the reciprocal of the sharpest angle between F and G. ∎

The previous result shows that the presence of a contractible path gives a useful bound that is applicable to any triangulation $\mathcal{T}$. On the other hand, the presence of an incontractible path also gives a useful bound. We start with a lemma and then prove the main result.

LEMMA 9.3. *Let Π be an incontractible path from $\boldsymbol{x} \in F$ to $\boldsymbol{y} \in G$. Let $\mathcal{T}$ be an arbitrary triangulation of P. Let $F_1, \ldots, F_m$ be an enumeration of all the faces (of all dimensions including d) of $\mathcal{T}$ that meet Π. Then $F_1 \cap \cdots \cap F_m = \emptyset$.*

Proof. Suppose that $F_1, \ldots, F_m$ have a common point $\boldsymbol{z}$. Since the triangulation is boundary conforming, $\boldsymbol{z}$ lies on a common subface of F and G (because the lowest-dimensional triangulation face meeting $\boldsymbol{x}$ must be a subface of F, and similarly for the lowest-dimensional face meeting $\boldsymbol{y}$). Furthermore, every point $\boldsymbol{v}$ on Π is covered by a simplex that also covers $\boldsymbol{z}$. Since simplices are convex, this simplex also covers the segment $\boldsymbol{vz}$. Thus, Π is contractible to $\boldsymbol{z}$, contradicting the assumption. ∎

THEOREM 9.4. *Let F, G be two P-faces, and let Π be an incontractible path from $\boldsymbol{x} \in F$ to $\boldsymbol{y} \in G$. Let $\mathcal{T}$ be an arbitrary triangulation of P. Then $\mathcal{T}$ contains a simplex T meeting Π such that*

$$\operatorname{minalt}(T) \leq c_d \operatorname{lth}(\Pi).$$

Proof. Let $F_1, \ldots, F_m$ be an enumeration of faces of $\mathcal{T}$ meeting Π. By the preceding lemma, $F_1 \cap \cdots \cap F_m = \emptyset$. Let the vertices of F_1 be denoted $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_s$, where $s \leq d$. Since Π meets F_1, there is a point, say, $\boldsymbol{z}$, on Π that can be written as a convex combination of the vertices of F_1:

$$\boldsymbol{z} = \lambda_0 \boldsymbol{v}_0 + \cdots + \lambda_s \boldsymbol{v}_s,$$

where each λ_i is nonnegative, and $\lambda_0 + \cdots + \lambda_s = 1$. Therefore, for some i, $\lambda_i \geq 1/(s+1) \geq 1/(d+1)$. Without loss of generality, say that $\lambda_0 \geq 1/(d+1)$. Note that since the F_i's are disjoint, there is some F_i that does not contain $\boldsymbol{v}_0$. Let this other face be denoted F_2.

Let $f : P \to \mathbf{R}$ be a PL continuous function defined as follows. We set $f(\boldsymbol{v}_0) = 1$. For all other vertices $\boldsymbol{v}$ of $\mathcal{T}$, set $f(\boldsymbol{v}) = 0$. Now extend f to all of P by linear interpolation over the simplices in $\mathcal{T}$. Note that $f(\boldsymbol{z}) = \lambda_0 f(\boldsymbol{v}_0) + \cdots + \lambda_s f(\boldsymbol{v}_s)$, and hence $f(\boldsymbol{z}) \geq 1/(d+1)$. On the other hand, let $\boldsymbol{w}$ be the point where Π meets F_2; note that $f(\boldsymbol{w}) = 0$ since f is identically zero on F_2 (because f is defined to be zero on all vertices of F_2).

We now conclude the proof using the same technique as in Theorem 9.2. Along path Π, f is PL and continuous and decreases by at least $1/(d+1)$ (from $\boldsymbol{z}$ to $\boldsymbol{w}$). Therefore, there is a point $\boldsymbol{u}$ on Π such that f has a directional derivative at $\boldsymbol{u}$ parallel to Π whose magnitude is at least $(1/(d+1))/\operatorname{lth}(\Pi)$. Let T be the simplex containing $\boldsymbol{u}$. Then on this simplex T, since the gradient is constant, $\|\nabla f\| \geq 1/((d+1)\operatorname{lth}(\Pi))$.

We obtain an analytic expression for ∇f on T as follows. Let us number the vertices of T with $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_d$. Let $r_i = f(\boldsymbol{v}_i) - f(\boldsymbol{v}_0)$ for $i = 0, \ldots, d$. Thus, $|r_i| \leq 1$ for each i. As earlier, ∇f on T is given by $(M_T^{\mathrm{T}})^{-1}\boldsymbol{r}$, so

$$\|\nabla f\| \leq \|M_T^{-1}\| \cdot \|\boldsymbol{r}\| \leq c_d / \operatorname{minalt}(T).$$

Combining this inequality with the inequality proved in the previous paragraph proves the theorem. ∎

10. QMG aspect ratio in terms of neighboring box sizes. In this section we begin our analysis of the aspect ratio bound for QMG. In general, we cannot establish a universal constant upper bound on the aspect ratio of the triangulation produced by QMG because if P has sharp angles, then any possible triangulation, including QMG, will have poor aspect ratio near the sharp angle, as proved by Theorem 9.2. Thus, we want to show that the sharpest angle of any simplex-generated QMG is very sharp only if the input polyhedron itself has a sharp angle.

In this section we argue that the worst-case aspect ratio produced by QMG is bounded in terms of the ratio of sizes of neighboring boxes. This section requires an understanding of the analysis in our paper [12]. In subsequent sections, we bound this box-size ratio in terms of the sharpest angle. Thus, the combination of these arguments bounds the aspect ratio of QMG in terms of $\theta(P)$.

From now on, we denote the sharpest angle in P by $\theta(P)$. In particular,

$$\theta(P) = \min(1, \min\{\theta(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)\}), \tag{10.1}$$

where $\theta(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ is defined as in (9.1) and the inner min is taken over all allowable choices of $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ for (9.1). The outer min is included because, as mentioned above, we are concerned only about the case of small angles. If the minimum angle is large, then it has no impact on our bounds.

Let B be a box. As above, we define $\operatorname{size}(B)$ to be the length of a side of B. Let B, B' be two neighboring protected boxes such that $\operatorname{co}(B)$ and $\operatorname{co}(B')$ have a common point. (Here, $\operatorname{co}(B)$ refers to the content of B at the time it became protected. The contents of two neighboring boxes might not have a common point if the boxes' common subface is completely outside P because of boundaries that cut through the boxes or because of duplication.) Suppose $\operatorname{size}(B) \geq \operatorname{size}(B')$. These boxes have *box-size ratio* $\operatorname{size}(B)/\operatorname{size}(B')$. Let r be the maximum box-size ratio in the whole triangulation produced by QMG. We argue in this section that the worst aspect ratio in QMG is at most $c_d r$.

Consider a simplex T generated by QMG. As in [12], this simplex comes from a chain of $d+1$ nested box subfaces. Unlike [12], these box subfaces can have different sizes; in particular, the subfaces in the chain can grow in size as the dimension of the box face increases.

A consequence of the alignment condition presented in section 6 is as follows. Let B_i, B_{i+1} be two boxes in a chain, so that $\dim(B_i) = i$ and $\dim(B_{i+1}) = i+1$. Let the close points of these boxes be $\boldsymbol{v}_i, \boldsymbol{v}_{i+1}$. Then either $\boldsymbol{v}_i = \boldsymbol{v}_{i+1}$ or

$$\operatorname{dist}(\boldsymbol{v}_{i+1}, \operatorname{aff}(B_i)) \geq c_d \operatorname{size}(B_{i+1}). \tag{10.2}$$

The reason is as follows. Let C be the close face of B_{i+1}. Let B^* be the i-dimensional face of B_{i+1} that contains B_i. If C is a subface of B^*, then the alignment condition implies that B^* must also be protected, and hence $B^* = B_i$ and $\boldsymbol{v}_i = \boldsymbol{v}_{i+1}$. Else C is not a subface of B^*, which means that $\boldsymbol{v}_i$ is bounded away by $c_d \operatorname{size}(B_{i+1})$ from B^* as argued in [12], because the neighborhoods $N(\cdot)$ defined earlier create an exclusion zone around B^*.

Let M_T be the $d \times d$ matrix associated with T defined above, with columns ordered according to the chain order. Because of (10.2), M_T, when scaled to unit box size, satisfies analogues of the inequalities that were developed in [12] in the case of unit box size.

In particular, let S_T be the $d \times d$ diagonal matrix whose ith entry is the box side length of the ith box face in the chain defining T. Then the matrix $N = M_T S_T^{-1}$

has its columns rescaled so that each column corresponds to a difference between two vertices in a unit-size cube. Slight generalizations of the bounds proved in [12] apply to N (actually, that paper considered the transpose N^{T}). In particular, the bounds in [12] imply that $\|N\|$ and $\|N^{-1}\|$ are at most c_d.

Since $\kappa(M_T) \leq \kappa(N)\kappa(S_T)$, we have from the last paragraph that $\kappa(M_T) \leq c_d\kappa(S_T)$. Note that all the box faces in a chain come from mutual neighboring cubes, so $\kappa(S_T) \leq r$. Therefore, $\kappa(M_T) \leq c_d r$. This argument has established the following theorem.

THEOREM 10.1. *Let $\rho_{\mathrm{QMG}}(P)$ denote the worst-case aspect ratio produced by QMG when applied to polyhedral domain P. Then there exist two neighboring protected boxes B, B' such that* $\mathrm{co}(B) \cap \mathrm{co}(B') \neq \emptyset$ *and such that*

$$\rho_{\mathrm{QMG}}(P) \leq c_d \cdot \mathrm{size}(B)/\,\mathrm{size}(B').$$

In subsequent sections, we bound the maximum box-size ratio in terms of the sharpest angle $\theta(P)$. The ultimate goal is Theorem 15.1, which bounds $\rho_{\mathrm{QMG}}(P)$ in terms of $\theta(P)$.

11. A bound on splitting for alignment.

As we saw in the preceding section, the aspect ratio of QMG can be bounded if we can bound the number of times boxes are split. The following is the key theorem about how many times a box can be split. The proof of Theorem 11.1 will be the topic of this and the next few sections.

THEOREM 11.1. *Let P be the input polyhedral region, whose sharpest angle is $\theta(P)$ defined by* (10.1). *Let B be a protected box produced by QMG. Then there exists an active box B_a that is an ancestor of B such that*

$$\mathrm{size}(B_a) \leq c_d\theta(P)^{-\phi(d)}\,\mathrm{size}(B), \tag{11.1}$$

where $\phi(d) = (d-1)(d-2)/2 + 1$, *and such that* $\mathrm{co}(B_a)$ *contains an incontractible path* Π *satisfying* $\mathrm{lth}(\Pi) \leq c_d\,\mathrm{size}(B_a)$. *We call B_a the* anchor *of B.*

Recall that QMG splits boxes in both the separation and alignment stages. The purpose of this section is to show that the amount of splitting for alignment is bounded by c_d, which is one step in the proof of Theorem 11.1. We start with two preliminary lemmas, which lead to the main result, Lemma 11.4, at the end of this section. That lemma is one step in the proof of Theorem 11.1.

LEMMA 11.2. *Let B be a box with a neighbor B' such that* $\mathrm{co}(B) \cap \mathrm{co}(B') \neq \emptyset$. *There is a constant c_d such that if* $\mathrm{size}(B') \leq c_d\,\mathrm{size}(B)$, *then* $\mathrm{co}(B') \subset \mathrm{co}(B)$.

Proof. Let $s = \mathrm{size}(B)$ and $s' = \mathrm{size}(B')$. Then $\mathrm{ex}(B)$ extends out by γs from all sides of B. Hence $\mathrm{ex}(B)$ contains any point within ∞-norm distance γs from B. In particular, if $(1+\gamma)s' \leq \gamma s$, then $\mathrm{ex}(B')$ would be completely contained in $\mathrm{ex}(B)$; let c_d in the lemma be this factor $\gamma/(1+\gamma)$. Let $\boldsymbol{x}$ be a point in $\mathrm{co}(B') \cap \mathrm{co}(B)$. Then every point in $\mathrm{co}(B')$ is reachable by a PL path in $\mathrm{co}(B')$ from $\boldsymbol{x}$. This means that all of $\mathrm{co}(B')$ is contained in the component of $P \cap \mathrm{ex}(B)$ that contains $\boldsymbol{x}$, which must be $\mathrm{co}(B)$. ∎

LEMMA 11.3. *Let B be a box that is split for alignment: in particular, say B is split during processing of O_F in phase k for some face F. Then there exists another active box B^* created by QMG such that* (1) $\mathrm{size}(B^*) = \mathrm{size}(B)$, (2) *$B^*$ was split before the phase k alignment stage (i.e., B^* was split during phases $0, \ldots, k-1$ or in the phase k separation stage), and* (3) *there is a PL path in P from* $\mathrm{co}(B)$ *to* $\mathrm{co}(B^*)$ *of length at most* $c_d\,\mathrm{size}(B)$.

Proof. Let us first prove the lemma for the simplified version of QMG in which all the boxes are full dimensional. In this case, the alignment condition was described in section 6 as follows: B is split for alignment because its high-priority close face, say, C, is not completely covered by (full-dimensional) boxes in the extended orbit of F the same size or larger when B is processed.

Consider the collection of boxes obtained by taking all boxes produced by any step of simplified QMG that are the same size as B. Consider also all those boxes larger than B that are leaf boxes (i.e., protected). Notice that this collection of boxes, say, Q, completely covers the input domain; some parts of $\mathbf{R}^d$ could be double covered because of duplication.

Let the enumeration of all boxes in Q that cover C be denoted $B_1, \ldots, B_m$; exclude B itself from this enumeration. Since F is close to C, F meets $N(C)$ and hence also $\mathrm{co}(B)$. Let $\boldsymbol{x}$ be a point where F meets $N(C)$. Among $B_1, \ldots, B_m$, consider only those B_i such that $\boldsymbol{x} \in \mathrm{co}(B_i)$. The case when $\boldsymbol{x} \notin \mathrm{co}(B_i)$ could occur only because of duplication; there could be duplicates of neighbors of B that do not contain $\boldsymbol{x}$.

Rename the remaining boxes again as $B_1, \ldots, B_m$; note that these boxes together with B must cover C. Since the alignment condition does not hold for B, one of them, say, B_i, is already split at the time B is processed. The box that is already split must have the same size as B (i.e., it cannot be one of the larger boxes in Q because those boxes are all protected). Without loss of generality, B_i is the earliest box among $B_1, \ldots, B_m$ to be split.

Case 1 is that this box B_i was split during a phase $0, \ldots, k-1$, or during phase k separation. In this case the lemma is proved with $B^* = B_i$; note that the two boxes contain $\boldsymbol{x}$ in their content so that condition (3) of the lemma is trivial.

The remaining case, case 2, is that B_i is split during the phase k alignment stage before B is processed. Rename B_i as B'. Note that since $\mathrm{co}(B')$ meets F, then B' must be in O_F. Since B' is the first box among $B_1, \ldots, B_m$ to be split during phase k alignment, C is still covered by boxes in the orbit of the same size or larger at the time the alignment condition is checked for B'. Let C' be the close face of B'. Note that C' is not covered by boxes in the orbit of the same size or larger, since we are assuming B' is split for alignment in phase k. It is not possible that $\dim(C') \geq \dim(C)$ because then C would have higher priority than C' and hence C' would not be selected as the close face of B'. (Recall that the priority rule favors faces of lower dimension and, among faces of the same dimension, favors faces completely covered by boxes in the orbit.) Thus, $\dim(C') < \dim(C)$. Start the proof of this lemma over again with B' and C'. In other words, consider the boxes in the quadtree at the level of B' that cover C'. Either for B' we will "exit" this argument in case 1 (i.e., we find a box B^* that satisfies the lemma for B'), or we will have to restart the argument another time.

But note that each time we restart the above argument, the dimension of the close face in question decreases by 1. Thus, we can repeat the argument at most d times before terminating at case 1. Let the sequence of boxes constructed by repeating this argument be $B, B', B'', \ldots, B^{(r)}, B^*$, where B^* is a box split before phase k alignment. Note that $r \leq d$ as just mentioned. Also, all boxes in this sequence are the same size, and $B^{(i)}$ is adjacent to $B^{(i+1)}$ for each i. Furthermore, $\mathrm{co}(B^{(r)})$ has a common point with $\mathrm{co}(B^*)$, and all of $B^{(1)}, \ldots, B^{(r)}$ are in O_F. This means that we can find a PL path in P from B to $B^{(r)}$ by traversing F through each box. (Recall that a box in O_F cannot meet any boundaries of F in its content.) The length of the PL path constructed in this manner is at most $c_d\,\mathrm{size}(B)$. This proves the lemma.

If we wanted to extend this proof to the case of the complete version of QMG (including lower-dimensional boxes), we would use the same proof as above, except that we would have to restate the meaning of "completely covered" in terms of the weight system mentioned in section 7. Because we have incompletely described the weight system and skipped the lemmas showing that lower-dimensional boxes do indeed act like proxies for the full-dimensional boxes that contain them, we do not have enough machinery to prove this lemma in the general case; hence we merely assert it. ∎

The preceding lemma now leads to the main result of this section, which says that during splitting for alignment, boxes can become only a constant factor smaller.

LEMMA 11.4. *Let B be a box that results from splitting for alignment during the processing of O_F for some k-face F. Then B is descended from an active box B' at the start of the phase k alignment stage such that* $\operatorname{size}(B') \le c_d \operatorname{size}(B)$.

Proof. Let B_0 be a parent of B, so that B_0 was split for alignment during the processing of O_F and so that $\operatorname{size}(B_0) = 2\operatorname{size}(B)$. By the preceding lemma, there is another box B^* that was either protected from an earlier phase or was split for separation such that there is a PL path Π in P from $\operatorname{co}(B^*)$ to $\operatorname{co}(B_0)$ of length at most $c_d \operatorname{size}(B_0)$.

Let B' be the ancestor of B_0 at the beginning of phase k alignment in O_F. Observe that there is a constant χ_d depending on d such that if B' satisfied $\operatorname{size}(B') \ge \chi_d \operatorname{size}(B_0)$, then $\operatorname{co}(B')$ would contain $\operatorname{co}(B^*)$ as a subset. This follows from the same proof technique used for Lemma 11.2; in particular, if B' were sufficiently larger than B_0, it would contain the whole path Π and also $\operatorname{co}(B^*)$.

On the other hand, it is impossible that $\operatorname{co}(B')$ contains $\operatorname{co}(B^*)$. This is because B^* was split for separation in phase k or was split in a phase earlier than k. Whatever P-faces caused B^* to be split would cause B' to be crowded, and hence B' could not end up in O_F.

Thus, we conclude $\operatorname{size}(B') < \chi_d \operatorname{size}(B_0)$, which proves the lemma. ∎

This lemma shows that all splitting for alignment can be lumped into the factor c_d in (11.1).

12. Splitting boxes for weak crowding.

Recall that a box is split for separation if and only if it is crowded. Recall also that there are two ways that B can be crowded in phase k: (1) $\operatorname{co}(B)$ contains a P-face of dimension $k-1$ or lower, or (2) $\operatorname{co}(B)$ contains a P-face F of dimension k, and another P-face G of dimension k or greater that is not a superface of F.

We call the former "weak crowding" and the latter "strong crowding." In this section we show that all splitting for weak crowding can also be lumped into the factor c_d in (11.1), which is another step toward proving Theorem 11.1.

LEMMA 12.1. *Let B be a box that is split for weak crowding; that is, in the phase k separation stage,* $\operatorname{co}(B)$ *meets a P-face of dimension $k-1$ or lower. Then B has an ancestor B_0 that is an active box at the beginning of phase k such that* $\operatorname{size}(B) \ge c_d \operatorname{size}(B_0)$.

Proof. The assumption implies that the P-face F of dimension $k-1$ or lower meets $\operatorname{ex}(B)$. Let B_0 be the ancestor of B from the beginning of phase k. We claim that B_0 can be at most a constant factor c_d larger than B. This is because the expansion factors for $\operatorname{ex}(B)$ and $\overline{\operatorname{ex}}(B)$ are off by a constant c_d. Recall that $\overline{\operatorname{ex}}(B)$ was defined in section 6 and is applied here with respect to face F. Therefore, the ancestor of B_0, if it is much larger than B, would contain this P-face F in $\overline{\operatorname{ex}}(B_0)$. Recall that if F meets $\overline{\operatorname{ex}}(B_0)$, then B_0 has a subface close to F. Hence B_0 would have been protected in phase $\dim(F)$, which is less than k, or would have been split for separation. Thus, B_0 is at most c_d larger than B. ∎

Thus, all splitting for weak crowding can also be lumped into the factor c_d in (11.1).

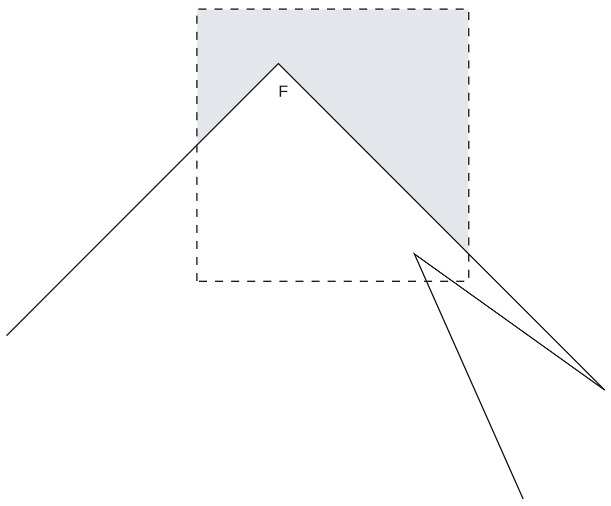


FIG. 13.1. *Lemma* 13.1 *in the case $k = d = 2$. Face F is a single vertex. The boundary of P is the solid line. The convex set C is the dashed square in the figure. The shaded region is U. Notice that every point in U is visible to F.*

13. Splitting for strong crowding.

13. Splitting for strong crowding. In this section we analyze splitting for strong crowding and finally prove Theorem 11.1. Recall that "strongly crowded" means that there is a P-face F of dimension k in co(B), and another P-face G of dimension $l \geq k$ in co(B) that is not a superface of F. From now on, we say that G is "foreign" to F if G is not a superface of F. We say that two points $\boldsymbol{x}$ and $\boldsymbol{y}$ are "visible" to each other with respect to P if segment $\boldsymbol{xy}$ lies in P. We start with two lemmas about visibility.

LEMMA 13.1. *Let P be a k-dimensional polyhedral domain in $\mathbf{R}^d$, and let C be a closed convex subset of $\mathbf{R}^d$. Suppose $P \cap C$ is not empty, and let U be a component of $P \cap C$. Suppose that U meets a P-face F and that U does not meet any faces of P that are foreign to F. Then every point in U is visible to every point in $F \cap U$, where "visibility" is with respect to U.*

See Figure 13.1 for an illustration of this lemma.

Proof. Let $\boldsymbol{x}$ be a point in $F \cap U$ and $\boldsymbol{y}$ a point in U. Consider the segment $L = \boldsymbol{xy}$; suppose that this segment is not contained in U. We will derive a contradiction. Since $\boldsymbol{y} \in U$ and U is closed, there must be some point $\boldsymbol{z} \in L$ different from $\boldsymbol{x}$ such that $\boldsymbol{z}$ is in U, but there is a sequence of points $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$ lying on L and converging to $\boldsymbol{z}$ that are not in U. Note that all of these points lie in C because C is convex and L joins two points in C. Thus, since these points are not in $P \cap C$, we conclude that they are not in P. This means that there is at least one facet H of P (where "facet" refers to a face of dimension $k-1$) passing through $\boldsymbol{z}$ such that aff(H) does not contain L as a subset.

But this is impossible, because every facet of P meeting U in this component is a superface of F by assumption. This means in particular that for H in the last paragraph, $\boldsymbol{x} \in$ aff(H). But since aff(H) is convex and contains $\boldsymbol{x}$ and $\boldsymbol{z}$, it also contains L. ∎

LEMMA 13.2. *Let P be a k-dimensional polyhedral domain in $\mathbf{R}^d$, and let C be a closed convex subset of $\mathbf{R}^d$. Suppose $P \cap C$ is not empty, and let U be a component of $P \cap C$. Suppose that U meets a P-face F, and suppose that U also meets a face*

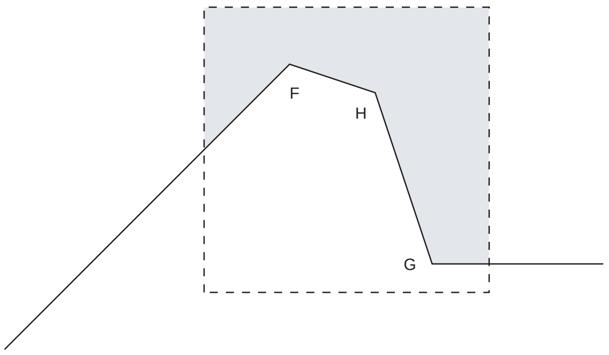


Fig. 13.2. *Lemma 13.2 for $d = 2$. Face F is a single vertex. The convex set C is the dashed square in the figure. The shaded region is U. Note that there is a vertex G of P that is foreign to F. This means that there is a face, namely, vertex H in U, that is also foreign to F but is visible to $F \cap U$.*

G of P foreign to F. Then for any point $\boldsymbol{x} \in F \cap U$, there is a point $\boldsymbol{y} \in U$ that is visible to $\boldsymbol{x}$ (with respect to U) such that $\boldsymbol{y}$ lies on a P-face foreign to F (which may or may not be G).

See Figure 13.2 for an illustration.

Proof. For $\lambda \in [0, 1]$, let $C(\lambda)$ denote the contraction by λ of C toward $\boldsymbol{x}$ (i.e., $\boldsymbol{v} \in C$ iff $\lambda\boldsymbol{v} + (1-\lambda)\boldsymbol{x} \in C(\lambda)$). Let $U(\lambda)$ be the component of $C(\lambda) \cap P$ that contains $\boldsymbol{x}$. Find the parameter value $\lambda^* > 0$, such that $U(\lambda^*)$ still meets a foreign face, but $U(\lambda^* - \epsilon)$ meets no faces foreign to F for all small $\epsilon > 0$. By the preceding lemma, every point in $U(\lambda^* - \epsilon)$ is visible to $\boldsymbol{x}$. Since the set of points visible to $\boldsymbol{x}$ is closed, this means that every point in $U(\lambda^*)$ is also visible to $\boldsymbol{x}$, and this set includes a point from a foreign face. $\square$

We now conclude the proof of Theorem 11.1. Let B be a protected box that is produced by QMG. Write down its sequence of ancestors $B_0, B_1, \ldots, B_r$, where $B_r = B$ and B_0 is the top box. This sequence is not necessarily unique if B_r is not full dimensional, in which case any sequence of ancestors will do. Now, delete boxes B_i in this sequence such that B_{i+1} arises from B_i via subface launching. Denote the new list $B_0, \ldots, B_r$ again. Each box is now a factor 2 smaller than its predecessor. From this list, delete boxes that are split either for alignment or for weak crowding, and denote the new list again as $B_0, B_1, \ldots, B_r$. This new list contains boxes that are split only for strong crowding, as well as the protected box B_r, which is not split. By Lemmas 11.4 and 12.1, in this new sequence of boxes, each box differs from its predecessor in size by a factor at most c_d.

Each box $B_0, \ldots, B_{r-1}$ is split during some phase 0 to $d - 1$. (There cannot be any strong crowding in phase d by definition of strong crowding.) Therefore, mark the location where each phase begins and ends in the sequence $B_0, \ldots, B_{r-1}$. This divides the sequence $B_0, \ldots, B_{r-1}$ into *periods*, where the kth period consists of boxes split during phase k.

Now subdivide each period into *subperiods*, using the following procedure. Focus on one particular period k, and suppose it starts at B_l and ends with B_m (i.e., B_l is the first box of the sequence split in phase k, and B_m is the last). Since B_m is strongly crowded, there is a k-face of P, say, F, meeting $\mathrm{co}(B_m)$, and there is another face G foreign to F meeting $\mathrm{co}(B_m)$. Pick a point $\boldsymbol{x} \in F \cap \mathrm{co}(B_m)$. Without loss of generality, by Lemma 13.2, we can assume that there is a segment in $\mathrm{co}(B_m)$ from $\boldsymbol{x}$

to a point $\boldsymbol{y} \in G$ (else choose a different G). Similarly, without loss of generality, $\boldsymbol{y}$ is not in any proper subface of G. (If $\boldsymbol{y}$ is in a proper subface of G, simply reselect G to be the subface: because G is foreign to F, every subface of G is also foreign to F.)

We will construct a PL path Π in $\mathrm{co}(B_m)$. The first segment of the path is $\boldsymbol{xy}$. Consider whether G has any boundary faces that meet $\mathrm{co}(B_m)$. If not, then the construction of Π is complete, and we let $\Pi = \boldsymbol{xy}$. The other case is that a boundary of G meets $\mathrm{co}(B_m)$. By Lemma 13.2, there is a segment in $\mathrm{co}(B_m)$ from $\boldsymbol{y}$ to a boundary face of G. (In this application of Lemma 13.2, the "polyhedral domain" in the lemma is G itself. Note that a boundary of G is foreign to G, i.e., is not a superface.) Append this new segment to Π. We can continue in this manner until we reach a point to which we will now reassign the name $\boldsymbol{y}$, such that $\boldsymbol{y}$ lies on P-face foreign to F, which we rename G, such that G does not have any boundaries that meet $\mathrm{co}(B_m)$. Note that $\mathrm{lth}(\Pi)$ is at most $d \cdot \sqrt{d}(1+\gamma)\,\mathrm{size}(B_m)$. The factor $\sqrt{d}(1+\gamma)\,\mathrm{size}(B_m)$ is the diameter of $\mathrm{ex}(B_m)$ (in the worst case when B_m is full dimensional) and hence is the maximum length of any segment in $\mathrm{co}(B_m)$, and the factor d comes from the fact that Π has at most d segments in it. This is because in the preceding construction of Π, each time a new segment is added, the dimension of the boundary face in question decreases by at least 1. Thus, $\mathrm{lth}(\Pi) \le c_d\,\mathrm{size}(B_m)$.

Note that $\Pi \subset \mathrm{co}(B_i)$ for each $i = l, \ldots, m$, since $\mathrm{co}(B_m) \subset \mathrm{co}(B_{m-1}) \subset \cdots \subset \mathrm{co}(B_l)$. On the other hand, in an ancestor of B_m, it might be possible to extend Π with one or more additional segments so that it reaches a lower-dimensional boundary face of G. Find the lowest numbered box B_q (largest in size) in the period $B_l, \ldots, B_m$ such that it is *not* possible to extend Π to a P-face of lower dimension that is a boundary of G. We will say that the subsequence $B_q, B_{q+1}, \ldots, B_m$ is one *subperiod* of period k. Each box in this subperiod is associated with the quintuple $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ defined in the last paragraph. If $q = l$, we are done; this is the only subperiod of period k.

On the other hand, if $q > l$, then in B_{q-1} it is possible to extend Π to reach a face of lower dimension than was reached in B_q. Extend Π with one or more additional segments, yielding a path Π' to a face G' that is a proper subface of G. Now we repeat the above argument to find the predecessor of B_{q-1}, say, $B_{q'}$, such that Π cannot be extended in $B_{q'}$ but can be extended in $B_{q'-1}$ (or else $q' = l$), and we let $B_{q'}, B_{q'+1}, \ldots, B_{q-1}$ be another subperiod of period k, associated with $(\boldsymbol{x}, \boldsymbol{y}', F, G', \Pi')$. We continue in this manner until we finally get back to B_l. The maximum number of subperiods in period k is seen to be $d - k$. The reason is that each time we back up to a new subperiod, the dimension of the face reached by Π decreases by at least 1. The maximum possible dimension of G initially is $d - 1$, and the minimum possible dimension is k (because no box among $B_l, \ldots, B_m$ is weakly crowded).

Thus, we have divided the sequence of boxes $B_0, \ldots, B_{r-1}$ into at most d periods numbered $0, \ldots, d-1$, and period k is divided into at most $d - k$ subperiods. We have also associated with each box a choice of $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$.

Now we can classify each box in the sequence according to whether its associated path Π is contractible or not. Let B_a be the highest numbered (smallest) box in $B_0, \ldots, B_{r-1}$ such that its associated path is incontractible. (Notice that B_a exists because B_0 certainly satisfies this condition.) Thus, B_a satisfies the conditions of Theorem 11.1 that it contains an incontractible path of length at most $c_d\,\mathrm{size}(B_a)$, and that it is an ancestor of B. All that remains is to establish (11.1). Note that the anchor box must be the last one in its subperiod, because all boxes in a subperiod have the same associated path. From now on, we consider only the portion $B_a, \ldots, B_r$

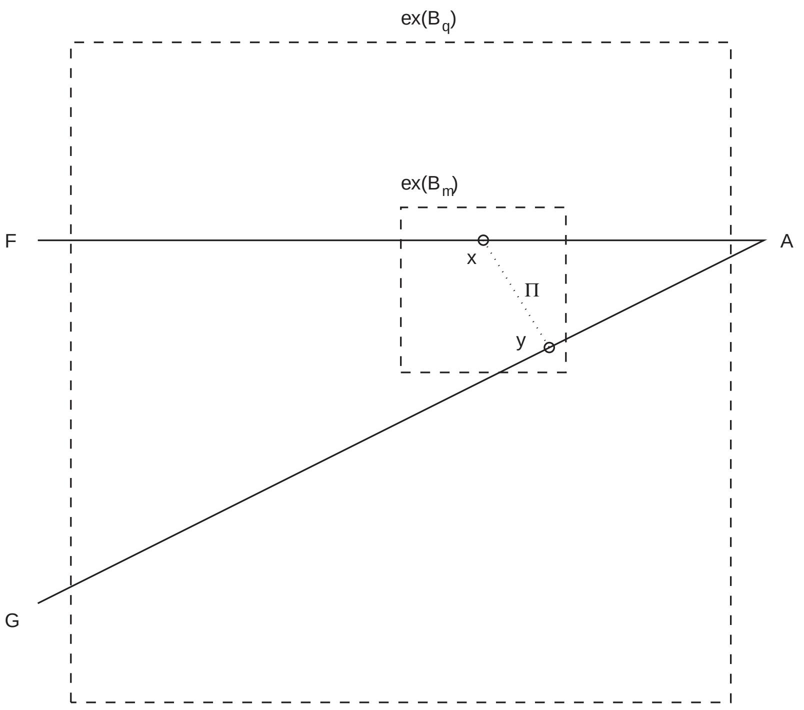


FIG. 13.3. *Items $\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi, A$ arising in the proof of* (13.1). *The boundaries of* $\mathrm{ex}(B_q)$ *and* $\mathrm{ex}(B_m)$ *are the dashed lines. In this figure A is zero dimensional and is equal to $F \cap G$, but in general A will be a superset of $F \cap G$.*

of the original sequence.

We start with the following intermediate result. Let B_m be a box in the sequence with $m > a$, and suppose it is contained in a subperiod beginning with B_q (so $a < q \leq m$). Then we claim

$$\mathrm{size}(B_m) \geq c_d \cdot \theta(P) \cdot \mathrm{size}(B_q). \tag{13.1}$$

Let us assume that B_m is a proper descendent of B_q because when $m = q$, (13.1) is trivially true. Let $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ be the quintuple associated with B_m; by definition of subperiod, the same quintuple is associated with B_q. Since Π is contractible (by choice of B_a), $F \cap G$ is nonempty. Let A denote $\mathrm{aff}(F) \cap \mathrm{aff}(G)$. We claim that A does not meet $\mathrm{ex}(B_q)$. See Figure 13.3 for an illustration of the items constructed in the proof of this claim. Note that no boundary face of F meets $\mathrm{co}(B_q)$, because $\mathrm{co}(B_q)$ does not meet any P-faces of dimension $k-1$ or less (because it is not weakly crowded). Thus, $\mathrm{co}(B_q)$ must contain all of $\mathrm{aff}(F) \cap \mathrm{ex}(B_q)$. Similarly, $\mathrm{co}(B_q)$ does not contain any boundary faces of G by construction of Π (else Π could be extended). Thus, $\mathrm{co}(B_q)$ also contains all of $\mathrm{aff}(G) \cap \mathrm{ex}(B_q)$. Suppose that $\mathrm{aff}(F) \cap \mathrm{aff}(G)$ met $\mathrm{ex}(B_q)$; then $\mathrm{co}(B_q)$ would have to contain all of $\mathrm{aff}(F) \cap \mathrm{aff}(G) \cap \mathrm{ex}(B_q) = A \cap \mathrm{ex}(B_q)$ by the foregoing argument. In particular, F and G would meet in $\mathrm{co}(B)$ at all points in $A \cap \mathrm{ex}(B_q)$. But this is impossible, because neither has any boundaries in $\mathrm{co}(B_q)$.

Since A does not meet $\mathrm{ex}(B_q)$, there is a lower bound of the form $c_d\,\mathrm{size}(B_q)$ on the distance from A to $\mathrm{ex}(B_m)$. This is because $\mathrm{ex}(B_q)$ extends a small fraction c_d multiplied by $\mathrm{size}(B_q)$ beyond $\mathrm{ex}(B')$ for any proper descendant B' of B_q. In particular, this means $\operatorname{dist}(\boldsymbol{x}, A) \geq c_d \operatorname{size}(B_q)$ and $\operatorname{dist}(\boldsymbol{y}, A) \geq c_d \operatorname{size}(B_q)$. On the other hand, $\operatorname{lth}(\Pi) \leq c_d \operatorname{size}(B_m)$, as argued above. Thus, in the definition of sharp angle (9.1) applied to $(\boldsymbol{x}, \boldsymbol{y}, \Pi)$, we see that F and G make an angle less than or equal to $c_d \operatorname{size}(B_m)/\operatorname{size}(B_q)$. Since $\theta(P)$ is the sharpest angle,

$$\theta(P) \leq c_d \operatorname{size}(B_m)/\operatorname{size}(B_q).$$

This equation proves (13.1).

We now deduce (11.1) from (13.1). If B_q is the beginning of a subperiod with $q > a$, and B_m is its end, then $\operatorname{size}(B_q) \leq c_d\theta(P)^{-1} \operatorname{size}(B_m)$ from (13.1). Thus, if ϕ stands for the total number of subperiods between B_{a+1} and B_r, then $\operatorname{size}(B_a) \leq (c_d\theta(P))^{-\phi} \operatorname{size}(B_r)$, where $c_d^{-\phi}$ accounts for the factor in box size shrinkage between the end of one subperiod and the beginning of the next, and $\theta(P)^{-\phi}$ accounts for box shrinkage within the ϕ subperiods. Since there are at most $d - k$ subperiods in period k, the total number of subperiods ϕ can be bounded $\phi(d) = d(d+1)/2$. Thus, $\operatorname{size}(B_a) \leq c_d\theta(P)^{-d(d+1)/2} \operatorname{size}(B_r)$ (where we have renamed $c_d^{-\phi(d)}$ as c_d).

In fact, we can immediately improve this estimate on $\phi(d)$ with the following observation. Note that if B_i is in period 0, then its path Π constructed above cannot be contractible. This is because F in phase 0 is a vertex and hence is disjoint from any foreign face G. Thus, the anchor box B_a either is the last box of period 0 or is in a later period. This means that the only subperiods that matter are in period 1 or later. Thus, we can improve the estimate to $\phi(d) = (d-1)d/2$.

We can further improve the estimate to (13.2) below with the following more complicated analysis. We claim that in period 1, a single subperiod suffices. The proof is as follows. Assume that the anchor box is in period 0 or 1 (else we would not need to include subperiods of period 1 in the count of ϕ, so (13.2) holds already). Let us review why we constructed subperiods in the first place. Let B_m be the box at the end of a subperiod, let $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ be its associated quintuple such that Π is contractible, and let B_q be the first box in the subperiod ending at B_m. In the above derivation of (13.1), we used the fact $\operatorname{aff}(F) \cap \operatorname{aff}(G)$ cannot meet $\operatorname{ex}(B_q)$. To derive this fact, we needed to know that G does not have any boundaries in $\operatorname{co}(B_q)$. The above method of constructing subperiods indeed assures that G does not have boundaries in $\operatorname{co}(B_q)$.

But consider the special case of period 1, so that $\dim(\operatorname{aff}(F)) = 1$. Let B_m be the last box in period 1, and redefine B_q to be the first box of period 1, or the child of the anchor B_a, whichever comes later. Let $(\boldsymbol{x}, \boldsymbol{y}, F, G, \Pi)$ be the quintuple for B_m. Since $\dim(F) = 1$, $\dim(\operatorname{aff}(F) \cap \operatorname{aff}(G))$ is either 0 or 1. The case when $\dim(\operatorname{aff}(F) \cap \operatorname{aff}(G)) = 1$ cannot occur by the way we construct Π. In particular, if $\dim(\operatorname{aff}(F) \cap \operatorname{aff}(G)) = 1$, then $\operatorname{aff}(F) \subset \operatorname{aff}(G)$, which means that G either is a superface of F (contradicting the choice of G as a foreign face) or has a boundary in $\operatorname{co}(B_m)$ (contradicting the fact that Π reaches a face of minimal dimension).

The other case is that $\dim(\operatorname{aff}(F) \cap \operatorname{aff}(G)) = 0$; in other words, $\operatorname{aff}(F) \cap \operatorname{aff}(G)$ is a single point $\{\boldsymbol{v}\}$. Since Π is contractible, F and G have a common subface which must therefore be $\{\boldsymbol{v}\}$ itself. Thus $\{\boldsymbol{v}\}$ is a face of P. But since B_q is not weakly crowded, $\operatorname{co}(B_q)$ cannot contain a 0-dimensional face of P. Thus, without making any assumption about whether G has boundaries in $\operatorname{co}(B_q)$, we have determined that $\operatorname{aff}(F) \cap \operatorname{aff}(G)$ does not meet $\operatorname{ex}(B_q)$. Therefore, a single subperiod suffices for period 1.

Thus, we have the following improved estimate for ϕ, which is the total number

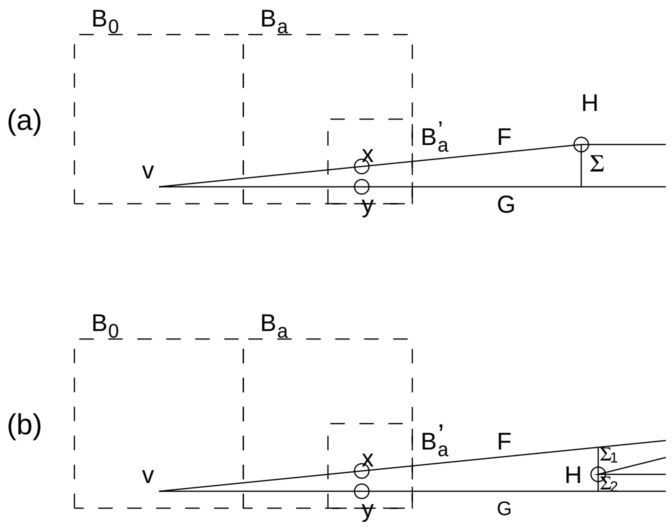


FIG. 13.4. *Faces arising in the proof of the strengthened version of Corollary* 13.3 *when* $d = 2$. *Box* B *and path* Π *are not depicted; both are enclosed in* B'_a *in the figure.*

of subperiods after B_a:

$$\phi(d) = (d-1)(d-2)/2 + 1. \tag{13.2}$$

This concludes the proof of Theorem 11.1.

Notice that by combining Theorems 9.4 and 11.1 we immediate obtain the following corollary.

COROLLARY 13.3. *Let* B *be a protected box generated by QMG. Then there exists an ancestor* B_a *of* B *such that* (11.1) *holds and such that for any triangulation* $\mathcal{T}$ *of* P*, there is a simplex* T *meeting* $\operatorname{co}(B_a)$ *such that* $\operatorname{minalt}(T) \le c_d \operatorname{size}(B_a)$.

In fact, we can strengthen Corollary 13.3 (though not Theorem 11.1) in the case $d = 2$. The strengthened version of Corollary 13.3 asserts that

$$\operatorname{size}(B_a) \le c \operatorname{size}(B) \tag{13.3}$$

holds when $d = 2$, in place of (11.1) (i.e., the factor of $\theta(P)^{-1}$ goes away). The argument for this strengthening is as follows. In the following argument, c denotes an absolute constant whose value may change from formula to formula.

Consider how the factor $\theta(P)^{-1}$ arises in the first place. Let B be a protected box and B_a its anchor. This factor comes when anchor B_a is from period 0, and then in period 1 we split a strongly crowded box that has a contractible path that defines a sharp angle θ. Since (11.1) and (13.3) are equivalent when $\theta(P)$ is large, let us assume that $\theta \le 0.1$. Let B'_a be the last box that is split for strong crowding at the end of period 1. Clearly (13.3) holds for this choice of B'_a (since splitting for alignment, as well as all splitting in phase 2, incurs only an additional factor c). The contractible path Π in B'_a is from $\boldsymbol{x}$ to $\boldsymbol{y}$. See Figure 13.4. We now must prove that for an arbitrary triangulation $\mathcal{T}$, there is a simplex T meeting Π satisfying $\operatorname{minalt}(T) \le c_d \operatorname{size}(B'_a)$.

In the figure, B_0 denotes a protected box for $\boldsymbol{v}$, the common subface of F and G. Note that there must be an incontractible path from $\boldsymbol{v}$ to a foreign face H whose length is at most a factor c more than $\operatorname{size}(B_0)$ by the preceding analysis. Thus, $\boldsymbol{x}$

and $\boldsymbol{y}$ must both be separated from $\boldsymbol{v}$ by at least $c_d \operatorname{dist}(\boldsymbol{v}, H)$. There are two possible ways to choose H; either it is a subface of one of F or G, which is (a) in the figure, or it is does not meet F and G, which is (b) in the figure.

Let $\mathcal{T}$ be an arbitrary triangulation. Let $T_1, \ldots, T_s$ be the triangles of T that meet Π. Take two cases. In the first case, suppose that at least one of $T_1, \ldots, T_s$, say, T_1, meets the segment denoted by Σ in (a) or that it meets the PL path $\Sigma_1 \cup \Sigma_2$ in (b). Since $\Sigma, \Sigma_1, \Sigma_2$ are all incontractible, this means by Theorem 9.4 that $\operatorname{minalt}(T_1) \le c \operatorname{lth}(\Sigma)$ for (a) or $\operatorname{minalt}(T_1) \le c \operatorname{lth}(\Sigma_1 \cup \Sigma_2)$ for (b). But now it is clear that $\operatorname{lth}(\Sigma), \operatorname{lth}(\Sigma_1), \operatorname{lth}(\Sigma_2)$ are all at most $c\theta \operatorname{dist}(\boldsymbol{v}, H)$, that is, less than or equal to $c \operatorname{size}(B'_a)$. Thus, $\operatorname{minalt}(T_1) \le c \operatorname{size}(B'_a)$, proving the corollary.

The other case is that none of $T_1, \ldots, T_s$ meet Σ in (a) or $\Sigma_1 \cup \Sigma_2$ in (b). But this means all of $T_1, \ldots, T_s$ are contained in the polygonal region bounded by $F \cup G \cup \Sigma$ in (a) or $F \cup G \cup \Sigma_1 \cup \Sigma_2$ in (b). The width of this polygonal region is at most $c \operatorname{size}(B'_a)$, so any triangle in this region has minalt at most $c \operatorname{size}(B'_a)$. This concludes the proof that Corollary 13.3 may be strengthened in the case $d = 2$.

14. Maximizing the minimum altitude.

In this section we consider the problem of computing a triangulation that maximizes the minimum altitude. Although this is not the problem for which QMG is intended, we nonetheless can obtain an interesting consequence from Corollary 13.3. Suppose we want to compute the triangulation of P that maximizes the minimum altitude. In other words, for a triangulation $\mathcal{T}$ of P, define

$$\mu_{\mathcal{T}}(P) = \min\{\operatorname{minalt}(T) : T \in \mathcal{T}\}$$

and then consider the triangulation $\mathcal{T}^*$ that solves

$$\mu(P) = \max\{\mu_{\mathcal{T}}(P) : \mathcal{T} \text{ is a triangulation of } P\}.$$

It follows from Corollary 13.3 that QMG solves this problem to within a factor $c_d \theta(P)^{-\phi(d)}$ for $d > 2$ and within a factor c (a universal constant) when $d = 2$. This is because the minimum altitude among all triangles produced by QMG is within a factor c_d of the smallest protected box generated by QMG. But, by the corollary, the smallest protected box is within a factor of $c_d \theta(P)^{-\phi(d)}$ of the minimum altitude of any possible triangulation.

Thus, in the case $d = 2$, $\mu_{\text{QMG}}(P) \ge c\mu(P)$, and for $d = 3$, $\mu_{\text{QMG}}(P) \ge c\theta(P)^2 \mu(P)$. In the case $d = 2$, more is known about this problem. In particular, the algorithm of Bern, Dobkin, and Eppstein [2] produces a triangulation $\mathcal{T}$ also satisfying $\mu_{\mathcal{T}}(P) \ge c\mu(P)$, and also $\mathcal{T}$ has an optimal (linear) number of triangles.

In the case $d = 3$, much less is known. For instance, we do not know whether the bound $c\theta(P)^{-2}$ is tight for QMG. We have constructed an example where the minimum altitude of the triangulation produced by QMG is off from $\mu(P)$ by a factor $c\theta(P)^{-1}$, but we have not found an example attaining the bound $c\theta(P)^{-2}$.

Another open question concerns a geometric characterization of $\mu(P)$. It follows from the optimality of QMG in the case $d = 2$ that $\mu(P)$ is within a constant factor of the minimum geodesic distance between P-faces that do not meet each other. Is there a similar simple geometric characterization of $\mu(P)$ for $d = 3$? For any d, Theorem 9.4 implies that the minimum geodesic distance between two nonmeeting faces of P is an upper bound on $\mu(P)$ (to within a constant c_d), but it is not known whether this bound is tight.

15. A bound on the QMG aspect ratio. This section establishes the optimality of the QMG aspect ratio using Theorem 11.1 and Corollary 13.3.

THEOREM 15.1. *Let $\rho_{\mathrm{QMG}}(P)$ denote the worst-case aspect ratio produced by QMG when applied to polyhedral domain P. Then*

$$\rho_{\mathrm{QMG}}(P) \le c_d \theta(P)^{-\phi(d)}.$$

Proof. Let B_1, B_2 be two neighboring protected boxes such that $\mathrm{co}(B_1)$ and $\mathrm{co}(B_2)$ have a common point. Assume that B_1 is protected in phase k and lies in O_F, assume B_2 is protected in phase l and lies in O_G, and assume $\mathrm{size}(B_1) > \mathrm{size}(B_2)$. By Theorem 10.1, it suffices to obtain an upper bound on $\mathrm{size}(B_1)/\,\mathrm{size}(B_2)$.

We can assume that $\mathrm{co}(B_2) \subset \mathrm{co}(B_1)$. If this relation did not hold, then by Lemma 11.2 we could immediately conclude that there is a bound of the form c_d on $\mathrm{size}(B_1)/\,\mathrm{size}(B_2)$.

By Theorem 11.1, B_2 has an anchor B_a containing an incontractible path such that

$$c_d \theta(P)^{-\phi(d)}\,\mathrm{size}(B_2) \ge \mathrm{size}(B_a). \tag{15.1}$$

Because B_1 is protected, every point in $\mathrm{co}(B_1)$ is visible to every point in $F \cap \mathrm{co}(B_1)$ by Lemma 13.1, and this includes $\mathrm{co}(B_2)$ as well. Therefore, any PL path inside $\mathrm{co}(B_1)$ is contractible to any point of $F \cap \mathrm{co}(B_1)$. Since $\mathrm{co}(B_a)$ contains an incontractible path, it cannot be a subset of $\mathrm{co}(B_1)$. Hence, $\mathrm{size}(B_a) \ge c_d\,\mathrm{size}(B_1)$. Combining this with (15.1) shows that $\mathrm{size}(B_1)/\,\mathrm{size}(B_2) \le c_d \theta(P)^{-\phi(d)}$. ◻

When $d = 2$, Theorem 15.1 shows that QMG is optimal because we already know that for any triangulation $\mathcal{T}$, $\rho_{\mathcal{T}}(P) \ge c\theta(P)^{-1}$ by Theorem 9.2, and $\phi(2) = 1$.

When $d = 3$, Theorem 15.1 shows that QMG has an aspect ratio bound of $c\theta(P)^{-2}$, whereas the lower bound from Theorem 9.2 is $c\theta(P)^{-1}$. In fact, a more complicated analysis of QMG for the $d = 3$ case establishes an upper bound of $c\theta(P)^{-1}$ on $\rho_{\mathrm{QMG}}(P)$. Here is a sketch of this analysis. Recall that the only case that needs attention is the case of a large protected box B_1 next to a much smaller protected box B_2, such that $\mathrm{co}(B_2) \subset \mathrm{co}(B_1)$. Suppose, for instance, that B_1 is protected in phase 0 (a similar argument applies to the case when B_1 is protected in phase 1), and suppose that B_2 is protected in phases 1 or 2. Find the largest ancestor B^* of B_2 with the property that $\mathrm{co}(B^*) \subset \mathrm{co}(B_1)$; as above, $\mathrm{size}(B_1)/\,\mathrm{size}(B^*) \le c$. Consider the chain of strongly crowded boxes from B^* down to B_2. Let P' be the intersection of P with the facet of B_1 separating it from B^*. Observe that splitting strongly crowded boxes in phases 1 and 2 of three-dimensional (3D) QMG applied to the chain of boxes B^* down to B_2 is very similar to phases 0 and 1 of 2D QMG running on the polygon P'. In other words, with a correct modification to the definition of $\mathrm{ex}(B)$ in two dimensions, whenever a 3D box on the boundary of B_1 is split for strong crowding, the corresponding 2D box would be split for strong crowding of P'.

Since 2D QMG is optimal with respect to maximizing the minimum altitude, we conclude that $\mathrm{size}(B_2)$ is bounded below by the minimum geodesic distance δ in P' between two faces that do not meet. But now it is easy to see that $\delta/\,\mathrm{size}(B_1)$ is bounded above by the sharpest angle θ at $\boldsymbol{v}$. Thus, $\mathrm{size}(B_1)/\,\mathrm{size}(B_2) \le c\theta(P)^{-1}$.

It is likely that this line of reasoning extends to higher dimensions, although we do not know the exact improvement to Theorem 15.1 possible with this analysis. Furthermore, we do not know whether our lower bound from Theorem 9.2 on the best attainable aspect ratio is tight in dimensions higher than 3.

We can summarize the conclusions of this section with the following theorem.

THEOREM 15.2. *Let $\mathcal{T}$ be an arbitrary triangulation of P with worst-case aspect ratio denoted $\rho_{\mathcal{T}}(P)$. Then*

$$\rho_{\mathrm{QMG}}(P) \le c_d \cdot (\rho_{\mathcal{T}}(P))^{\psi(d)},$$

where $\psi(d) = 1$ for $d = 2, 3$ and $\psi(d) \le \phi(d)$ for higher dimensions.

16. Bounded aspect ratio triangulations.

In the preceding section we showed that the QMG triangulation has an aspect ratio bound. In the next section we will show that, among all triangulations with bounded aspect ratio, QMG has the minimum cardinality, up to a constant factor. First, we present some preliminary lemmas that apply generally to any triangulation with bounded aspect ratio.

LEMMA 16.1. *Let $\mathcal{T}$ be a triangulation of a polyhedral region P whose aspect ratio is at most ρ. Let T_1, T_2 be two simplices that share a common vertex $\boldsymbol{v}$. Then* $\operatorname{minalt}(T_1) \le \zeta_1(\rho, d) \operatorname{minalt}(T_2)$, *where the function $\zeta_1(\cdot,\cdot)$ is defined below.*

We omit the proof of this lemma, which is contained in [10]. Here is a sketch. Two simplices S_1, S_2 of $\mathcal{T}$ that share an edge $\boldsymbol{v}_1\boldsymbol{v}_2$ satisfy $\operatorname{minalt}(S_1) \le \rho \operatorname{minalt}(S_2)$ by the chain of inequalities:

$$\begin{aligned} \operatorname{minalt}(S_1) &\le \|\boldsymbol{v}_1 - \boldsymbol{v}_2\| \\ &\le \operatorname{maxside}(S_2) \\ &= \operatorname{asp}(S_2) \operatorname{minalt}(S_2) \\ &\le \rho \operatorname{minalt}(S_2). \end{aligned} \tag{16.1}$$

Two simplices T_1, T_2 that share a vertex $\boldsymbol{v}$ are connected by a chain of simplices $S_1 (= T_1), S_2, \ldots, S_p (= T_2)$ that all share $\boldsymbol{v}$ and such that S_i and S_{i+1} have a common edge. This is because $\mathcal{T}$ is a triangulation of P, which is a manifold with boundary. It can be shown that the number of simplices p that can share a common vertex is bounded above in terms of ρ because the solid angle of each S_i at $\boldsymbol{v}$ is bounded below in terms of ρ. Thus, p is bounded above in terms of ρ: it turns out that $p \le c_d \rho^{d-1}$. Thus, the lemma is true with $\zeta_1(\rho, d) = \rho^{c_d \rho^{d-1}}$.

Now for the first result of the section. This lemma bounds the rate at which simplices can grow in a bounded aspect ratio triangulation.

LEMMA 16.2. *Let $\mathcal{T}$ be a triangulation of P whose aspect ratio bound is ρ. Let Π be a PL path in P from $\boldsymbol{x}$ to $\boldsymbol{y}$. Suppose that $\boldsymbol{x}$ is contained in a simplex T. Then every simplex T' containing $\boldsymbol{y}$ satisfies*

$$\operatorname{minalt}(T') \le c_d \zeta_1(\rho, d) \max(\operatorname{minalt}(T), \operatorname{lth}(\Pi)). \tag{16.2}$$

Proof. Let $F_1, \ldots, F_s$ be an enumeration of the faces of $\mathcal{T}$ met by Π. Let F_s be the lowest-dimensional face of $\mathcal{T}$ containing $\boldsymbol{y}$.

Case 1. *F_s has a vertex in common with T.* Since every simplex containing $\boldsymbol{y}$ also contains F_s, then T' has a common vertex with T. In this case the lemma is true with the first term of the max in (16.2) by the preceding lemma.

Case 2. *F_s does not have a vertex in common with T.* In this case, define a PL continuous function $f : P \to \mathbf{R}$ that is 1 on vertices of F_s and 0 on all other vertices and is linearly interpolated by $\mathcal{T}$. Then, as in the proof of Theorem 9.4, there must be a simplex S that meets Π such that the gradient of f on S is at least $1/\operatorname{lth}(\Pi)$, and hence this simplex satisfies $\operatorname{minalt}(S) \le c_d \operatorname{lth}(\Pi)$. Notice that S and F_s must have a common vertex because if not, the gradient of f on S would be 0. Since S and

T' have a common vertex, we apply the preceding lemma to bound $\text{minalt}(T')$ by the second term of the max in (16.2), proving the lemma. $\square$

Here is our other main result about bounded aspect ratio triangulations.

LEMMA 16.3. *Let C be a cube in $\mathbf{R}^d$ of side length s. Let $T_1, \ldots, T_n$ be a set of n simplices with pairwise disjoint interiors, satisfying* $\text{minalt}(T_i) \geq \mu$ *and* $\text{asp}(T_i) \leq \rho$ *for each i. Suppose each T_i meets C. Then*

$$n \leq c_d \rho^d + c_d s^d / \mu^d. \tag{16.3}$$

Proof. For each T_i, identify a point $\boldsymbol{x}_i \in T_i \cap C$. Now contract each T_i about $\boldsymbol{x}_i$ until we obtain a new simplex T_i' such that $\text{minalt}(T_i') = \mu$. Since $T_i' \subset T_i$, the set $T_1', \ldots, T_n'$ still enjoys the property that interiors are pairwise disjoint. Since $\boldsymbol{x}_i \in T_i'$, each T_i' still meets C. Finally, contraction affects maxside and minalt by the same scale factor, so $\text{asp}(T_i') = \text{asp}(T_i)$.

We know that $\text{maxside}(T_i') = \text{asp}(T_i')\,\text{minalt}(T_i') \leq \rho\mu$. Therefore, for every point $\boldsymbol{y}$ in T_i', $\text{dist}(\boldsymbol{y}, \boldsymbol{x}_i) \leq \rho\mu$. Let $\boldsymbol{x}_0$ be the centroid of C. Then we have for each i that $\text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_0) \leq c_d s$. Combining these inequalities yields the bound that for every point $\boldsymbol{y} \in T_i'$ for each i, $\text{dist}(\boldsymbol{y}, \boldsymbol{x}_0) \leq \rho\mu + c_d s$. Thus, all of $T_1', \ldots, T_n'$ are contained in a ball B of radius $\rho\mu + c_d s$ around $\boldsymbol{x}_0$. The volume of this ball is at most $c_d(\rho^d\mu^d + s^d)$. Each simplex has volume at least $c_d\mu^d$ by equation (8.1). Since the simplices have disjoint interiors, their number is bounded above by $\text{vol}(B)/(c_d\mu^d)$, which proves the lemma. $\square$

17. A bound on the cardinality of QMG.

In this section we show that the cardinality of the triangulation produced by QMG is always within a constant factor of optimal among all bounded aspect ratio triangulations, where the constant depends on the aspect ratio. We start with the following lemma.

LEMMA 17.1. *Let $\boldsymbol{x}$ be an arbitrary point in P, and let T be the simplex generated by QMG that contains $\boldsymbol{x}$. (If there is more than one, the result holds for any choice of T.) Let $\mathcal{S}$ be some other triangulation of P with aspect ratio bound $\rho_{\mathcal{S}}$, and let S be the simplex in $\mathcal{S}$ that contains $\boldsymbol{x}$. (If there is more than one, then the result holds for any choice of S.) Then*

$$\text{minalt}(S) \leq c_d \zeta_1(\rho_{\mathcal{S}}, d) \cdot \theta(P)^{-(2d+2)\phi(d)} \cdot \text{minalt}(T). \tag{17.1}$$

Proof. Recall that each simplex generated by QMG is associated with a full-dimensional protected box, namely, the last box in its chain. Furthermore, each full-dimensional protected box is associated with a full-dimensional anchor box as in Theorem 11.1. Therefore, transitively, each simplex generated by QMG is associated with an anchor box.

Let T be such a simplex, and B_a its anchor box. Clearly

$$\text{minalt}(T) \leq c_d \,\text{size}(B_a), \tag{17.2}$$

since T lies in $\text{ex}(B_a)$. On the other hand, there is also an inequality in the other direction. The reason is as follows. Let B be the full-dimensional protected box containing T. Then, as argued in section 10, $\text{minalt}(T)$ is bounded below by the size of the smallest neighbor of B, which, by the proof of Theorem 15.1, is bounded below by $c_d\theta(P)^{\phi(d)}\,\text{size}(B)$. There is a lower bound on $\text{size}(B)$ in terms of $\text{size}(B_a)$ given by (11.1). Combining these bounds yields

$$\text{minalt}(T) \geq c_d \theta(P)^{2\phi(d)}\,\text{size}(B_a). \tag{17.3}$$

Let $\boldsymbol{x}$ be the arbitrary point in P specified by the lemma. Let T be the simplex generated by QMG that contains $\boldsymbol{x}$, let B be the protected box associated with T, and let B_a be the anchor of B. In the next few paragraphs we will construct a PL path Σ from $\boldsymbol{x}$ to a simplex $\bar{S} \in \mathcal{S}$ satisfying $\text{minalt}(\bar{S}) \leq \text{size}(B_a)$ and such that $\text{lth}(\Sigma)$ is bounded above in terms of $\text{size}(B_a)$.

Observe that $\text{co}(B_a)$ contains an incontractible path Π_1 by definition of "anchor." Therefore, let S_1 be the simplex in $\mathcal{S}$ that meets Π_1 and satisfies $\text{minalt}(S_1) \leq c_d \, \text{size}(B_a)$, as specified in Theorem 9.4. Let $\boldsymbol{y}$ be a point in $\Pi_1 \cap S_1$. Construct the shortest PL path (the geodesic path) from $\boldsymbol{x}$ to $\boldsymbol{y}$ lying in $\text{co}(B_a)$, and call it Σ_1. Note that there is no a priori upper bound on $\text{lth}(\Sigma_1)$ in terms of $\text{size}(B_a)$ because $\text{co}(B_a)$ could contain geodesic paths possibly much longer than $\text{size}(B_a)$.

Let $T_1, T_2, \ldots, T_p$ be an enumeration of the QMG simplices met by Σ_1, listed in the order they are encountered starting with $\boldsymbol{x}$. Note that no simplex can appear twice in this enumeration; this is because Σ_1 is a geodesic path and therefore would not return to the same simplex more than once.

Let T_q be the first simplex in the sequence that fails to satisfy (17.3). If there is no such q, then take $q = p + 1$. Thus, $T_1, \ldots, T_{q-1}$ all satisfy (17.3). We claim that $q - 1 \leq c_d \theta(P)^{-2d\phi(d)}$. This follows from (16.3). Observe that $T_1, \ldots, T_{q-1}$ is a set of simplices with disjoint interiors all meeting $\text{ex}(B_a)$. We use (17.3) as a lower bound on the minimum altitudes of $T_1, \ldots, T_{q-1}$, $c_d \, \text{size}(B_a)$ as the size of $\text{ex}(B_a)$, and Theorem 15.1 to get upper bounds on aspect ratios. In this use of (16.3), the second term dominates the first on the right-hand side.

Suppose that $q = p+1$; in other words, every simplex in the enumeration satisfies (17.3). Then define $\Sigma = \Sigma_1$; we claim that

$$\text{lth}(\Sigma) \leq c_d \theta(P)^{-2d\phi(d)} \, \text{size}(B_a). \tag{17.4}$$

This is because Σ passes through $q-1$ simplices, and the length of its segment in each simplex is at most $c_d \, \text{size}(B_a)$. This choice of Σ has all the properties named above: it connects $\boldsymbol{x}$ to a point on a simplex S_1 (which satisfies $\text{minalt}(S_1) \leq c_d \, \text{size}(B_a)$) and satisfies (17.4). This concludes the construction of $\bar{S}$ for the case $q = p + 1$; in particular, we take $\bar{S} = S_1$.

The other case is that $q < p-1$. In this case, we truncate Σ_1 at the point where it enters T_q, which we denote $\boldsymbol{x}_1$; call the truncated path Σ'_1. Clearly Σ'_1 satisfies (17.4) by the same argument as in the last paragraph. Now notice that the anchor box for T_q cannot be B_a because T_q does not satisfy (17.3) by choice of q.

Therefore, identify the anchor of T_q, which we will call B_b, and start the construction anew from $\boldsymbol{x}_1$. In other words, find the incontractible path Π_2 in $\text{co}(B_b)$, find the simplex S_2 of $\mathcal{S}$ that meets the incontractible path and has altitude at most $c_d \, \text{size}(B_b)$, and let Σ_2 be the path $\boldsymbol{x}_1$ to a point in $S_2 \cap \Pi_2$. Note that $\text{size}(B_b) \leq \text{size}(B_a)/2$, because B_b must be smaller than B_a so that (17.3) can be satisfied for the new anchor. Find the first simplex in this new path that fails to satisfy (17.3) for B_b and so on. Notice that Σ'_2, the truncation of Σ_2, satisfies (17.4) with $\text{size}(B_b)$ taking the place of $\text{size}(B_a)$ on the right-hand side. Therefore, the upper bounds given by the right-hand side of (17.4) on $\text{lth}(\Sigma'_1), \text{lth}(\Sigma'_2), \ldots$ form a series decreasing by a factor of 2 each time. Eventually the procedure terminates at Σ_l because there is a finite lower bound on the smallest protected box in QMG. When the procedure terminates, concatenate $\Sigma'_1, \Sigma'_2, \ldots, \Sigma'_{l-1}, \Sigma_l$ into a PL path Σ. (This concatenation is possible because Σ'_1 ends at $\boldsymbol{x}_1$, which is where Σ'_2 begins, and so on.)

This path Σ has the following properties. It satisfies (17.4) with the original B_a on the right-hand side, multiplied by an additional factor of 2 that arises from summing a decreasing geometric series. It connects $\boldsymbol{x}$, the given point in P contained in a simplex T of QMG anchored at B_a, to a point $\boldsymbol{y}$ that is in a simplex S_l in triangulation $\mathcal{S}$ and that satisfies $\operatorname{minalt}(S_l) \le c_d \operatorname{size}(B_a)$. In this case, we take $\bar{S} = S_l$ to satisfy the conditions mentioned above.

This is exactly the setup we need to apply Lemma 16.2. Let S be the simplex in $\mathcal{S}$ that contains $\boldsymbol{x}$. From Lemma 16.2 applied to Σ we conclude that

$$\begin{aligned}\operatorname{minalt}(S) &\le c_d\zeta_1(\rho_{\mathcal{S}}, d) \cdot \max(\operatorname{lth}(\Sigma), \operatorname{minalt}(\bar{S}))\\ &\le c_d\zeta_1(\rho_{\mathcal{S}}, d) \cdot \theta(P)^{-2d\phi(d)} \operatorname{size}(B_a).\end{aligned} \tag{17.5}$$

The second line was obtained by substituting the bound (17.4) for $\operatorname{lth}(\Sigma)$ and then noting that this bound dominates the upper bound of $c_d \operatorname{size}(B_a)$ that applies to $\operatorname{minalt}(\bar{S})$.

Now the lemma is proved because we combine (17.5) with the bound on $\operatorname{size}(B_a)$ in terms of $\operatorname{minalt}(T)$ given by (17.3). □

THEOREM 17.2. *Let $n_{\mathrm{QMG}}(P)$ be the number of simplices in the triangulation produced by QMG. Let $\mathcal{S}$ be some other triangulation of P with aspect ratio bound $\rho_{\mathcal{S}}$, and let the cardinality of $\mathcal{S}$ be $n_{\mathcal{S}}$. Then*

$$n_{\mathrm{QMG}}(P) \le c_d\zeta_1(\rho_{\mathcal{S}}, d)^d \rho_{\mathcal{S}}^d \cdot \theta(P)^{-(2d+2)d\phi(d)} \cdot n_{\mathcal{S}}. \tag{17.6}$$

Proof. Let $f_{\mathrm{QMG}} : P \to \mathbf{R}$ be the piecewise constant function defined as follows. Let T be a simplex generated by QMG. The value of f_{QMG} on T is defined to be $1/\operatorname{vol}(T)$. On boundaries of simplices, a measure-zero set, we leave f_{QMG} undefined. Function $f_{\mathcal{S}} : P \to \mathbf{R}$ is defined similarly in terms of $\mathcal{S}$. Note that

$$n_{\mathrm{QMG}} = \int_{\boldsymbol{x}\in P} f_{\mathrm{QMG}}(\boldsymbol{x})\, d\boldsymbol{x}$$

because the value of the integral over each individual simplex is exactly 1. A similar expression holds for $n_{\mathcal{S}}$.

Define piecewise constant functions $g_{\mathrm{QMG}} : P \to \mathbf{R}$ to be $1/\operatorname{minalt}(T)^d$ on T, where T is a simplex in QMG, $g_{\mathcal{S}}$ similarly for $\mathcal{S}$. Finally, define $h_{\mathcal{S}}$ to the piecewise constant function that is $1/\operatorname{maxside}(S)^d$ on S, as S ranges over simplices in $\mathcal{S}$.

Then we have the following chain of inequalities.

$$\begin{aligned}n_{\mathrm{QMG}} &= \int_{\boldsymbol{x}\in P} f_{\mathrm{QMG}}(\boldsymbol{x})\, d\boldsymbol{x}\\ &\le c_d \int_{\boldsymbol{x}\in P} g_{\mathrm{QMG}}(\boldsymbol{x})\, d\boldsymbol{x}\\ &\le c_d\zeta_1(\rho_{\mathcal{S}}, d)^d \cdot \theta(P)^{-(2d+2)d\phi(d)} \cdot \int_{\boldsymbol{x}\in P} g_{\mathcal{S}}(\boldsymbol{x})\, d\boldsymbol{x}\\ &\le c_d\zeta_1(\rho_{\mathcal{S}}, d)^d \rho_{\mathcal{S}}^d \cdot \theta(P)^{-(2d+2)d\phi(d)} \cdot \int_{\boldsymbol{x}\in P} h_{\mathcal{S}}(\boldsymbol{x})\, d\boldsymbol{x}\\ &\le c_d\zeta_1(\rho_{\mathcal{S}}, d)^d \rho_{\mathcal{S}}^d \cdot \theta(P)^{-(2d+2)d\phi(d)} \cdot \int_{\boldsymbol{x}\in P} f_{\mathcal{S}}(\boldsymbol{x})\, d\boldsymbol{x}\\ &= c_d\zeta_1(\rho_{\mathcal{S}}, d)^d \rho_{\mathcal{S}}^d \cdot \theta(P)^{-(2d+2)d\phi(d)} \cdot n_{\mathcal{S}}.\end{aligned}$$

In these inequalities, we used (8.1) to obtain the second line, (17.1) for the third line, the aspect ratio bound for $\mathcal{S}$ for the fourth line, and (8.1) again for the fifth line. This proves the theorem. □

Note that this theorem allows the ratio $n_{\text{QMG}}/n_{\mathcal{S}}$ to be arbitrarily large if the competing triangulation $\mathcal{S}$ has bad aspect ratio. This is not merely an artifact of our analysis but is a feature of bounded aspect ratio triangulations, as illustrated by the following example. Consider a $p \times 1$ rectangle with $p \gg 1$. On such a domain, QMG would require $\Omega(p)$ triangles (as would any algorithm guaranteeing bounded aspect ratio), but this domain can be triangulated with just two triangles by inserting a diagonal. This latter triangulation has aspect ratio of $\Omega(p)$.

Note also that $\rho_{\mathcal{S}} \geq c_d\theta^{-1}$ by Theorem 9.2. Thus, the entire right-hand side of (17.6) can be bounded above with the more compact formula $f(\rho_{\mathcal{S}}, d) \cdot n_{\mathcal{S}}$.

18. Running time analysis. In this section we briefly discuss the running time of QMG. The running time for the separation stages is proportional to the number of boxes created multiplied by the time per box. There is no prior upper bound on the number of boxes in terms of the input. There is also no prior upper bound on the number of boxes in terms of the output, that is, in terms of the number of simplices produced, which we denote s. However, a modification to QMG would allow us to claim that the total number of boxes is bounded above by a multiple of s. The modification would be an additional operation to short circuit a series of splitting operations that make no progress. More specifically, the modification is as follows. When we split a box, we check whether only one of its children has nonempty content. If so, we discard the other children, and we immediately shrink that box by a power-of-two factor, until it is sufficiently small that we can be guaranteed that the next split will produce more than one child with nonempty content.

The amount of time to process a box depends on the combinatorial complexity of its content. A crude upper bound is that the complexity of the content is bounded by $O(N)$, where N is the complexity of the input domain P. Processing the content requires a connected component computation; the time for this computation in higher dimensions is $O(N^2)$, although, as mentioned in section 5, more efficient algorithms are available for two and three dimensions.

Thus, an estimate for the separation stage running time, using the modification mentioned above, is $O(N^2 s)$ operations. The operations in the alignment stage (checking complete coverage) can be done with a hash table as mentioned in section 7. Thus, alignment requires $O(s)$ operations. Finally, the triangulation part of the algorithm also requires $O(s)$ operations. Thus, the total running time is $O(N^2 s)$.

19. Implementation. A 2D version of QMG, called "tripoint," was implemented by S. Mitchell in C++ and is available on the World Wide Web [11]. A full version of QMG has been implemented in C++ by S. Vavasis; the implementation QMG1.1 is more general than the version described in this paper because it can also handle nonmanifold features, including several kinds of internal boundaries. QMG1.1 is available on the Web [16]. The implementation is slightly different from the algorithm described in this article; in particular, the alignment procedure uses an adaptive method for selecting tolerances and a different rule for choosing a close face. Computational experiments will be described elsewhere.

20. Open questions. Some of the open questions raised by this work include the following:

(1) Is there a triangulation algorithm with stronger optimality properties? For instance, the QMG aspect ratio is optimal (up to a constant factor) only in two and three dimensions.

(2) Several open questions were posed in section 14. For instance, is there a

characterization of the maximal value of the min-altitude in a triangulation of 3D polyhedra?

(3) Can this work be extended to curved boundaries? It appears that the main bottleneck is a solution to the subproblem of triangulating a uniform grid of boxes posed in the companion paper [12].

(4) Is there a mesh generation algorithm for three-dimensional domains that guarantees dihedral angles bounded by $\pi/2$? Such a bound is important for some finite element problems [17]. It is known [3] how to solve the corresponding problem in two dimensions.

(5) Can the running time bound be improved?

(6) The optimality properties demonstrated here all involve constant factors that are apparently very large. The QMG implementation in practice often produces meshes that are factors of 50 off from the minimum number of tetrahedra, and also factors of 50 off from the best aspect ratio. This leads to the following practical open question: Is there a mesh generation algorithm with theoretical guarantees comparable to QMG's, and yet with better theoretical constants or better performance in practice?

Acknowledgments. Preparation of this article has been an on-again, off-again process spanning a number of years. Several people have given us very useful comments on earlier versions, including Chandrajit Bajaj, Marshall Bern, Paul Chew, John Gilbert, Joe Mitchell, and the two referees of [13]. We thank Gail Pieper of Argonne and an anonymous referee for a careful reading of an earlier draft of this article.

REFERENCES

[1] B. S. Baker, E. Gross, and C. Rafferty, *Nonobtuse triangulation of polygons*, Discrete Comput. Geom., 3 (1988), pp. 147–168.

[2] M. Bern, D. Dobkin, and D. Eppstein, *Triangulating polygons without large angle*, Internat. J. Comput. Geom. Appl., 5 (1995), pp. 171–192.

[3] M. Bern and D. Eppstein, *Mesh generation and optimal triangulation*, in Computing in Euclidean Geometry, D. Z. Du and F. K. Hwang, eds., World Scientific, Singapore, 1995, pp. 47–123.

[4] M. Bern, D. Eppstein, and J. Gilbert, *Provably good mesh generation*, J. Comput. System Sci., 48 (1994), pp. 384–409.

[5] M. Bern and P. Plassmann, *Mesh Generation*, Tech. Report CSE-97-019, Department of Computer Science and Engineering, Penn State University, State College, PA, 1997.

[6] B. Chazelle and L. Palios, *Triangulating a nonconvex polytope*, in Proceedings of the Fifth ACM Symposium on Computational Geometry, ACM Press, New York, 1989, pp. 393–400.

[7] L. P. Chew, *Guaranteed-Quality Triangular Meshes*, Tech. report TR-89-983, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[8] L. P. Chew, *Guaranteed-quality mesh generation for curved surfaces*, in Proceedings of the Ninth ACM Symposium on Computational Geometry, ACM Press, New York, 1993, pp. 274–280.

[9] C. Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, Cambridge, 1987.

[10] G. Miller, S.-H. Teng, W. Thurston, and S. Vavasis, *Geometric separators for finite-element meshes.* SIAM J. Sci. Comput., 19 (1998), pp. 364–386.

[11] S. Mitchell, *Tripoint Information*, http://endo.sandia.gov/9225/Personnel/samitch/csstuff/csguide.html, 1996.

[12] S. A. Mitchell and S. Vavasis, *An aspect ratio bound for triangulating a d-grid cut by a hyperplane (extended abstract)*, in Proceedings of the 12th ACM Symposium on Computational Geometry, 1996, pp. 48–57. A full version is available online at ftp://ftp.cs.cornell.edu/pub/vavasis/papers/asp9.ps.Z.

[13] S. A. Mitchell and S. A. Vavasis, *Quality mesh generation in three dimensions*, in Proceedings of the ACM Computational Geometry Conference, 1992, pp. 212–221.

[14] J. Ruppert, *A new and simple algorithm for quality 2-dimensional mesh generation*, in Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 1993, pp. 83–92.

[15] E. Schönhardt, *Über die Zerlegung von Dreieckspolyedern in Tetraeder*, Math. Ann., 98 (1928), pp. 309–312.

[16] S. A. Vavasis, *QMG: Software for Finite-Element Mesh Generation*, http://www.cs.cornell.edu/home/vavasis/qmg-home.html, 1995.

[17] S. A. Vavasis, *Stable finite elements for problems with wild coefficients*, SIAM J. Numer. Anal., 33 (1996), pp. 890–916.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 4, pp. 1371–1400

CONTAINMENT AND OPTIMIZATION OF OBJECT-PRESERVING CONJUNCTIVE QUERIES*

EDWARD P. F. CHAN† AND RON VAN DER MEYDEN‡

Abstract. In the optimization of queries in an object-oriented database (OODB) system, a natural first step is to use the typing constraints imposed by the schema to transform a query into an equivalent one that logically accesses a minimal set of objects. We study a class of queries for OODBs called *conjunctive queries*. Variables in a conjunctive query range over heterogeneous sets of objects. Consequently, a conjunctive query is equivalent to a union of conjunctive queries of a special kind, called *terminal* conjunctive queries. Testing containment is a necessary step in solving the equivalence and minimization problems. We first characterize the containment and minimization conditions for the class of terminal conjunctive queries. We then characterize containment for the class of all conjunctive queries and derive an optimization algorithm for this class. The equivalent optimal query produced is expressed as a union of terminal conjunctive queries, which has the property that the number of variables as well as their search spaces are minimal among all unions of terminal conjunctive queries. Finally, we investigate the complexity of the containment problem. We show that it is complete in Π_2^p.

Key words. object-oriented database, query optimization, conjunctive queries, containment, equivalence, minimization, complexity

AMS subject classifications. 68Q15, 68Q25

PII. S0097539794262446

1. Introduction. The initial attempts at constructing object-oriented databases (OODBs) provided only navigational programming languages for manipulating data [21, 5]. The lack of query languages like those available in relational systems has been criticized as a major drawback of the object-oriented approach [30, 4]. Consequently, most, if not all, commercial OODBs now provide, or will provide, some form of high-level declarative query language (e.g., [23, 13, 22, 17, 18]). These query languages, like those of the relational model, transfer the burden of choosing an efficient execution plan for a query to the database system. This has led to a resurrection of the study of query optimization in the object-oriented setting (e.g., [26, 7, 6, 27, 15, 19, 10, 12]). Most of these papers develop transformations that reduce the cost of evaluating a given query but do not necessarily produce an *optimal* equivalent query.

In the setting of relational databases, a well-accepted notion of query optimality exists for the class of *conjunctive* queries [11], and the classical theory is based on the notion of query *containment*. A query Q_1 is said to be contained in a query Q_2 if, in every database instance, the set of answers to Q_1 is a subset of the set of answers to Q_2. In this paper, we study the containment and optimization problems for a class of conjunctive queries in an object-oriented setting. The closely related *equivalence* problem has previously been addressed for object-oriented queries by Hull and Yoshikawa [15]. Our results are complementary to their work, in that the language in [15] is object-generating, while our language is *object-preserving*. Our language enables a user to retrieve objects from a database, but not to create new complex

*Received by the editors January 14, 1994; accepted for publication (in revised form) March 13, 1999; published electronically March 6, 2000.
http://www.siam.org/journals/sicomp/29-4/26244.html

†Department of Computer Science, University of Waterloo, Waterloo, ON, N2L 3G1 Canada (epfchan@emap.uwaterloo.ca).

‡School of Computing Sciences, University of Technology, Sydney, Australia (ron@socs.uts.edu.au).

objects. Moreover, our language, like the one in [7], is defined on an inheritance hierarchy, whereas most languages studied in the literature are basically languages for complex objects without inheritance. The need to deal with inheritance introduces an extra level of complexity into the containment and optimization problems.

In an OODB, classes are named collections of similar objects. A class C may be refined into subclasses. Conversely, the class C is said to be a superclass of its subclasses. Subclasses are specializations of their superclasses. Consequently, objects in a class are also contained in its superclasses. Specialization of a class is often achieved by refining and/or adding properties to its superclasses. Since properties of a superclass are also properties of its subclasses, a subclass is said to inherit the properties of its superclasses. Class–subclass relationships form an acyclic directed graph called an inheritance or generalization hierarchy.

Inheritance is a powerful modeling tool because it allows for a better structured and more concise description of the schema and helps in factoring out shared implementations in applications [3]. Objects belonging to the same class share some common properties. Properties are attributes or methods defined on types; they are applicable only to instances of the types. In effect, therefore, types are constraints imposed on objects in the classes. Properties are formally denoted as attribute type pairs in this paper. A natural first step in query optimization is to use the typing constraints implied by the schema to minimize the search space for variables involved in the query. The following example illustrates how this idea may be applied to the kind of object-oriented conjunctive query we consider.

Example 1.1. Figure 1 is a schema for a vehicle rental database. It keeps track of all rental transactions for vehicles in the company. In this application, Auto, Trailer, and Truck are subclasses of the superclass Vehicle. There are clients, called discount customers, who are known to the company and receive special treatment. Discount customers receive a special rate and are not required to pay a deposit on the vehicles rented. However, discount customers are only allowed to rent automobiles and not other types of vehicles. Note that this constraint is captured by the more restrictive typing of the attribute VehRented in the subclass Discount of the class Client. Let us assume further that all superclasses are partitioned by their respective subclasses.

Suppose we want to find all those vehicles that have been rented to a discount client. Expressed in a calculus-like language, the query looks like the following:

$$Q_1 : \{x \mid \exists y\ (x \in \text{Vehicle}\ \&\ y \in \text{Discount}\ \&\ x \in y.\text{VehRented})\}.$$

Since discount clients are allowed to rent automobiles only, the above query is equivalent to the following query:

$$Q_2 : \{x \mid \exists y\ (x \in \text{Auto}\ \&\ y \in \text{Discount}\ \&\ x \in y.\text{VehRented})\}.$$

Q_2 is considered to be more optimal since the number of variables as well as their search spaces are minimal, given the typing constraints implied by the schema. Let us consider another query. Assume that we want to find those clients who rented a truck. It can be expressed as follows:

$$Q_3 : \{x \mid \exists y\ (x \in \text{Client}\ \&\ y \in \text{Truck}\ \&\ y \in x.\text{VehRented})\}.$$

Since discount clients are allowed to rent automobiles only (but not other kinds of vehicles); Q_3 is the same as the following query:

$$Q_4 : \{x \mid \exists y\ (x \in \text{Normal}\ \&\ y \in \text{Truck}\ \&\ y \in x.\text{VehRented})\}. \qquad \square$$

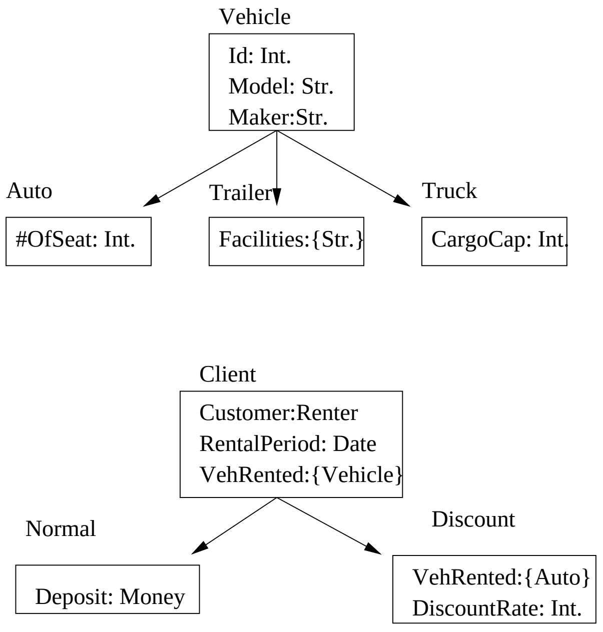


FIG. 1. *A vehicle rental database.*

Relational conjunctive queries have been studied extensively in the literature. A variable in a relational query ranges over a homogeneous relation. On the other hand, as illustrated by the above example, variables in an object-oriented query range over classes which could consist of heterogeneous sets of objects. This is because a class may be refined to various subclasses in which shared attribute names may correspond to different types or classes. For example, the variable x in Q_3 in Example 1.1 ranges over a heterogeneous set *Client* = *Normal* $\cup$ *Discount*. All the members of this set have the attribute *VehRented*, but only for members x of *Normal* can x.VehRented contain an element of the class *Truck*. This implies that clients who rent a truck are normal clients. This constitutes a significant divergence from the relational case. For instance, syntactically correct relational conjunctive queries are always satisfiable, but this is not true for object-oriented conjunctive queries [9]. The additional complexity is also reflected in the containment problem, as illustrated by the following example.

Example 1.2. The schema in Figure 2 records the employer–employee relationships among a group of people. The Employee attribute indicates the set of employees hired by a person.

Consider the following two queries defined on the above inheritance hierarchy. Q_1 retrieves all people x who hire a person u and a male v such that u is also an employee of v and u hires a female employee w. Q_2 finds all those people x who hire a male employee y who in turn hires a female employee z. Expressed in our language, they are as follows.

$$\begin{aligned} Q_1 : \{x \mid \exists u\, \exists v\, \exists w\ (&x \in \text{Person} \;\&\; u \in \text{Person} \;\&\; v \in \text{Male} \;\&\; w \in \text{Female} \\ &\& u \in x.\text{Employee} \;\&\; v \in x.\text{Employee} \;\&\; u \in v.\text{Employee} \\ &\&\; w \in u.\text{Employee})\}. \end{aligned}$$

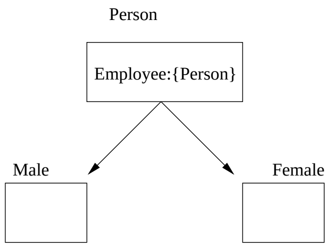


FIG. 2. *An employer-employee schema.*

$$Q_2 : \{x \mid \exists y\, \exists z\ (x \in \text{Person} \ \&\ y \in \text{Male} \ \&\ z \in \text{Female} \ \&\ y \in x.\text{Employee} \ \&\ z \in y.\text{Employee})\}.$$

The above two queries are best visualized as the two graphs in Figure 3.

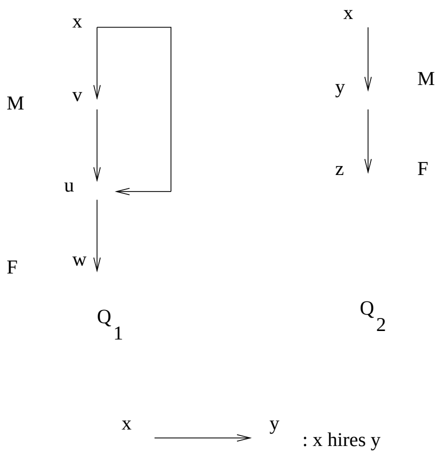


FIG. 3. *A query graph.*

We claim that Q_2 contains Q_1, meaning that whenever there is an answer for Q_1, it will also be an answer for Q_2. The person u is either a male or a female. If u is a male, then y and z in Q_2 can be mapped to u and w, respectively. Similarly if u is a female, variables y and z in Q_2 can be mapped to v and u, respectively. Thus, whenever there is an answer for Q_1, it will also be an answer for Q_2. $\square$

The above examples illustrate the kind of conjunctive queries we are interested in. Examples 1.1 and 1.2 demonstrate that the analysis of containment of conjunctive queries is more difficult than its counterpart in the relational case. This is due to the fact that the domains of attributes impose certain constraints on a query, and the analysis of the containment problem also involves analysis of disjunctive information.

The following is an overview of the problem and the approach we took in solving it. Given a conjunctive query $Q(\mathbf{S})$, where $\mathbf{S}$ is an object-oriented database schema denoted by an inheritance hierarchy, we want to find an equivalent query $Q'(\mathbf{S})$ that is, in some sense, optimal. Moreover, we are interested in determining when a conjunctive query is contained in another one. Both problems require an understanding of what a conjunctive query represents. We first observe that a conjunctive query, like the ones in Examples 1.1 and 1.2, can be decomposed as a union of a special kind of conjunctive queries called terminal conjunctive queries. As typing constraints in an inheritance hierarchy are restrictions on objects in a database, not every terminal conjunctive query is satisfiable. With typing constraints implied by an inheritance hierarchy, unsatisfiable terminal conjunctive queries can be determined and are eliminated from a union. Having removed unsatisfiable terminal conjunctive queries, characterization of containment and optimization are then derived. The technique employed and the result obtained are similar to those in select-project-join-union (SPJU)-expressions in the relational theory [24].

Most work on query optimization in OODBs concentrates on complex object optimization without considering the typing constraints imposed by the inheritance hierarchy (e.g., [26, 6, 27, 15, 19, 12]). Type checking of queries in the presence of *nonstrict* inheritance hierarchy was studied in [7]. Our work is different from all previous approaches in several important respects. First, we use the typing constraints imposed by an inheritance hierarchy to study the containment, equivalence, and optimization of queries. Second, our optimization is an exact minimization while most of the previous work deals with algebraic transformations and/or heuristics (e.g., [26, 27, 6, 19, 12]). Third, with reasons similar to those noted in [24], characterizing equivalence does not suffice to solve the optimization problem. Instead, we need to understand the containment problem as well. This work, to our best knowledge, is the first work that provides a characterization for containment of queries in an object-oriented setting. This result could also find applications in a view definition and classification in an OODB. For instance, to correctly integrate a virtual class or view into an inheritance hierarchy, it is imperative to resolve the containment problem for the view definition language [25]. Last, we demonstrate that the idea of containment mappings of relational conjunctive queries [11] can be extended to its object-oriented counterpart. Our proposed language, on the other hand, is perhaps more restrictive than some other query languages studied in the literature.

The next section defines the class of conjunctive queries and the basic notation needed throughout the discussion. In characterizing the containment and equivalence of terminal conjunctive queries, it is assumed that the query involved is satisfiable. We present an efficient algorithm for solving the satisfiability problem for terminal conjunctive queries in section 3. The results in that section were proven in [9] and are needed in the subsequent discussions. Sections 4 and 5 characterize the containment, equivalence, and minimization conditions for *terminal* conjunctive queries. In section 6, we solve the containment problem and derive an algorithm for optimizing the class of all conjunctive queries. The notion of optimization captures the intuition of minimization of the number of variables as well as their search spaces. In section 7, we analyse the complexity of testing containment of conjunctive queries. The main result shows that the problem is Π_2^p-complete. Finally, we give our conclusions in section 8.

2. Definitions and notation.

In this section, we introduce notation that is necessary for the rest of the discussion.

2.1. Types, classes, and schemas.

We suppose, given the following pairwise disjoint sets:

1. A set $\mathcal{T}$ of *atomic type domains*, where each atomic type domain is an *infinite* set of *atomic values*. Examples of atomic type domains are the set of integers

and the set of strings over some alphabet. We asssume distinct atomic type domains are disjoint.
2. A countably infinite set $\mathcal{O}$ of symbols which are called *object identifiers*.
3. A set $\mathcal{B}$ of *atomic types*, containing for each atomic type domain $T \in \mathcal{T}$, a symbol $\mathbf{T}$ naming that type domain. For brevity, we abuse notation by using the same symbol T for a type domain and its name. Thus $\mathcal{B} = \mathcal{T}$.
4. A countably infinite set $\mathcal{A}$ of symbols, the *attributes*.
5. A countably infinite set $\mathcal{C}$ of symbols which are called *classes*.

The set $\bigcup_{T \in \mathcal{T}} T$ is said to be the set of *atomic values*. The elements of $\mathcal{A}$ will be used as attribute names in tuple types, and the elements of $\mathcal{C}$ serve as names for user-defined classes.

A *type expression* over a set $\mathbf{C} \subseteq \mathcal{C}$ of class names is an expression defined as follows:

1. Every element of $\mathcal{B}$ is a type expression, called an *atomic type*.
2. Every element of $\mathbf{C}$ is a type expression, called a *class*.
3. If t is an atomic type or a class, then $\{t\}$ is type expression, called a *set* type.
4. If $a_1, \dots, a_n$ are distinct attributes in $\mathcal{A}$ and $t_1, \dots, t_n$ are atomic types, set types or classes, where $n \geq 0$, then $[a_1 : t_1, \dots, a_n : t_n]$ is a type expression, called a *tuple* type. As in a relation scheme, the order of attributes is immaterial. The empty tuple [] is also a tuple type. The type t_i is said to be the *type* of the attribute a_i, for each $i = 1, \dots, n$.

We write *type-expr*$(\mathbf{C})$ for the set of all type expressions over $\mathbf{C}$.

Following [20, 8], we introduce the notion of schema. A *schema* $\mathbf{S}$ is a triple $(\mathbf{C}, \sigma, \prec)$, where $\mathbf{C}$ is a finite subset of $\mathcal{C}$, σ is a function from $\mathbf{C}$ to tuple types, and $\prec$ is a partial order on $\mathbf{C}$. The mapping σ associates to each class in $\mathbf{C}$ a tuple type in *type-expr*$(\mathbf{C})$, which describes its structure. As noted in [14], there is no loss of representational power in restricting the structures of classes to be tuple types. The relationship $\prec$ among classes represents the user-defined *inheritance hierarchy*. We assume that the hierarchy has no cycle of length greater than 1. A class $A \in \mathbf{C}$ is said to be *terminal* if there is no class $B \neq A$ such that $B \prec A$. Otherwise, A is *nonterminal*. A class B is a *descendant* (or an *ancestor*) of a class A if $B \prec A$ (or $A \prec B$, respectively).

Following [1, 20], we derive from this hierarchy a subtyping relation $\leq$ among expressions in *type-expr*$(\mathbf{C})$. Let $\mathbf{S} = (\mathbf{C}, \sigma, \prec)$ be a schema. The *subtyping* relation among expressions in *type-expr*$(\mathbf{C})$ is the smallest ordering $\leq$, which satisfies the following axioms:

1. $A \leq A$ if $A \in \mathcal{B}$.
2. $B \leq C$ if $B \prec C$, for all classes B and C in $\mathbf{C}$.
3. $\{t\} \leq \{s\}$, for all types s, t such that $t \leq s$.
4. $[a_1 : t_1, \dots, a_n : t_n, \dots, a_{n+p} : t_{n+p}] \leq [a_1 : s_1, \dots, a_n : s_n]$, for all atomic types, set types or classes $t_1, \dots, t_n, s_1, \dots, s_n$ such that $t_i \leq s_i$, for all $i = 1, \dots, n$.

The order of arguments in a tuple type is immaterial, so we have $[a_3 : C_3, a_2 : C_2, a_1 : C_1] \leq [a_1 : C_1, a_2 : C_2]$.

For any expressions E_1 and E_2 in *type-expr*$(\mathbf{C})$, E_1 is a *subtype* of E_2 if $E_1 \leq E_2$. It is worth noting that the subtyping relation is a reflexive and transitive relation. As inheritance hierarchies are given by users, some schemas may not be meaningful. Let $\mathbf{S} = (\mathbf{C}, \sigma, \prec)$ be a schema. $\mathbf{S}$ is *consistent* if, for all classes B and C such that $B \prec C$, we have $\sigma(B) \leq \sigma(C)$. We consider only consistent schemas throughout this paper. The schemas we have defined are essentially the same as those defined in ODMG-93 [8]. Thus, our results are applicable to systems conforming to this standard. Let $C \in \mathbf{C}$. Attributes in $\sigma(C)$ are called the *attributes* of C. The *type of* $C.A$, denoted *type*$(C.A)$, is the type t of A in $\sigma(C)$. In consistent schemas, subclasses are specializations of their superclasses. Specialization of subclasses is represented formally by refining types of inherited attributes and/or adding new attribute type pairs.

2.2. States, domains, and objects. Let $\mathbf{S} = (\mathbf{C}, \sigma, \prec)$ be a schema and $\leq$ be the subtyping relation on *type-expr*$(\mathbf{C})$. Let $\mathbf{O}$ be a finite subset of $\mathcal{O}$ and $\mathbf{I}_c$ be a function from $\mathbf{O}$ to $\mathbf{C}$. Given $\mathbf{O}$ and $\mathbf{I}_c$, each type expression T in *type-expr*$(\mathbf{C})$ is interpreted as a set of possible values, called the *domain* of T, denoted as *dom*(T). In order to represent inapplicable attributes, we introduce a new symbol "Λ." The *domain* of a type with respect to $\mathbf{O}$ and $\mathbf{I}_c$ is defined as follows:

1. If $\mathbf{T} \in \mathcal{B}$ is an atomic type naming the type domain T, then $dom(\mathbf{T}) = T$.
2. For each class $D \in \mathbf{C}$, we define $dom(D) = \{o \mid o \in \mathbf{O}$ and $\mathbf{I}_c(o) = E$, where $E \prec D\}$.
3. For each set type $\{t\}$, we define $dom(\{t\}) = \{v \mid v \subseteq dom(t)\}$.
4. For each tuple type $[a_1 : t_1, \ldots, a_n : t_n]$, we define $dom([a_1 : t_1, \ldots, a_n : t_n]) = \{[a_1 : v_1, \ldots, a_n : v_n] \mid v_i \in dom(t_i) \cup \{\Lambda\}$ for all $i = 1, \ldots, n\}$.

Note that the value of an attribute of a tuple may be the null value Λ. This is to be interpreted as the attribute being inapplicable.

A *state* $\mathbf{s}$ on a schema $\mathbf{S} = (\mathbf{C}, \sigma, \prec)$ is a triple $(\mathbf{O}, \mathbf{I}_c, \mathbf{I}_v)$, where $\mathbf{O}$ is a finite subset of $\mathcal{O}$, $\mathbf{I}_c$, which is a function from $\mathbf{O}$ to $\mathbf{C}$, and $\mathbf{I}_v$ is a function from $\mathbf{O}$ to tuple values in domains of types with respect to $\mathbf{O}$ and $\mathbf{I}_c$. The function $\mathbf{I}_v$ maps each element in $\mathbf{O}$ to a tuple value which satisfies the following:

$$\forall o \in \mathbf{O},\ \mathbf{I}_v(o) \in dom\left(\sigma\big(\mathbf{I}_c(o)\big)\right).$$

That is, $\mathbf{I}_v$ defines the data value of an object and the (tuple) value of an object defined on a class must satisfy the type specification associated with the class. The set $\{\langle o, \mathbf{I}_v(o)\rangle \mid o \in \mathbf{O}\}$ is the set of objects in the state $\mathbf{s}$. Two objects in a state are *identical* if and only if they have the same identifier, so we may sometimes abuse terminology by referring to the identifier o as an *object* of $\mathbf{s}$.

Let $[a_1 : v_1, \ldots, a_n : v_n]$ be a tuple value. Then $[a_1 : v_1, \ldots, a_n : v_n].a_i$ is v_i. We call v_i the value of attribute a_i in the tuple. If the attribute a does not occur in a tuple, then the value of attribute a in the tuple is Λ.

In many, if not most, existing object-oriented database systems (e.g., [22, 17, 23]), an object is defined on exactly one most specialized class in an inheritance hierarchy. Consequently, as in [1, 17], we assume the following throughout the discussion.

2.2.1. Terminal class partitioning assumption. Given any state $\mathbf{s} = (\mathbf{O}, \mathbf{I}_c, \mathbf{I}_v)$ on a schema $\mathbf{S}$, the class $\mathbf{I}_c(o)$ is a terminal class in $\mathbf{S}$ for every object o in $\mathbf{O}$. That is, every nonterminal class is partitioned by its terminal descendants.

2.3. A class of object-preserving conjunctive queries. In this subsection, we define a calculus-like query language for an object-oriented database.

Queries are constructed from a set of variables, symbols from the set of atomic values, the equality operator "$=$," the membership operator "$\in$," the OR operator "$\sqcup$," the logical operator "&," as well as the existential quantifier "$\exists$." The set of variables is assumed to be disjoint from other sets of symbols.

First we define the concept of term. Terms enable us to refer to an object or a component of an object. A *term* is an expression of one of the following forms: c or x or $x.A$, where c is an atomic value, i.e., c is in some atomic type domain, x is a variable, and A is an attribute. A term of the form x or $x.A$ is called a *variable term.* An *attribute term* is of the form $x.A$.

An *atom* or an *atomic formula* is defined to be one of the following:

1. $x \in C_1 \sqcup \cdots \sqcup C_n$, where the C_i's are classes or atomic types, and x is a variable. An atom $x \in C_1 \sqcup \cdots \sqcup C_n$ is called a *range* atom and it asserts that the variable x denotes an object in the class C_i or a value in the atomic type C_i, for some $1 \leq i \leq n$.
2. $t_1 = t_2$, where t_1 and t_2 are terms. Such an atom is called an *equality* atom. An equality atom asserts that the operands denote identical objects or are of the same atomic value.
3. $x \in y.A$, where x and y are variables. The atom $x \in y.A$ is called a *membership* atom. A membership atom asserts that the object or atomic value denoted by x is a member of the *set object* denoted by $y.A$.

It is worth noting that path expressions of the form $x.A_1 \dots A_n$ (as used in [33]) and of the form $x.A_1[y_1] \dots A_n[y_n]$ (as used in [16]), where x and y_i's are variables or atomic values, can all be represented indirectly in our language. Likewise, atoms of the forms $c \in x.A$ and $y.A \in C_1 \sqcup \cdots \sqcup C_n$, and of the form $x.A \in y.B$, where x and y are variables and c is an atomic value, can again be expressed indirectly in our language.

A *formula* is constructed from atomic formulas, the logical operator "&," as well as existential quantifiers. *Bound* and *free* variables are defined in the usual manner. A *query* is an expression of the form $\{\ t \mid \Phi(t)\}$, where t is either a variable or an atomic value and $\Phi(t)$ is a formula. The term t is called the *distinguished term* of the query. A query $\{s_0 \mid \Phi(t)\}$ is called *conjunctive* if $\Phi(t)$ is of the form $\exists s_1 \cdots \exists s_m(M)$, where M is a formula containing no quantifier that is a conjunction of atomic formulas.[1] $\exists s_1 \cdots \exists s_m$ is called the *prefix* and M is called the *matrix* of the formula or of the query. We also make use of *union queries*, which are expressions of the form $Q_1 \cup \cdots \cup Q_n$, where each Q_i is a conjunctive query.

2.4. Semantics of queries. We now give the semantics of queries and define the notion of query containment. It is convenient, for technical reasons that will become apparent below, to state the semantics in terms of a mapping to a language that uses the objects and atomic values of a state as basic syntactic entities. We remark that the semantics are slightly nonstandard, in that they require that the terms occurring in an atom have a nonnull value in order for the atom to be true: this was handled in [9] using a three-valued logic, but since only the truth of atoms is relevant to the containment question for conjunctive queries, we simplify this here.

Define a *term over a state* $\mathbf{s} = (\mathbf{O}, \mathbf{I}_c, \mathbf{I}_v)$ to be an expression of one of the following forms: an atomic value c, an object $o \in \mathbf{O}$, or an expression $o.A$, where $o \in \mathbf{O}$ is an object and A is an attribute. The *value* $Val(t)$ of a term t over $\mathbf{s}$ is defined as follows:

1. If t is an atomic value c, then $Val(t) = c$.
2. If t is an object $o \in \mathbf{O}$, then $Val(t) = o$.
3. If t is the expression $o.A$, where $o \in \mathbf{O}$ is an object and A is an attribute, then $Val(t)$ is the value of attribute A in $\mathbf{I}_v(o)$.

Note that $Val(t)$ may be an atomic value, object, set, or the null value Λ.

[1]The results in this paper can be extended to conjunctive queries with more than one free variable.

An *atom over* $\mathbf{s}$ is an expression of one of the following forms:

1. $t \in C_1 \sqcup \cdots \sqcup C_n$, where t is a term over $\mathbf{s}$ and the C_i are classes or atomic types;
2. $t_1 = t_2$, where t_1 and t_2 are terms over $\mathbf{s}$, or
3. $t \in o.A$, where t is a term over $\mathbf{s}$, where o is an object of $\mathbf{s}$, and where A is an attribute.

We define certain atoms $\mathbf{A}$ over a state $\mathbf{s}$, to be *satisfied* in $\mathbf{s}$, written $\mathbf{s} \models \mathbf{A}$, as follows:

1. $\mathbf{s} \models t \in C_1 \sqcup \cdots \sqcup C_n$, where t is a term over $\mathbf{s}$ and each C_i is a class or atomic type, if $\mathit{Val}(t) \in \mathit{dom}(C_i)$ for some $i = 1, \dots, n$;
2. $\mathbf{s} \models t_1 = t_2$, where t_1 and t_2 are terms over $\mathbf{s}$, if $\mathit{Val}(t_1) = \mathit{Val}(t_2)$ and neither value is equal to Λ;
3. $\mathbf{s} \models t \in o.A$, where t is a term over $\mathbf{s}$, the o is an object of $\mathbf{s}$, and A is an attribute, if $\mathit{Val}(t)$ is not equal to Λ and $\mathit{Val}(o.A)$ is a set that contains $\mathit{Val}(t)$.

Note that in order for an atom to be satisfied, all of its terms must have nonnull values.

An *assignment* for a query Q in a state $\mathbf{s} = (\mathbf{O}, \mathbf{I}_c, \mathbf{I}_v)$ is a function α mapping each variable of Q either to an atomic value or to an object in $\mathbf{O}$. Assignments may be extended to mappings from the terms and atoms of Q to terms and atoms over $\mathbf{s}$, respectively, as follows:

1. for terms which are atomic values c we define $\alpha(c) = c$,
2. for terms of the form $x.A$, where x is a variable, we define $\alpha(x.A)$ to be the expression $o.A$, where $o = \alpha(x)$, and
3. for atoms $\mathbf{A}$ of Q we define $\alpha(\mathbf{A})$ to be the atom over $\mathbf{s}$ obtained by substituting for each term t in $\mathbf{A}$ the term $\alpha(t)$.

Using the notion of satisfaction of atoms over a state in that state, we now define a formula Φ to be satisfied in a state $\mathbf{s}$ with respect to an assignment α, written $\mathbf{s}, \alpha \models \Phi$, in the usual way. For atomic formulae $\mathbf{A}$ we have $\mathbf{s}, \alpha \models \mathbf{A}$ if $\mathbf{s} \models \alpha(\mathbf{A})$. The cases of Boolean operators and quantifiers are as in the standard semantics of first order logic, where the universe consists of the union of the sets $\mathit{dom}(T)$, where T ranges over all type expressions.

A query $Q = \{t \mid \Phi(t)\}$ is said to be *satisfied in a state* $\mathbf{s}$ *with respect to an assignment* α, written $\mathbf{s}, \alpha \models Q$, if $\mathbf{s}, \alpha \models \Phi(t)$. The assignment α is called a *satisfying* assignment for Q in this case. We say that the object or value a is an *answer* of Q with respect to $\mathbf{s}$ if there exists a satisfying assignment α for Q such that $a = \alpha(t)$, where t is the distinguished term of Q. If Q is a query and $\mathbf{s}$ is a state, we write $Q(\mathbf{s})$ for the set of all answers of Q with respect to $\mathbf{s}$. For union queries Q of the form $Q_1 \cup \cdots \cup Q_n$ we define $Q(\mathbf{s})$ to be the set $Q_1(\mathbf{s}) \cup \cdots \cup Q_n(\mathbf{s})$.

A query Q is said to be *satisfiable* if there is a state $\mathbf{s}$ such that $Q(\mathbf{s})$ is nonempty. Given two queries Q_1 and Q_2 (on a schema $\mathbf{S}$), Q_1 is said to *contain* Q_2 with respect to $\mathbf{S}$, denoted $Q_1 \supseteq Q_2$, if $Q_1(\mathbf{s}) \supseteq Q_2(\mathbf{s})$, for all states $\mathbf{s}$ on $\mathbf{S}$. Two queries Q_1 and Q_2 are said to be *equivalent* with respect to schema $\mathbf{S}$, denoted $Q_1 \equiv Q_2$, if they contain each other with respect to $\mathbf{S}$.

We note that for conjunctive queries, the existential quantifiers are not strictly essential: the query $\{t \mid \exists x_1 \cdots \exists x_n(\Phi)\}$ is equivalent to the query $\{t \mid \Phi\}$. Consequently, we assume henceforth, for purposes of analysis, that queries do not contain existential quatifiers. This yields the following simple characterization of satisfaction: α is a satisfying assignment for a conjunctive query Q in a state $\mathbf{s}$ if and only if

$\mathbf{s} \models \alpha(\mathbf{A})$ for all atoms $\mathbf{A}$ of Q. (It is still sometimes convenient to write formulae with quantifiers in order to scope variables and avoid naming conflicts.)

2.5. Well-formed conjunctive queries. We consider only those queries in which each term either denotes an object or a value, or a set of objects or values, but not both. We call such queries *well-formed.* The following defines when a query is well formed. First we note that, given a conjunctive query, additional equalities among terms could be inferred with the following algorithm. It is easy to see that the inferences performed in the algorithm are correct.

```
ALGORITHM EQUALITYGRAPH. Given a conjunctive query, generate additional
implied equality edges.
 Input: A conjunctive query $Q$.
 Output: An undirected graph $E(Q)$, called the complete equality relationship graph
         for $Q$.
 Method: The edges $\{x, y\}$ in the graph $E(Q)$ are called equality edges, and are also
         denoted by '$x = y$'.
   (1) Generate a graph with terms in $Q$ as nodes. (If $x.A$ is a term of $Q$, then
       so is $x$.) Generate additional nodes and equality edges by applying the
       following three steps exhaustively to the graph until no more edges can
       be derived.
       (i) For each node $t$, derive the equality edge $t = t$. For each equality
           atom "$s = t$" of $Q$, generate an equality edge between the node $s$
           and the node $t$.
       (ii) If $s = t$ and $t = u$ are equality edges, then derive the equality edge
            $s = u$.
       (iii) If $x$ and $y$ are variable nodes, $x = y$ is an equality edge, and $x.A$ is
             a node in the graph, then add the node $y.A$, if it does not already
             exist, and derive the equality edge $x.A = y.A$.
   (2) Output the graph constructed.
```

By substeps (i) and (ii), the complete equality relationship graph $E(Q)$ for a conjunctive query Q yields an equivalence relation R, defined by tRt', if there exists an equality edge between t and t'. For each term t in $E(Q)$, the equivalence class $[t]$ of R containing t is the set $\{t' \mid t'$ is a node in $E(Q)$ and there is an equality edge between t and $t'\}$. These sets are said to be the *equivalence classes* of $E(Q)$.

Let Q be a query. An occurrence of a term $x.A$ in the matrix of Q is a *set occurrence* if the occurrence appears on the right-hand side of a membership atom. All other occurrences of terms in the matrix of Q are *object occurrences.* A term s is an *object* term if some term t in the equivalence class $[s]$ has an object occurrence in the query. A term s is a *set* term if there is a set occurrence in the query of some term $t \in [s]$. Intuitively, a set term is one that *must* denote a set, and an object term is one that *must* denote an object or atomic value. A conjunctive query Q is *well formed* if

(i) every term in Q is either an object term or a set term, but *not* both,
(ii) each object term of the form $x.A$ is equated, directly or indirectly, to some variable or atomic value; that is, there is a variable or an atomic value in the equivalence class $[x.A]$, and

(iii) every variable in Q ranges over exactly one disjunction of classes or atomic types; that is, there is exactly one range atom associated with each variable.

Condition (i) is necessary for the satisfiability of the query and arises from the obvious constraint that no term can simultaneously denote both an object and a set. It is worth remarking that this condition implies that a set term cannot occur within an equality atom in the query, for such an occurrence would be an object occurrence, making the term simultaneously a set term and and object term. Note, moreover, that no object term can ever denote a set, for, by conditions (ii) and (iii), an object term must denote an element of some union of classes and atomic types.

For terms denoting objects or atomic values, condition (ii) is not a real restriction, since such a term can always be equated to some new existentially quantified variable ranging over all classes and atomic types. This condition is needed to simplify the discussion in the subsequent sections. In the case of condition (iii), note that because of the terminal class partitioning assumption, a query is unsatisfiable if it contains both $x \in C$ and $x \in D$, where C and D are distinct terminal classes. Such a query may be satisfiable if C and D are nonterminal classes, but in this case the two range atoms can be replaced (given a schema) with the single atom $x \in C_1 \sqcup \cdots \sqcup C_n$, where $C_1, \ldots, C_n$ are the common terminal descendants of C and D. Moreover, if the variable x occurs in no range atom, then we may clearly add the atom $x \in C_1 \sqcup \cdots \sqcup C_n$, where $C_1, \ldots, C_n$ are all terminal classes and atomic types, without changing the meaning of the query.

For the rest of this paper, we use the term *conjunctive queries* to denote well-formed conjunctive queries. Well-formed queries include *safe* as well as *unsafe* queries that produce infinite answers [31].

2.6. Terminal conjunctive queries. A *terminal* conjunctive query is a conjunctive query in which every range atom is of the form "$x \in C$," where C is a terminal class or an atomic type.

Every conjunctive query can be expressed as a union of terminal conjunctive queries, as follows. First, define an *expansion* of a query Q to be a query obtained by replacing each range atom of the form $x \in C_1 \sqcup \cdots \sqcup C_n$ by one of the atoms $x \in C_i$. For example, the query

$$\{x \mid (x \in C \sqcup D) \wedge (y \in E \sqcup F) \wedge (x.A = y)\}$$

has four expansions, one of which is the query

$$\{x \mid x \in D \wedge y \in E \wedge x.A = y\}.$$

Note that every expansion of a conjunctive query is a terminal conjunctive query.

PROPOSITION 2.1. *Let Q be a conjunctive query and let $Q_1, \ldots, Q_n$ be all the expansions of Q. Then Q is equivalent to $Q_1 \cup \cdots \cup Q_n$.*

Proof. See [9]. $\square$

This result states that, semantically, a conjunctive query corresponds to a union of terminal conjunctive queries. Each terminal conjunctive query in a union could have variables defined on different domains. To solve the containment and equivalence problems, it is necessary to solve the satisfiability problem and to identify exactly the set of objects or values over which a variable is ranging. In section 3, we present an algorithm for solving these problems for the terminal conjunctive queries. This algorithm employs typing constraints to determine satisfiability of a terminal conjunctive query. Queries that are not terminal could, by Proposition 2.1, first be decomposed into a union of terminal conjunctive queries. We can then apply the algorithm in section 3 to each subquery in the union to determine its satisfiability and delete the unsatisfiable subqueries from the union. In sections 4 and 5, we shall derive algorithms for testing containment and for minimizing terminal conjunctive queries. Section 6.1 deals with containment of unions of conjunctive queries, which can be used to determine containment of arbitrary conjunctive queries.

3. An efficient algorithm for testing satisfiability of terminal conjunctive queries.

Testing the satisfiability of restricted classes of conjunctive queries is an NP-complete problem [9]. However, determining if a terminal conjunctive query is satisfiable is tractable. We present in this section an algorithm that solves this problem in polynomial time, from [9], along with a sketch of its correctness proof. The aspect of this proof that is germane to our purposes in the present paper is that if the input query is satisfiable, it is possible to construct a "minimal" state with respect to which the query returns a nonempty result. Various properties of the state constructed are needed in the proof of the characterization of containment of terminal conjunctive queries.

One of the reasons for unsatisfiability of a query is the incompatibility of the typing of its terms implied by the schema. Note that a query Q and schema $\mathbf{S} = (\mathbf{C}, \sigma, \prec)$ together determine a type $\mathit{type}(t)$ for every term t of the query. (In every case, this type is in fact a class or atomic type.) For every atomic value c in an atomic type T, we define $\mathit{type}(c) = T$. For every variable x in Q, define $\mathit{type}(x)$ to be the unique class or atomic type C such that Q contains an atom of the form $x \in C$. For every term of the form $x.A$ in Q, define $\mathit{type}(x.A)$ as the type of attribute A in $\sigma(\mathit{type}(x))$, if $\mathit{type}(x)$ is a class and A is an attribute of $\sigma(\mathit{type}(x))$, and undefined otherwise.

In order for the query to be satisfiable, the type assigned to its terms must be consistent. By the terminal class partitioning assumption, terms denoting the same object must belong to the same terminal class or atomic type, and an object belonging to a set must be of a type admissible for that set. The following definition helps to check these conditions. If T is an atomic type or a class, we say D is a *terminal subtype* of T if either T is an atomic type and D is T, or D is a terminal descendant of T. Let t be an object term in Q. Define $\mathit{SatType}(t)$ to be the set of all D such that

1. for all terms $u \in [t], D$ is a terminal subtype of $\mathit{type}(u)$, and
2. for every '$u \in z.A$' in Q, where $u \in [t], D$ is a terminal subtype of C, where $\mathit{type}(z.A)$ is $\{C\}$.

Intuitively, $\mathit{SatType}(t)$ is the set of terminal types that are consistent with all the typing information on t derivable from the query. Since every object term must be equated to some variable, which must range over a terminal type, or to an atomic value, $\mathit{SatType}(t)$ contains at most a single element. Note that the definition depends only on the equivalence class of t, so we may also write $\mathit{SatType}([t])$ for $\mathit{SatType}(t)$. Note also that if Q is satisfiable, then $\mathit{SatType}(t)$ is a singleton set for every variable object term t in Q. For if $\mathit{SatType}(t)$ were empty, then it would be impossible to construct a satisfying assignment.

```
ALGORITHM SATTESTUT. Verify if a terminal conjunctive query $Q$ is satisfiable.
Input: A terminal conjunctive query $Q$ on $\mathbf{S}$.
Output: yes if $Q$ is satisfiable, and no otherwise.
Method: Compute the complete equality relationship graph $E(Q)$ for $Q$.
    (1) If there is an object term of the form $x.A$ for which $type(x.A)$ is undefined
        or equal to a set type, or there is a set term of the form $x.A$, for which
        $type(x.A)$ is undefined or is not equal to a set type, then output no and
        exit.
    (2) If there is an object term $t$ in $Q$ such that $SatType(t)$ is empty, then
        output no and exit.
    (3) If there is an object term $t$ with two distinct atomic values $c_1$ and $c_2$
        both in $[t]$, then output no and exit.
    (4) Output yes.
```

LEMMA 3.1. *If the algorithm SatTestUT outputs yes, then Q is satisfiable.*

Proof. Suppose the algorithm outputs *yes.* We construct a state $\mathbf{s}_Q$ and a satisfying assignment α for Q in $\mathbf{s}_Q$. This assignment will be called the *canonical* asssignment for Q in $\mathbf{s}_Q$. The details of the construction will be applied in later results.

First, to each equivalence class $[t]$ of the complete equality relationship graph of Q, we associate a distinct value $[t]_\alpha$ and a terminal class or an atomic type $type_\alpha([t])$. The type $type_\alpha([t])$ is defined to be $type(s)$ for any term s of the query in $[t]$. This is well defined in the case of object terms by statement (2). For set terms $x.A$, note that if $t \in [x.A]$, then t must be of the form $y.A$ for some variable $y \in [x]$, by the fact that set terms do not occur in equations in Q and construction of $E(Q)$. Thus $type_\alpha([x.A])$ may be defined to be $type(x.A)$ in this case. The values $[t]_\alpha$ are assigned as follows.

Val1: If $type_\alpha([t])$ is an atomic type, and there is an atomic value c in $[t]$, then $[t]_\alpha = c$. By statement (3), c is the unique atomic value in $[t]$.

Val2: If $type_\alpha([t])$ is an atomic type and there is no atomic value in $[t]$, then take $[t]_\alpha$ to be any value of type $type_\alpha([t])$, such that no two distinct equivalence classes are assigned the same value. (This is possible because the atomic types are infinite.)

Val3: If $type_\alpha([t])$ is a terminal class, then the value $[t]_\alpha$ is defined to be just the equivalence class $[t]$ itself. In this case these values will be interpreted below as objects in the state to be constructed.

Val4: If none of the above cases apply, then $type_\alpha([t])$ is a set type. In this case $[t]_\alpha$ is defined to be the set $\{[v]_\alpha \mid$ "$v \in z.A$" is an atom in Q for some variable z in $[t]\}$. (Note that the variables v must be object variables, so the values $[v]_\alpha$ are already defined by cases Val1–Val3.)

Observe that it follows from this definition that distinct equivalence classes are assigned distinct values.

We now construct a state $\mathbf{s}_Q = (\mathbf{O}, \mathbf{I}_c, \mathbf{I}_v)$. We take the set of objects $\mathbf{O}$ of this state to be $\{[t] \mid [t]$ is an equivalence class of Q such that $type_\alpha([t])$ is a terminal class$\}$. These objects are assigned to terminal classes by putting $\mathbf{I}_c([t]) = type_\alpha([t])$, for every $[t] \in \mathbf{O}$. The mapping $\mathbf{I}_v$, defined below, maps each $[t] \in \mathbf{O}$ to a tuple value on $type_\alpha([t])$. First, for each $[t] \in \mathbf{O}$, define $attr([t])$ to be the set of attributes A such that $x.A$ is a term of Q for some variable $x \in [t]$. We now define attribute values for the object $[t]$ as follows. Note that by conditions (1) and (2), the set $attr([t])$ is a subset of the set of attributes in $\mathbf{I}_c([t])$. For each attribute $A \in attr([t])$, the value assigned to the attribute A for the object $[t]$ is $[x.A]_\alpha$, where x is any variable in $[t]$ such that $x.A$ is a term of Q. (Note that the definition of the complete equality relationship graph ensures that if x and y are variables in $[t]$ such that $x.A$ and $y.A$ are terms of Q, then $[x.A] = [y.A]$, so this definition is independent of the choice of x.) If A is an attribute in $type_\alpha([t])$ but not in $attr([t])$, a null value is assigned to A for the object $[t]$. This completes the definition of the state $\mathbf{s}$.

Finally, we define a mapping α from the variables of Q to this state. For each variable x, we let $\alpha(x)$ be the value $[x]_\alpha$. By Lemma 3.2, Q is satisfiable. ☐

LEMMA 3.2. *The query Q is satisfied in the state $\mathbf{s}_Q$ under the assignment α.*

See [9] for more details (on a slightly different approach to the proof).

THEOREM 3.3. *A terminal conjunctive query Q on $\mathbf{S}$ is satisfiable if and only if the algorithm SatTestUT outputs yes.*

Proof. See [9]. ☐

Unless otherwise stated, we consider only satisfiable terminal conjunctive queries for the rest of this paper.

4. Containment of terminal conjunctive queries. We now set about developing a condition that characterizes containment of terminal conjunctive queries. We remark that some of the results of this section depend crucially on the notion of atoms over a state defined in section 2.4. For the rest of sections 4 and 5, a query Q refers to a terminal conjunctive query Q.

We begin by defining a relation that is intended to capture the equations that must be satisfied under any satisfying assignment for a query. Recall that the terms in an equation must have nonnull interpretations for the equation to hold. Given a query Q, define the relation $\approx$ on the set of terms by $s \approx t$ if either

1. s and t are both terms in the complete equality relationship graph of Q and $[s] = [t]$, or
2. s and t are the same term, which is an atomic value.

Note that the relation $\approx$ is an equivalence relation when restricted to the set of terms of the complete equality relationship graph of Q. However, $\approx$ is not an equivalence relation in general, since we do not have $t \approx t$ for terms t that are not an atomic value or in the complete equality relationship graph of Q. Intuitively, this reflects the fact that such terms may have interpretation Λ under a satisfying assignment, so that the equation $t = t$ does not hold. (We do have $c \approx c$ for atomic values c because the equation $c = c$ will hold under every satisfying assignment.)

We can extend the relation $\approx$ on terms to a relation on atoms by defining $\mathbf{A} \approx \mathbf{A}'$ if $\mathbf{A}$ and $\mathbf{A}'$ are atoms of the same syntactic form and the terms in corresponding positions are $\approx$-related. (For example, if $[x] = [y]$ and $z.A$ is a term of the complete equality relationship graph, then we have $x \in z.A \approx y \in z.Z$, but not $x = z.A \approx y \in z.Z$, because these atoms are of different syntactic forms.)

LEMMA 4.1. *If α is a satisfying assignment for Q in a state $\mathbf{s}$, then*

(i) *If s and t are terms with $s \approx t$, then both $Val(\alpha(s))$ and $Val(\alpha(t))$ are nonnull and $Val(\alpha(s)) = Val(\alpha(t))$.*

(ii) *If $\mathbf{A}$ and $\mathbf{A}'$ are atoms with $\mathbf{A} \approx \mathbf{A}'$, then $\mathbf{s} \models \alpha(\mathbf{A})$ if and only if $\mathbf{s} \models \alpha(\mathbf{A}')$.*

Proof. The claim of part (i) is trivial for constants c, since we always have that $Val(\alpha(c)) = c$ is nonnull. The case where s and t are terms of the complete equality relationship graph with $[s] = [t]$ is by induction on the construction of the complete equality relationship graph, proving also the additional property that $Val(\alpha(t))$ is nonnull for any term in the complete equality relationship graph.

Note that for all terms t of Q, we must have that $Val(\alpha(t))$ is nonnull, since this is required for the atom in which t occurs to be satisfied under α. (There is one exception to this observation, the case in which t is the distinguished term and not equal to a variable. But then t must be an atomic value c, for which $Val(\alpha(t)) = c$ is nonnull.) For equations $s = t$ in Q we must have $Val(\alpha(s)) = Val(\alpha(t))$ in order for α to be satisfying. It is trivial that for an equality edge $t = t$ we have $Val(\alpha(t)) = Val(\alpha(t))$. This establishes the base case of the induction. (The case of equations $t = t$, for terms introduced later in the construction, is similar but uses the additional property.)

Consider next the case of edges $t_1 = t_2$ and $t_2 = t_3$ inducing an edge $t_1 = t_3$. Since we have $[t_1] = [t_2]$ and $[t_2] = [t_3]$, it follows from the inductive hypothesis that $Val(\alpha(t_i))$ is nonnull for $i = 1, \ldots, 3$ and $Val(\alpha(t_1)) = Val(\alpha(t_2))$ and $Val(\alpha(t_2)) = Val(\alpha(t_3))$. It is immediate that $Val(\alpha(t_1)) = Val(\alpha(t_3))$.

Finally, suppose that x and y are variables with $x = y$, an edge of the complete equality relationship graph, and that $x.A$ is a node of the complete equality relationship graph. By the induction hypothesis $Val(\alpha(x))$ and $Val(\alpha(y))$ are nonnull and equal, and $Val(\alpha(x).A)$ is nonnull. It follows that $Val(\alpha(y.A)) = Val(\alpha(y).A)$ is equal to $Val(\alpha(x).A)$ and hence nonnull, and equal to $Val(\alpha(x.A))$.

Part (ii) follows directly from part (i) using the fact that the definition of satisfaction depends only on the values of the terms in the atoms $\mathbf{A}$ and $\mathbf{A}'$. $\square$

In addition to the atoms in a query, certain other atoms will always be satisfied with respect to any satisfying assignment for the query. For example, if a query contains atoms $x = y$ and $y \in C$, then every satisfying assignment also makes the atom $x \in C$ true. The following notion is intended to characterize such atoms. A query Q is said to *derive* an atom $\mathbf{A}$ if

1. $\mathbf{A}$ is of the form $t \in T$ where T is a basic type and $t \approx c$ for some atomic value c of type T, or
2. $\mathbf{A}$ is of the form $t_1 = t_2$, where t_1 and t_2 are terms satisfying $t_1 \approx t_2$, or
3. Q contains an atom $\mathbf{A}'$ such that $\mathbf{A}' \approx \mathbf{A}$.

We write $Q \vdash \mathbf{A}$ if Q derives the atom $\mathbf{A}$. The following result shows that every atom derived by a query in fact holds under any satisfying assignment.

LEMMA 4.2. *Let α be any satisfying assignment for the query Q in the state $\mathbf{s}$. If $\mathbf{A}$ is an atom such that $Q \vdash \mathbf{A}$, then $\mathbf{s} \models \alpha(\mathbf{A})$.*

Proof. We consider each clause of the definition of derivation:

1. Suppose $\mathbf{A}$ is of the form $t \in T$, where T is an atomic type and $t \approx c$, where c is an atomic value of type T. By Lemma 4.1, we have $Val(\alpha(t)) = Val(\alpha(c)) = c$. It follows that $\mathbf{s} \models \alpha(\mathbf{A})$.
2. If $\mathbf{A}$ is of the form $s = t$ with $s \approx t$, then by Lemma 4.1(i), $Val(\alpha(s)) = Val(\alpha(t))$, with both nonnull, so $\mathbf{s} \models \alpha(\mathbf{A})$ by definition.
3. Otherwise, Q contains an atom $\mathbf{A}'$ such that $\mathbf{A}' \approx \mathbf{A}$. Since α is a satisfying assignment, we have that $\mathbf{s} \models \alpha(\mathbf{A}')$. By Lemma 4.1(ii), it follows that $\mathbf{s} \models \alpha(\mathbf{A})$ also. $\square$

We now set about showing that a converse to this result holds for the state $\mathbf{s}_Q$ constructed form a query Q. We begin by relating atoms over this state to atoms of the query. Define an *inverse* of the canonical mapping α for the query Q to be a function ω mapping each object of $\mathbf{s}_Q$ and each atomic value to either a variable of Q or an atomic value, such that

1. for all atomic values c such that there exists a variable y of Q with $\alpha(y) = c$, we have that $\omega(c)$ is a variable with this property,
2. for all atomic values c such that there exists no variable y of Q with $\alpha(y) = c$,

we have $\omega(c) = c$, and

3. for all objects $[t]$ of $\mathbf{s}_Q$ we have that $\omega([t])$ is a variable in $[t]$. (Note that each object of $\mathbf{s}_Q$ is an equivalence class of terms that must contain a variable by condition (ii) of the definition of well-formed queries.)

We may extend ω to a mapping from terms over the state $\mathbf{s}_Q$ to terms formed using the variables of Q by defining $\omega([t].A) = \omega([t]).A$ for all terms of the form $[t].A$. The following result explains why we call such a mapping an inverse of α. (To understand the condition on (ii), observe that ω is not defined on sets occurring in $\mathbf{s}_Q$.)

LEMMA 4.3. *If ω is an inverse of the canonical mapping α from Q to $\mathbf{s}_Q$, then*

(i) *for all terms t of Q we have $\omega(\alpha(t)) \approx t$,*

(ii) *for all terms t over $\mathbf{s}_Q$ for which $Val(t)$ is nonnull and not equal to a set, we have $\omega(Val(t)) \approx \omega(t)$, and*

(iii) *for all terms t over $\mathbf{s}_Q$ for which $Val(t)$ is nonnull, $\omega(t)$ is either an atomic value or a term in the complete equality relationship graph of Q.*

Proof. For (i) we consider three cases, according to whether $\alpha(t)$ is an atomic value, object, or attribute term.

Consider first the case where $\alpha(t)$ is the atomic value c. Note that t cannot be an attribute term since these must be mapped to attribute terms. If t is an atomic value, t must be equal to c, so $\omega(\alpha(t)) = c \approx c = t$. If t is a variable, there exists a variable y with $\alpha(y) = c$ and $\omega(c) = y$. Since the construction guarantees that distinct equivalence classes are mapped by α to distinct values, we must have $t \in [y]$. Thus $\omega(\alpha(t)) = y \approx t$.

Next, consider the case in which $\alpha(t)$ is the object $[x]$. As this case can only arise from case Val3 of the construction, we have $t \in [x]$, so $\omega(\alpha(t)) = \omega([x]) \approx t$.

Finally, if $\alpha(t)$ is of the form $o.A$, where o is an object of $\mathbf{s}_Q$, then t is a term of the form $x.A$, where x is a variable, and $o = [x]$. Thus $\omega(\alpha(t)) = \omega([x].A) = \omega([x]).A$. Let $\omega([x])$ be the variable y. Because y must be in $[x]$ and $x.A$ is a term of the complete equality graph, $y.A$ is also a term of the complete equality graph, with $[x.A] = [y.A]$. Hence we have that $\omega(\alpha(t)) = y.A \approx x.A = t$.

We prove (ii) and (iii) together. Note that if t is an atomic value or an object of $\mathbf{s}_Q$, then $\omega(t)$ is a variable of Q or an atomic value, so the claim of (iii) holds. In this case we also have $Val(t) = t$, so the claim of (ii) follows directly from the fact that $\omega(t)$ is an atomic value or a term of the complete equality relationship graph, so that $\omega(t) \approx \omega(t)$.

Suppose next that t is a term over $\mathbf{s}_Q$ of the form $[x].A$ (where, without loss of generality, x is a variable) for which $Val(t)$ is defined and not equal to a set. By definition, $\omega([x].A) = \omega([x]).A = y.A$ for some variable $y \in [x]$. Moreover, by construction of $\mathbf{s}_Q$, there exists a variable $z \in [x]$ such that $z.A$ is a term in the complete equality relationship graph. It follows that $\omega([x].A)$ are in the complete equality relationship graph, establishing the claim of (iii) in this case. (Note also that $x.A$ is in the complete equality relationship graph.)

Moreover, we have either $Val(t) = [x.A]$ or $Val(t) = c$ for some atomic value c. To complete the proof of (ii) we consider each of these cases individually.

Suppose first that $Val(t) = [x.A]$. Let $\omega([x])$ be the variable $y \in [x]$. Then $x \approx y$. Hence $\omega(Val(t)) = \omega([x.A]) \approx x.A \approx y.A = \omega(t)$. Next, if $Val(t)$ is the atomic value c then, by construction of $\mathbf{s}_Q$, one of the following cases applies:

1. In case (Val1) of the construction, we have $c \in [x.A]$. Thus $\omega(Val(t)) = \omega(c)$. If there does not exist a variable v with $\omega(v) = c$, then $\omega(c) = c \approx x.A$. If there does exist such a variable, and $\omega(c) = v$, then we must have $v \in [x.A]$.

Hence, here $\omega(c) = v \approx x.A$. In either case, it follows that $\omega(\mathit{Val}(t)) \approx \omega(t)$.

2. In case (Val2) of the construction, $x.A$ is of atomic type, but $[x.A]$ contains no atomic value. In this case, there exists a variable y in $[x.A]$, for which we must have $\alpha(y) = c$. Without loss of generality, take y to be the variable such that $\omega(c) = y$. Then $\omega([x].A) = \omega([x]).A \approx x.A \approx y = \omega(\mathit{Val}(t))$. ☐

We may extend an inverse ω of α to a mapping from atoms $\mathbf{A}$ over $\mathbf{s}_Q$ to atoms over the variables of Q by defining $\omega(\mathbf{A})$ to be the atom obtained by substituting for each term t over $\mathbf{s}_Q$ in $\mathbf{A}$ the term $\omega(t)$.

We now show that the atoms derived by a query Q completely capture the set of atoms of a certain form holding in the state $\mathbf{s}_Q$. Define an atom over $\mathbf{s}_Q$ to be *terminal* if it is of one of the following forms:

1. $t \in T$, where t is a term over $\mathbf{s}_Q$ and T is an atomic type or terminal class,
2. $s = t$, where s and t are terms over $\mathbf{s}_Q$ such that neither $\mathit{Val}(s)$ nor $\mathit{Val}(s)$ is a set, and
3. $t \in o.A$, where t is a term over $\mathbf{s}_Q$ and o is an object of $\mathbf{s}_Q$.

Intuitively, the terminal atoms are those that may occur as images under satisfying assigments of a well-formed query.

LEMMA 4.4. *Let ω be any inverse of α. Then for all* terminal *atoms A over $\mathbf{s}_Q$ such that $\mathbf{s}_Q \models \mathbf{A}$ we have $Q \vdash \omega(\mathbf{A})$.*

Proof. We consider each of the possible cases for the atom A.

1. If $\mathbf{A}$ is of the form $c \in T$, where c is an atomic value of type T, then $Q \vdash \omega(\mathbf{A})$ by the first clause of the definition of derivation.
2. Suppose $\mathbf{A}$ is of the form $[t] \in C$, where C is a terminal class and $[t]$ is an object of $\mathbf{s}_Q$, with $I_c([t]) = C$. By construction of $\mathbf{s}_Q$, there exists a variable $x \in [t]$ such that $x \in C$ is an atom of Q. Since $\omega([t])$ is also an element of the equivalence class $[t]$, we have $x \approx \omega([t])$, so the atom $\omega(A)$ is $\approx$-related to the atom $x \in C$. Hence, $Q \vdash \omega(\mathbf{A})$ by the third clause of the definition of derivation.
3. Suppose that $\mathbf{A}$ is of the form $s = t$, where s and t are terms over $\mathbf{s}_Q$. For this equation to hold in $\mathbf{s}_Q$, we must have $\mathit{Val}(s) = \mathit{Val}(t)$, with both nonnull. Since the atom is terminal, neither value is a set. Hence, by Lemma 4.3(ii), we have $\omega(s) \approx \omega(\mathit{Val}(s)) = \omega(\mathit{Val}(s)) \approx \omega(t)$. It follows that $Q \vdash \omega(s) = \omega(t)$ by the second clause of the definition of derivation.
4. Suppose that $\mathbf{A}$ is an atom of the form $a \in b.A$, where a is a value or object of $\mathbf{s}_Q$ and b is an object of $\mathbf{s}_Q$. By construction of $\mathbf{s}_Q$, for this atom to hold in $\mathbf{s}_Q$ there must exist an object term t in Q and a set term $x.A$ such that $t \in x.A$ is an atom of Q and $\alpha(t) = a$ and $\alpha(x) = b$. Since $\omega(a) = \omega(\alpha(t)) \approx t$ and $\omega(b) = \omega(\alpha(x)) \approx x$ by Lemma 4.3(i), it follows the third clause of the definition of derivation that $Q \vdash \omega(t) \in \omega(b).A$, i.e., $Q \vdash \omega(\mathbf{A})$. ☐

We note that the corresponding property could not be established for atoms over $\mathbf{s}_Q$ expressing equations between sets. For example, for the query $Q = \{x \mid 1 \in x.A \wedge 1 \in y.B\}$ we find that in $\mathbf{s}_Q$ we have $\mathbf{s}_Q \models [x].A = [y].B$, since $\mathit{Val}([x].A) = \{1\} = \mathit{Val}([y].B)$, but not $Q \vdash x.A = y.B$, since $[x.A] \neq [y.B]$.

We are now ready to state the characterization of containment of queries. First, define a *variable mapping* from a query Q_2 to a query Q_1 to be a function μ mapping each variable of Q_2 to either a variable of Q_1 or to an atomic value. Such a mapping can be extended to a mapping from terms in the complete equality graph of Q_2 to expressions formed from atomic values and variables of Q_1 by taking $\mu(c) = c$ for all atomic values c, and $\mu(x.B) = \mu(x).B$ for all variables x. (Note that the expression $\mu(x).B$ need not be a term of the complete equality graph of Q_1, or even a term. For example, if $\mu(x)$ is the atomic value c, then this expression is $c.B$, which is not a term, and is uninterpretable in our language, since atomic values do not have attributes.)

We will deal with the composition of various such mappings. Recall that if $f : X \to Y$ and $g : Y \to Z$ are functions, then the composition $g \circ f$ is the function from X to Z defined by $g \circ f(x) = g(f(x))$.

Define a *containment mapping* μ from a query Q_2 to a query Q_1 to be a variable mapping from Q_2 to Q_1 such that

1. if t_1 is the distinguished term of Q_1 and t_2 is the distinguished term of Q_2, then $\mu(t_2) \approx_1 t_1$ (where $\approx_1$ is the relation derived from Q_1), and
2. for every atom A of Q_2, we have $Q_1 \vdash \mu(A)$.

The following result shows that containment mappings characterize containment between terminal conjunctive queries.

THEOREM 4.5. *If Q_1 and Q_2 are terminal conjunctive queries, then $Q_1 \subseteq Q_2$ if and only if there exists a containment mapping from Q_2 to Q_1.*

Proof. Suppose first that μ is a containment mapping from Q_2 to Q_1. We need to show that $Q_1 \subseteq Q_2$. For this, let $\mathbf{s}$ be any state and suppose that α is a satisfying assignment for Q_1 in the state $\mathbf{s}$, so that $\alpha(t_1) \in Q_1(\mathbf{s})$, where t_1 is the distinguished term of Q_1. We show that $\alpha(t_1)$ is also in $Q_2(\mathbf{s})$. Define the assignment β for Q_2 in $\mathbf{s}$ by $\beta = \alpha \circ \mu$. If $\mathbf{A}$ is an atom of Q_2, then since μ is a containment mapping we have by definition of containment that $Q_1 \vdash \mu(\mathbf{A})$. By Lemma 4.2 it follows that $\mathbf{s} \models \alpha(\mu(\mathbf{A}))$, i.e., that $\mathbf{s} \models \beta(\mathbf{A})$. Since this holds for every atom of Q_2 it follows that β is a satisfying assignment of Q_2 in $\mathbf{s}$. Thus $Q_2(\mathbf{s})$ contains $\beta(t_2)$, where t_2 is the distinguished term of Q_2. Because μ is a containment mapping, we have $\mu(t_2) \approx t_1$. Thus, by Lemma 4.1 we have that $\beta(t_2) = \alpha(\mu(t_2)) = \alpha(t_1)$ is in $Q_2(\mathbf{s})$, as promised. This completes the proof that $Q_1 \subseteq Q_2$, establishing the implication from right to left in the lemma.

To prove the converse, assume that there exists no containment mapping from Q_2 to Q_1. We show that Q_1 is not contained in Q_2 by establishing that $Q_1(\mathbf{s}_{Q_1})$ is not a subset of $Q_2(\mathbf{s}_{Q_1})$. In particular, we argue that if α is the canonical assignment of Q_1 in $\mathbf{s}_{Q_1}$ and t_1 is the distinguished term of Q_1, then $\alpha(t_1)$ is not in $Q_2(\mathbf{s}_{Q_1})$. Note that, on the other hand, $\alpha(t_1)$ is an element of $Q_1(\mathbf{s}_{Q_1})$ by Lemma 3.2.

To show that $\alpha(t_1)$ is not in $Q_2(\mathbf{s}_{Q_1})$, assume to the contrary that β is a satisfying assignment for Q_2 in $\mathbf{s}_{Q_1}$ with $\beta(t_2) = \alpha(t_1)$, where t_2 is the distinguished term of Q_2. We derive a contradiction to the assumption that there exists no containment mapping from Q_2 to Q_1. In particular, let ω be any inverse of α and consider the mapping $\mu = \omega \circ \beta$. Note that this must be a variable mapping from Q_2 to Q_1. We claim that μ is a containment mapping from Q_2 to Q_1.

To see this, note first that $\mu(t_2) = \omega(\beta(t_2)) = \omega(\alpha(t_1)) \approx t_1$. Thus μ satisfies the first clause of the definition of containment mapping. Next, note that if $\mathbf{A}$ is an atom of Q_2, then since β is a satisfying assignment we have that $\mathbf{s}_{Q_1} \models \beta(\mathbf{A})$. Since $\beta(\mathbf{A})$ is a terminal atom over $\mathbf{s}_{Q_1}$ we have by Lemma 4.4 that $Q_1 \vdash \omega(\beta(\mathbf{A}))$, i.e., that $Q_1 \vdash \mu(\mathbf{A})$. Thus μ satisfies the second condition of the definition of containment mapping, completing the proof that μ is a containment mapping from Q_2 to Q_1 and yielding the desired contradiction. $\square$

5. Minimization of terminal conjunctive queries.

In this section, we define a notion of minimality of terminal conjunctive queries and derive an algorithm that, given a terminal conjunctive query as input, finds a minimal equivalent query among all terminal conjunctive queries.

Let Q be a terminal conjunctive query. A *minimal* terminal conjunctive query of Q is a terminal conjunctive query equivalent to Q with the number of variables minimal among such terminal conjunctive queries. We now show how to find minimal queries.

We begin with a number of lemmas concerning containment mappings. In the rest of the discussion, we use subscripting to indicate the query with respect to which we compute the equivalence classes.

LEMMA 5.1. *Let μ be a containment mapping from the satisfiable terminal conjunctive query Q_2 to the satisfiable terminal conjunctive query Q_1.*

(i) *If t is a term of the complete equality relationship graph of Q_2, then $\mu(t)$ is either an atomic value or a term of the complete equality relationship graph of Q_1.*

(ii) *Let $\approx_1$ and $\approx_2$ be the relations on terms corresponding to the queries Q_1, Q_2, respectively. For all terms s and t, if $s \approx_2 t$ then $\mu(s) \approx_1 \mu(t)$.*

Proof. We establish (i) and (ii) simultaneously. If $s \approx_2 t$ because both s and t are the atomic value c, then we have $\mu(s) = \mu(t) = c$, so $\mu(s) \approx_1 \mu(t)$. It therefore remains to show that if $[s]_2 = [t]_2$, where s and t are terms of the complete equality relationship graph of Q_2, then $\mu(s)$ and $\mu(t)$ are terms of the complete equality relationship graph of Q_1 and $[\mu(s)]_1 = [\mu(t)]_1$. We prove this by induction on the construction of the complete equality relationship graph of query Q_2. Note that it is immediate from the fact that μ is a containment mapping that for all terms s of Q such that $\mu(s)$ is not an atomic value, we have $\mu(s)$ equal to a term in the complete equality relationship graph of Q_1. This is because s occurs in an atom $\mathbf{A}$ of Q_2 and Q_1 derives the atom $\mu(\mathbf{A})$.

In the case of edges $s = s$, we clearly have $\mu(s) \approx_1 \mu(s)$. For edges $s = t$ corresponding to atoms of Q_2, we have $\mu(s) \approx_1 \mu(t)$ because μ is a containment mapping. Suppose that an edge $s = u$ is derived from edges $s = t$ and $t = u$ of the complete equality relationship graph of Q_2 for which we have $\mu(s) \approx_1 \mu(t)$ and $\mu(t) \approx_1 \mu(u)$. Since all the latter terms are in Q_1, it follows from the fact that $\approx_1$ is an equivalence relation on the terms of Q_1 that $\mu(s) \approx_1 \mu(u)$.

Finally, suppose that an edge $x.A = y.A$ is derived from an edge $x = y$ and a term $x.A$ of the complete equality relationship graph of Q_2. We assume by way of induction that $\mu(x) \approx_1 \mu(y)$ and that the term $\mu(x.A)$ occurs in the complete equality relationship graph of Q_1. Now $\mu(x.A) = \mu(x).A$, so $\mu(x)$ must be a variable, else Q_1 would not be satisfiable. Since $\mu(x) \approx_1 \mu(y)$, it follows similarly that $\mu(y)$ must be a variable. Thus, $\mu(y.A) = \mu(y).A$ is a term of the complete equality relationship graph of Q_1 and $\mu(x).A \approx_1 \mu(y).A$, by the inductive hypothesis and the construction of the complete equality relationship graph of Q_1. ▯

LEMMA 5.2. *Let μ be a containment mapping from the satisfiable terminal conjunctive query Q_2 to the satisfiable terminal conjunctive query Q_1. If $\mathbf{A}$ is an atom such that $Q_1 \vdash \mathbf{A}$, then $Q_2 \vdash \mu(\mathbf{A})$.*

Proof. We consider the three cases of the definition of derivation. First, suppose $\mathbf{A}$ is of the form $t \in T$, where T is an atomic type and t is $\approx$-related to the atomic value c of type T. Then by Lemma 5.1(ii) we have $\mu(t) \approx_1 c$, so $Q_1 \vdash \mu(t) \in T$. Second, if $\mathbf{A}$ is the atom $s = t$ and $s \approx_2 t$, then by Lemma 5.1(ii) we have $\mu(s) \approx_1 \mu(t)$, so $Q_1 \vdash \mu(s) = \mu(t)$. Finally, if Q_2 contains the atom $\mathbf{A}' \approx_2 \mathbf{A}$, then by Lemma 5.1(ii) we have $\mu(\mathbf{A}') \approx_1 \mu(\mathbf{A})$. Since μ is a containment mapping it is also the case that $Q_1 \vdash \mu(\mathbf{A}')$. It follows using the definition of derivation and the fact that $\approx_1$ is an equivalence relation on terms of the complete equality relationship graph that $Q_1 \vdash \mu(\mathbf{A})$. ☐

LEMMA 5.3. *Let Q_1, Q_2, and Q_3 be satisfiable terminal conjunctive queries. If μ_1 is a containment mapping from Q_1 to Q_2 and μ_1 is a containment mapping from Q_2 to Q_3, then $\mu_2 \circ \mu_1$ is a containment mapping from Q_1 to Q_3.*

Proof. Let t_1, t_2, and t_3 be the distinguished terms of Q_1, Q_2, and Q_3, respectively, and let $\approx_1, \approx_2$, and $\approx_3$ be the relations on terms derived form these queries. Since μ_1 and μ_2 are containment mappings, we have $\mu_1(t_1) \approx_2 t_2$ and $\mu_2(t_2) \approx_3 t_3$. It follows, using Lemma 5.1(ii), that $\mu_2 \circ \mu_1(t_1) \approx_3 \mu_2(t_2) \approx_3 t_3$. This establishes that $\mu_2 \circ \mu_1$ satisfies the first condition of the definition of containment mapping.

It remains to show that if $\mathbf{A}$ is an atom of Q_1, then $Q_3 \vdash \mu_2 \circ \mu_1(\mathbf{A})$. Since μ_1 is a containment mapping we have that $Q_2 \vdash \mu_1(\mathbf{A})$. It follows, using Lemma 5.2 and the fact that μ_2 is a containment mapping, that $Q_3 \vdash \mu_2 \circ \mu_1(\mathbf{A})$. ☐

Let $Q = \{t \mid M\}$ be a conjunctive query and suppose μ is a variable mapping on Q. Define $\mu(Q)$ to be the conjunctive query with the distinguished term $\mu(t)$ obtained by replacing each atom $\mathbf{A}$ of Q by the atom $\mu(\mathbf{A})$.

PROPOSITION 5.4. *Let Q be a terminal conjunctive query. Suppose there is a containment mapping μ from Q to itself. Then $\mu(Q)$ is equivalent to Q.*

Proof. It is easy to check that μ is a containment mapping from Q to $\mu(Q)$, so we have by Theorem 4.5 that $\mu(Q) \subseteq Q$. To show $Q \subseteq \mu(Q)$, we show that there is a containment mapping from $\mu(Q)$ to Q. We claim that the identity mapping i is such a mapping. Note first that the distinguished term of $\mu(Q)$ is $\mu(t)$, where t is the distinguished term of Q, and we have $i(\mu(t)) = \mu(t) \approx t$, because μ is a containment mapping. It remains to show that for every atom $\mathbf{A}$ of Q, we have for the corresponding atom $\mu(\mathbf{A})$ of $\mu(Q)$ that $Q \vdash i(\mu(\mathbf{A}))$. That is, we need $Q \vdash \mu(\mathbf{A})$. This is immediate from the fact that μ is a containment mapping. ☐

The following describes how to obtain a minimal terminal conjunctive query. Say that a variable mapping μ from a query Q_1 to a query Q_2 is *bijective* if $\mu(x)$ is a variable of Q_2 for every variable x of Q_1 and the restriction of μ to the set of variables of Q_1 is a bijective mapping to the set of variables of Q_2.

THEOREM 5.5. *A terminal conjunctive query Q is minimal if all containment mappings from Q to itself are bijective.*

Proof. Suppose that all containment mappings from Q to itself are bijective, but that Q is not minimal. We establish a contradiction. Since Q is not minimal, there is a minimal terminal conjunctive query Q' equivalent to Q which has fewer variables. Now by Theorem 4.5, the fact that $Q' \equiv Q$ implies that there is a containment mapping μ from Q to Q' and also that there is a containment mapping ω from Q' to Q. It follows from Lemma 5.3 that the composite mapping $\omega \circ \mu$ is a variable mapping from Q to itself. By the assumption, $\omega \circ \mu$ is bijective. But this means that the number of variables of Q' is at least as large as the number of variables of Q, contradicting the choice of Q'. ☐

The converse to Theorem 5.5 follows from the following.

THEOREM 5.6. *Let Q_1 and Q_2 be minimal terminal conjunctive queries. Suppose $Q_1 \equiv Q_2$. Then every containment mapping from one query to the other is bijective.*

Proof. Let μ be a containment mapping from Q_1 to Q_2. Since these queries are equivalent, there also exists a containment mapping ω from Q_2 to Q_1. By Lemma 5.3, the composite mapping $\mu \circ \omega$ is a containment mapping from Q_2 to itself. Suppose that the image of the set of variables of Q_2 under $\mu \circ \omega$ does not contain all the variables of Q_2. Then the query $(\mu \circ \omega)(Q_2)$, which is equivalent to Q_2 by Proposition 5.4, has fewer variables than Q_2. This contradicts the minimality of Q_2. This shows that the image of the set of variables of Q_2 under $\mu \circ \omega$ contains all the variables of Q_2. It follows that Q_1 has at least as many variables as Q_2. A similar argument using the containment mapping $\omega \circ \mu$ from Q_1 to Q_1 shows that Q_2 has at least as many variables as Q_1. Since $\mu \circ \omega$ covers the variables of Q_2, it now follows that μ must in fact be a bijection between the variables of Q_1 and Q_2. $\square$

Given a satisfiable terminal conjunctive query Q, the algorithm to find a minimal terminal conjunctive query Q' equivalent to Q is as follows. Consider all containment mappings μ from Q to itself. Choose Q' to be one of the queries $\mu(Q)$ that has the fewest variables among such queries. The minimality of Q' follows from Theorem 5.6.

Note that there may be some further optimizations possible for the query Q' so obtained, since this may contain atoms of the form $c \in T$ or $c = c$, where c is an atomic value of type T. Such atoms, since they are derivable even from an empty query, can be deleted, yielding an equivalent query.

6. Containment and optimization of conjunctive queries. In this section, we study the containment of conjunctive queries and show how to obtain optimal conjunctive queries. The optimal conjunctive queries obtained are expressed as unions of terminal conjunctive queries and are optimal among all unions of terminal conjunctive queries. In section 6.1, we characterize containment of conjunctive queries by solving the containment problem for unions of terminal conjunctive queries. In section 6.2, we give our notion of optimality. In section 6.3, we derive an algorithm, given a conjunctive query, for finding an optimal union of terminal conjunctive queries. We first use an example to illustrate our notion of optimality and the approach taken in obtaining an optimal query.

Example 6.1. Let us consider the following query defined on the schema in Example 1.1.

$$Q_1 : \{x \mid \exists y\, \exists z\ (x \in \text{Vehicle}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Client}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$

This query retrieves all those vehicles that have been rented to a discount client. By Proposition 2.1, Q_1 is equivalent to the union of the following terminal conjunctive queries:

$$S_1 : \{x \mid \exists y\, \exists z\ (x \in \text{Auto}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Normal}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$
$$S_2 : \{x \mid \exists y\, \exists z\ (x \in \text{Auto}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Discount}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$
$$S_3 : \{x \mid \exists y\, \exists z\ (x \in \text{Trailer}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Normal}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$
$$S_4 : \{x \mid \exists y\, \exists z(x \in \text{Trailer}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Discount}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$
$$S_5 : \{x \mid \exists y\, \exists z(x \in \text{Truck}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Normal}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$
$$S_6 : \{x \mid \exists y\, \exists z(x \in \text{Truck}\ \&\ y \in \text{Discount}\ \&\ z \in \text{Discount}\ \&\ x \in y.\text{VehRented}\ \&\ x \in z.\text{VehRented})\}.$$

With algorithm SatTestUT, it can be shown that S_3, S_4, S_5, and S_6 are unsatisfiable. The reason for this is that discount clients are only allowed to rent automobiles and not other types of vehicles. Hence Q_1 is equivalent to $S_1 \cup S_2$. There is a containment mapping from S_2 to S_1; the mapping is to map x to x and y and z to y. By Theorem 4.5, S_1 is redundant and is removed from the union. S_2 can further be minimized by mapping x to x and y and z to y. The resulting optimal query obtained is as follows:

$$S_2' : \{x \mid \exists y(x \in \text{ Auto } \& \ y \in \text{ Discount } \& \ x \in y.\text{VehRented})\}. \qquad \square$$

6.1. A characterization of containment of unions of terminal conjunctive queries. By Proposition 2.1, understanding containment of unions of terminal conjunctive queries suffices to solve the containment problem of conjunctive queries. We have found a characterization of containment for terminal conjunctive queries. We are now ready to state the containment condition for two unions of terminal conjunctive queries.

THEOREM 6.1. *Let $M = Q_1 \cup \cdots \cup Q_s$ and $N = P_1 \cup \cdots \cup P_t$ be two unions of terminal conjunctive queries. $M \subseteq N$ if and only if for each Q_i in M there is a P_j in N such that $Q_i \subseteq P_j$.*

Proof. "If." This part is trivial.

"Only if." Let Q_i be a subquery in M. Let $\mathbf{s}_{Q_i}$ be the state constructed for Q_i. Suppose α is a satisfying assignment from Q_i to $\mathbf{s}_{Q_i}$ and let ω be an inverse of α. Since $M \subseteq N$, there is some P_j such that there is a satisfying assignment γ of object identifiers and atomic values in the state $\mathbf{s}_{Q_i}$ to variables in P_j that gives rise to the answer $\alpha(t_i)$, where t_i is the distinguished term of Q_i. Then $\omega o \gamma$ is a variable mapping from P_j to Q_i. We show that the mapping $\omega o \gamma$ is a containment mapping.

Let $\omega o \gamma(t_j) = v$, where t_j is the distinguished term of P_j. Then $\gamma(t_j) = \alpha(t_i)$. Hence, $\omega(\gamma(t_j)) = \omega(\alpha(t_i))$. By Lemma 4.3(i), $\omega(\alpha(t_i)) \approx t_j$. Let $\mathbf{A}$ be an atom of P_j. Then $\gamma(\mathbf{A})$ is a terminal atom over $\mathbf{s}_{Q_i}$ and $\mathbf{s}_{Q_i} \models \gamma(\mathbf{A})$. By Lemma 4.4, $Q_i \vdash \omega$ $(\gamma(\mathbf{A}))$. Hence $\omega o \gamma$ is a containment mapping. By Theorem 4.5, $Q_i \subseteq P_j$. $\square$

As a corollary, we solve the problem of determining when one conjunctive query contains the other one.

6.2. Search-space-optimal queries. We now introduce our notion of optimality, which tries to capture the intuition that the number of variables as well as their search spaces are minimal among all equivalent queries. In a conjunctive query, each variable is associated with a set of terminal classes or atomic types which denotes the search space of the variable. Without knowing the physical data organization for various classes, a good criterion of evaluating various equivalent queries is by comparing the set of variables in a query and their associated search spaces.

Example 6.2. Let us consider again the vehicle rental schema. The following three queries can be shown to be equivalent:

$$\begin{aligned} Q_1 : \{x \mid \exists y \exists z \ (&x \in \text{Auto} \ \& \ y \in \text{Discount} \ \& \ z \in \text{Vehicle} \ \& \ x \in y.\text{VehRented} \\ &\& \ z \in y.\text{VehRented})\}. \\ Q_2 : \{x \mid \exists y \ (&x \in \text{Vehicle} \ \& \ y \in \text{Discount} \ \& \ x \in y.\text{VehRented})\}. \\ Q_3 : \{x \mid \exists y \ (&x \in \text{Auto} \ \& \ y \in \text{Discount} \ \& \ x \in y.\text{VehRented})\}. \end{aligned}$$

If we consider the domain of the type of a variable as its search space, then Q_1 has more variables and has a larger search space than Q_2. Although Q_2 and Q_3 have the same number of variables, the search space associated with variables in Q_2 is greater than that in Q_3. Q_3 is considered to be more optimal since the number of variables as well as the search space are minimal. $\Box$

Existing work on exact minimization tries to minimize the number of joins in an expression [2]. The notion of optimality we shall propose attempts to generalize that idea.

A *multiset* is a set or bag of elements in which *duplicate* elements are allowed. Let S and T be two multisets. The *bag union* of S and T, denoting $S \uplus T$, is a multiset obtained from merging elements in the operands, such that for every element x in S or T, the number of occurrences of x in the bag union is the sum of the numbers of occurrences of x in S and in T. Clearly the bag union operator is commutative and associative. We say S is a *bag subset* of T and denotes $S \sqsubseteq T$ if for every element x in S, if there are n occurrences of x in S, then there are at least n occurrences of x in T.

Let Q be a conjunctive query and x a variable in Q. Define *term-class*(Q, x) $=\{E \mid x \in C_1 \sqcup \cdots \sqcup C_n$ is the range atom in Q associated with the variable x, and E is a terminal subtype of C_i, for some $1 \leq i \leq n\}$. Informally, *term-class*(Q, x) gives the terminal descendent classes or atomic types over which the variable x is ranging in the query. Let $x_1, \ldots, x_n$ be the set of variables in Q. Then *term-class*(Q) is a multiset defined as *term-class*$(Q, x_1) \uplus \cdots \uplus$ *term-class*(Q, x_n).

We are now ready to define our notion of optimality. Let $Q = Q_1 \cup \cdots \cup Q_n$ and $P = P_1 \cup \cdots \cup P_m$ be two unions of conjunctive queries. Q is said to be *at least as optimal* as P, denotes $Q \leq P$, if *term-class*$(Q_1) \uplus \cdots \uplus$ *term-class*$(Q_n) \sqsubseteq$ *term-class*$(P_1) \uplus \cdots \uplus$ *term-class*(P_m).

A query Q is *search-space-optimal* among a set of queries $\mathbf{S}$ if for all P in $\mathbf{S}$ such that P is equivalent to $Q, P \leq Q$ implies $Q \leq P$. For search-space-optimal queries, the object search spaces are minimal among all equivalent queries in the set $\mathbf{S}$. The query Q_3 in Example 6.2 is a search-space-optimal query.

COROLLARY 6.2. *If Q is a minimal terminal conjunctive query, then Q is a search-space-optimal query among all terminal conjunctive queries.*

Proof. The proof of the corollary follows from Theorem 5.6 and the derivability of range atoms. $\Box$

6.3. Optimization of unions of terminal conjunctive queries. In this subsection, we study the optimization of unions of terminal conjunctive queries. We show how to obtain a search-space-optimal query among all unions of terminal conjunctive queries.

A union of terminal conjunctive queries $Q_1(\mathbf{s}, t) \cup \cdots \cup Q_n(\mathbf{s}, t)$ is *nonredundant* if there are no Q_i and $Q_j, i \neq j$, such that $Q_i \subseteq Q_j$. We can transform a union of terminal conjunctive queries to an equivalent nonredundant union by finding Q_i and Q_j, $i \neq j$, such that $Q_i \subseteq Q_j$ and deleting Q_i from the union until no more subquery can be removed.

The following is an important property about nonredundant unions of terminal conjunctive queries.

THEOREM 6.3. *Let $M = Q_1 \cup \cdots \cup Q_s$ and $N = P_1 \cup \cdots \cup P_t$ be two nonredundant unions of terminal conjunctive queries. Then $M \equiv N$ if and only if for each Q_i in M, there is a unique P_j in N such that $Q_i \equiv P_j$ and vice versa. Moreover, $s = t$.*

Proof. "If" follows from Theorem 6.1.

For "Only if", suppose $M \equiv N$. Let Q_i be a subquery in M. By Theorem 6.1, there is a P_j in N such that $Q_i \subseteq P_j$. By the assumption on equivalence and by

Theorem 6.1, there is Q_p in M such that $P_j \subseteq Q_p$. If $i \neq p$, then $Q_i \subseteq P_j$ and $P_j \subseteq Q_p$. This implies $Q_i \subseteq Q_p$ and contradicts the nonredundancy of M. Hence $i = p$ and $Q_i \equiv P_j$. If $s \neq t$, say, $s < t$, then there is a P_j which is redundant. This is a contradiction. It follows that $s = t$. ☐

The following is an algorithm for finding an optimal union of terminal conjunctive queries for a conjunctive query.

Algorithm Optimization. Given a conjunctive query Q, find an equivalent union of terminal conjunctive queries which is search-space-optimal among all unions of terminal conjunctive queries.

Input: A conjunctive query Q.
Output: An equivalent union of terminal conjunctive queries.
Method:
(1) Convert Q into an equivalent union of terminal conjunctive queries.
(2) Remove unsatisfiable subqueries from the union using the algorithm *SatTestUT*.
(3) Remove any redundant subqueries from the union.
(4) Minimize each of the remaining subqueries.
(5) Output the union of resulting terminal conjunctive queries.

Theorem 6.4. *The union of terminal conjunctive queries output by Algorithm Optimization is equivalent to Q and is a search-space-optimal query among all unions of terminal conjunctive queries.*

Proof. By Proposition 2.1, Theorem 3.3, and definitions of redundancy and minimization, the union, say Q', output is equivalent to the input Q. Let $P = P_1 \cup \cdots \cup P_n$ be a union of terminal conjunctive queries that is equivalent to Q. Without loss of generality, let us assume that P is nonredundant and each subquery is minimal. By Theorem 6.3, there is a one-to-one correspondence between the two unions. By Theorem 5.6 and the fact that both unions are unions of terminal conjunctive queries, $Q' \leq P$. ☐

Let us look at the following example.

Example 6.3. Let us consider a query defined on the schema in Figure 4:

$$Q_1 : \{x \mid \exists y\, \exists s\ (x \in N \ \&\ y \in H \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$

By Proposition 2.1, Q_1 is equivalent to the union of the following terminal conjunctive queries:

$$S_1 : \{x \mid \exists y\, \exists s\ (x \in T_1 \ \&\ y \in I \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$
$$S_2 : \{x \mid \exists y\, \exists s\ (x \in T_2 \ \&\ y \in I \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$
$$S_3 : \{x \mid \exists y\, \exists s\ (x \in T_3 \ \&\ y \in I \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$
$$S_4 : \{x \mid \exists y\, \exists s\ (x \in T_1 \ \&\ y \in J \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$
$$S_5 : \{x \mid \exists y\, \exists s\ (x \in T_2 \ \&\ y \in J \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$
$$S_6 : \{x \mid \exists y\, \exists s\ (x \in T_3 \ \&\ y \in J \ \&\ s \in J \ \&\ y = x.B \ \&\ y \in x.A \ \&\ s \in x.A)\}.$$

With algorithm SatTestUT, S_2, S_3, S_4, and S_5 are unsatisfiable. Hence Q_1 is equivalent to $S_1 \cup S_6$. Neither subquery in the union contains the other and hence the union is nonredundant. Since variables in S_1 range over different terminal classes, S_1 is minimal. It can easily be shown that S_6 can be minimized further. A minimal form is as follows:

$$S'_6 : \{x \mid \exists y\ (x \in T_3 \ \&\ y \in J \ \&\ y = x.B \ \&\ y \in x.A)\}.$$

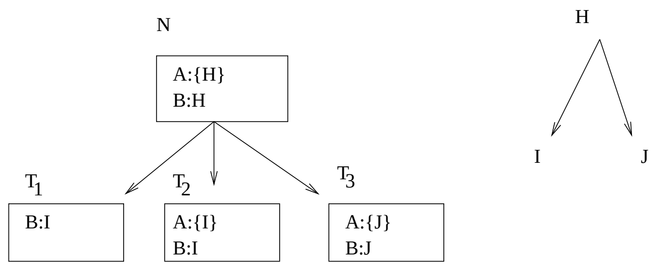


FIG. 4. *An inheritance hierarchy.*

The union of terminal conjunctive queries output by the algorithm is $S_1 \cup S'_6$, which is a search-space-optimal query for Q_1 among all unions of terminal conjunctive queries. □

Algorithm *Optimization* only produces an optimal query expressed as a union of terminal conjunctive queries. This form need not be the most desirable form to be executed. For instance, Q_1 in Example 6.3 is equivalent to the following query:

$$Q_2 : \{x \mid \exists y \, \exists s \; (x \in T_1 \sqcup T_3 \;\&\; y \in H \;\&\; s \in J \;\&\; y = x.B \;\&\; y \in x.A \;\&\; s \in x.A)\}.$$

Throughout the discussion, we made *no* assumption on how data are being physically organized. It could be the case that, given certain information on data organization, Q_2 is a better form to be evaluated than the union produced by the algorithm. However, the union of terminal conjunctive queries produced could be used as a basis to generate an equivalent query in a more desirable form. It would be interesting to see how other information could be used to synthesize a more optimal query for the union.

7. Complexity of the containment problem. In this section, we investigate the time complexity for determining containment of conjunctive queries. We begin with the simple case of terminal conjunctive queries. The containment problem for terminal conjunctive queries is clearly in NP. A relational query is called a select-project-join (SPJ)-query if only selections with constant, projection, and natural join are used in the query. It is well known that the class of relational SPJ-expressions can be expressed as a tagged tableau [2]. Every such tagged tableau can be translated into a conjunctive query without the set membership construct. In [2], it was shown that the problem of determining containment of SPJ-expressions is an NP-complete problem. Consequently, the containment problem of terminal conjunctive queries is also NP-complete. In fact, it can be shown that the containment problem of terminal conjunctive queries involving only range and set membership atoms is NP-complete.

COROLLARY 7.1. *The problem of determining containment for terminal conjunctive queries is an NP-complete problem.*

Proof. The proof follows from the argument above. □

Corollary 7.1 implies that testing containment of conjunctive queries is NP-hard. We shall show that the problem is in Π_2^P of the polynomial hierarchy [28]. The proof of this result is similar to a proof in [24]. A language L is in Π_2^P if its complement is in Σ_2^P, the class of languages which can be recognized by a nondeterministic polynomial-time algorithm with an oracle from NP. An *oracle* for a class of decision problems **C** enables one to decide any problem in **C** in unit time. The classes Σ_2^P and Π_2^P contain NP and are contained in PSPACE.

THEOREM 7.2. *The problem of determining containment for conjunctive queries is in* Π_2^P.

Proof. Let E_1 and E_2 be two conjunctive queries. We describe a nondeterministic polynomial-time algorithm with an oracle from NP that answers "yes" if and only if $E_1 \not\subseteq E_2$. By Theorem 6.1, $E_1 \not\subseteq E_2$ if and only if there is a terminal conjunctive query in the union for E_1, say E_3, such that $E_3 \not\subseteq E_2$. E_3 can be guessed nondeterministically from E_1 by assigning terminal classes or atomic types to variables involved. After guessing the query, we test if E_3 is satisfiable. Testing satisfiability of terminal conjunctive queries can be performed in polynomial time [9]. If E_3 is satisfiable, then we ask the oracle if $E_3 \subseteq E_2$. If the oracle answers "no," then the algorithm answers "yes." It remains to show that determining $E_3 \subseteq E_2$ is in NP. By Theorem 6.1, $E_3 \subseteq E_2$ if and only if there is a terminal conjunctive query in the union for E_2, say E_4, such that $E_3 \subseteq E_4$. Thus we can guess E_4 as before and then check in nondeteministic polynomial time that $E_3 \subseteq E_4$. Hence our claim is proved. ◻

We are now ready to show that the containment problem is Π_2^P-hard.

A Π_2 formula of quantified propositional logic is an expression,

$$\varphi = \forall p_1 \cdots p_n \exists p_{n+1} \cdots p_{n+m}[\alpha],$$

where α is a formula of propositional logic containing only the propositional variables $p_1, \ldots, p_{n+m}$. Such an expression is true if for every assignment of Boolean truth value to the variables $p_1, \ldots, p_n$, there exists an assignment of truth values to the variables $p_{n+1}, \ldots, p_{n+m}$ under which the formula α is true. The set Π_2-SAT is the set of all true Π_2 formulae. This is a generalization of the problem of satisfiability to the polynomial hierarchy. It is known that the set Π_2-SAT is complete for the level Π_2^p of this hierarchy [29, 32].

THEOREM 7.3. *There exists a fixed schema* **S** *such that problem of deciding, given queries* Q_1 *and* Q_2 *on* **S**, *whether* $Q_1 \subseteq Q_2$ *is* Π_2^p*-hard.*

Proof. By reduction from Π_2-SAT. We show that for every Π_2 formula φ, there is a pair of conjunctive queries Q_1, Q_2 such that φ is true if and only if Q_1 is contained in Q_2.

Define the schema **S** to contain classes C, R, G, V, AND, and NOT. The subclass relationships between these classes are given by $R \prec C$ and $G \prec C$. Thus, the terminal classes are R, G, V, AND, and NOT. The tuple type for both R and G is $[a : C,\ b : INT,\ c : V]$, for V is the empty tuple type $[]$, for AND is $[in_1 : V, in_2 : V, out : V]$, and for NOT is $[in : V, out : V]$.

We first describe the query Q_1. Part of this query will encode the truth tables for conjunction and negation. Intuitively, there are (four) variables ranging over AND which represent the lines of the truth table for "$\wedge$," there are (two) variables ranging over NOT which represent the lines of the truth table of "$\neg$," and there are variables t and f in V which denote the truth values *true* and *false*, respectively. We write $TT(t, f)$ for the following formula:

$$\begin{aligned}
&\exists u_1 \in AND[u_1.\,in_1 = t \;\&\; u_1.\,in_2 = t \;\&\; u_1.\,out = t] \;\&\\
&\exists u_2 \in AND[u_2.\,in_1 = t \;\&\; u_2.\,in_2 = f \;\&\; u_2.\,out = f] \;\&\\
&\exists u_3 \in AND[u_3.\,in_1 = f \;\&\; u_3.\,in_2 = t \;\&\; u_3.\,out = f] \;\&\\
&\exists u_4 \in AND[u_4.\,in_1 = f \;\&\; u_4.\,in_2 = f \;\&\; u_4.\,out = f] \;\&\\
&\exists v_1 \in NOT[v_1.\,in = t \;\&\; v_1.\,out = f] \;\&\\
&\exists v_2 \in NOT[v_2.\,in = f \;\&\; v_2.\,out = t].
\end{aligned}$$

Next, suppose φ has n universally quantified variables. Part of the query Q_1 will have the function of assigning a truth value to each of these variables. We construct for each $i = 1, \ldots, n$ the query $ASGN_i(t, f)$ given by

$$\exists w_{i1} \in R \exists w_{i2} \in C \exists w_{i3} \in G\big[w_{i1}.a = w_{i2} \;\&\; w_{i2}.a = w_{i3} \;\&\; w_{i2}.b = i \;\&\; w_{i3}.b = i \;\&\; w_{i2}.c = t \;\&\; w_{i3}.c = f\big].$$

We now define the query Q_1 to be

$$\{t \mid t \in V \;\&\; \exists f \in V[TT(t, f) \;\&\; ASGN_1(t, f) \;\&\; \cdots \;\&\; ASGN_n(t, f)]\}.$$

After moving the quantifiers to the front, it is clear that Q_1 is a conjunctive query.

Let α be a formula of propositional logic in the propositional constants $p_1, \ldots, p_{n+m}$. We define inductively the formula Φ_α with free variables amongst $\mathbf{x} = x_{p_n}, \ldots, x_{p_{n+m}}$ and x_α. If α is the propositional constant p_i, where $i = 1, \ldots, n$, then

$$\Phi_\alpha = \exists z_{i1} \in R \exists z_{i2} \in G\big[z_{i1}.a = z_{i2} \;\&\; z_{i2}.b = i \;\&\; z_{i2}.c = x_{p_i}\big].$$

If α is the propositional constant p_i, where $i = n+1, \ldots, n+m$, then Φ_α is the null formula **true**. (Note that the variable x_α is x_{p_i} in these cases.) If $\alpha = \beta \& \gamma$, then $\Phi_\alpha(\mathbf{x}, x_\alpha)$ is

$$\exists y_\beta \in AND\, \exists x_\beta x_\gamma \in V \left[\begin{array}{l} y_\beta.\,in_1 = x_\beta \;\&\; y_\beta.\,in_2 = x_\gamma \;\&\; y_\beta.\,out = x_\alpha \;\&\; \\ \Phi_\beta(\mathbf{x}, x_\beta) \;\&\; \Phi_\gamma(\mathbf{x}, x_\gamma) \end{array} \right].$$

If $\alpha = \neg\beta$, then $\Phi_\alpha(\mathbf{x}, x_\alpha)$ is

$$\exists y_\beta \in NOT\, \exists x_\beta \in V\big[x.\,in = x_\beta \;\&\; y_\beta.\,out = x_\alpha \;\&\; \Phi_\beta(\mathbf{x}, x_\beta)\big].$$

We define Q_2 to be the query

$$\big\{x_\alpha \mid x_\alpha \in V \;\&\; \exists x_{p_1} \cdots x_{p_n} \in V\big[\Phi_\alpha(\mathbf{x}, x_\alpha)\big]\big\}.$$

Observe that the class C does not occur in Q_2. Thus, after moving the quantifiers to the front, this query is a terminal conjunctive query.

Note that expansions of Q_1 are obtained by replacing each of the n range atoms $w_{i2} \in C$ by either $w_{i2} \in R$ or $w_{i2} \in G$. There are therefore 2^n such expansions. We first show that each of these expansions uniquely determines an assignment of truth values to the propositional constants $p_1, \ldots, p_n$.

Suppose $1 \leq i \leq n$. Because the constant i has only two occurrences in Q_1, if μ is a mapping from the variables z_{i1}, z_{i2}, x_{p_i} to the variables of an expansion E of Q_1 that preserves the atom $z_{i2}.b = i$ of the formula Φ_{p_i}, then we must have $\mu(z_{i2}) = w_{i2}$ or $\mu(z_{i2}) = w_{i3}$. In case the atom $w_{i2} \in C$ of $ASGN_i$ is expanded as $w_{i2} \in G$, the mapping $z_{i1} \mapsto w_{i1}, z_{i2} \mapsto w_{i2}, x_{p_i} \mapsto t$, is the only mapping that preserves all the equality and range atoms of Φ_{p_i}. This determines the assignment of *true* to p_i. Similarly, in case $w_{i2} \in C$ is expanded as $w_{i2} \in R$, the mapping $z_{i1} \mapsto w_{i2}, z_{i2} \mapsto w_{i3}, x_{p_i} \mapsto f$ is the only such mapping. This determines the assignment of *false* to p_i.

Next, suppose that μ is a mapping from the variables of the formulae Φ_{p_i} for $i = 1, \ldots, n$ and the variables $x_{p_{n+1}}, \ldots, x_{p_{n+m}}$ to the variables of an expansion E of Q_1, such that the atoms of Q_2 containing these variables are preserved. As noted above, this implies that the variables $x_{p_1}, \ldots, x_{p_n}$ are mapped to either t or f. The same holds for the variables $x_{p_{n+1}}, \ldots, x_{p_{n+m}}$, because of the range atoms $x_{p_i} \in V$ in Q_2. Let θ be the truth value assignment that assigns the constant p_i to be *true* if and only if $\mu(x_{p_i}) = t$. Under these conditions, a straightforward induction on the complexity of α shows that there exists a unique extension of the mapping μ to a mapping from the variables of Q_2 to the variables of Q_1 that preserves all the atoms of Q_2. Furthermore, we have $\mu(x_\alpha) = t$ if and only if the formula α is true with respect to the assignment θ. Note also that this mapping μ is a containment mapping from Q_2 to the expansion E if and only if $\mu(x_\alpha) = t$.

It now follows from the observations above that there exists a containment mapping from Q_2 to E for each expansion E of Q_1 if and only if the quantified formula φ is true. □

THEOREM 7.4. *The problem of determining containment for conjunctive queries is complete in* Π_2^P.

Proof. The proof follows from Theorems 7.2 and 7.3. □

8. Conclusion. Query optimization is an important and yet difficult problem in an OODB. The types of attributes in an inheritance hierarchy can be considered as constraints imposed on objects in a state. In this paper, we studied the containment, equivalence, and optimization problems for a class of natural queries called conjunctive queries. A conjunctive query can be expressed as a union of terminal conjunctive queries. We first characterized containment and minimization for terminal conjunctive queries. We then solved the problems of containment and optimization for the class of object-preserving conjunctive queries. The optimal queries are expressed as unions of terminal conjunctive queries. The notion of optimality captures the intuition that an optimal equivalent query logically accesses, in certain sense, the least number of objects in a database. It was shown that testing containment of terminal conjunctive queries is an NP-complete problem. Moreover, the containment problem of conjunctive query in general is Π_2^p-complete.

Acknowledgments. The first author wishes to acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada. Work done in part while the first author was a visitor and the second author an employee of NTT Basic Research Labs, Tokyo. The authors also thank anonymous referees for their constructive comments which greatly enhanced the readability of this paper.

REFERENCES

[1] S. ABITEBOUL AND P. KANELLAKIS, *Object identity as a query language primitive*, in Proceedings of 1989 ACM SIGMOD, Portland, OR, pp. 159–173; also appears as Technical Report 1022, INRIA, April 1989.

[2] A. V. AHO, Y. SAGIV, AND J. D. ULLMAN, *Equivalence among relational expressions*, SIAM J. Comput., 8 (1979), pp. 218–246.

[3] M. ATKINSON, F. BANCILHON, D. DEWITT, K. DITTRICH, D. MAIER, AND S. ZDONIK, *The object-oriented database system manifesto*, in Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989, W. Kim, J.-M. Nicolas, and S. Nishio, eds., North-Holland–Elsevier, Amsterdam, 1990, pp. 40–57.

[4] F. BANCILHON, *Object-oriented database systems*, in Proceedings of the 7th ACM Symposium on Principles of Database Systems, Austin, TX, 1988, pp. 152–162.

[5] J. BANERJEE, W. KIM, AND K. C. KIM, *Queries in object-oriented databases*, in Proceedings of the 4th IEEE International Conference on Data Engineering, Los Angeles, CA, 1988, pp. 31–38.

[6] C. BEERI AND Y. KORNATZKY, *Algebraic optimization of object-oriented query languages*, in Proceedings of the 3rd International Conference on Database Theory, Paris, France, Lecture

Notes in Comput. Sci. 470, Springer-Verlag, Berlin, 1990, pp. 72–88.

[7] A. Borgida, *Type systems for querying class hierarchies with non strict inheritance*, in Proceedings of the 8th ACM Symposium on Principles of Database Systems, Philadelphia, PA, 1989, pp. 394–401.

[8] *The Object Database Standard: ODMG*-93, Release 1.1, R. G. G. Cattell, ed., Morgan Kaufmann, San Mateo, CA, 1994.

[9] E. P. F. Chan, *Testing satisfiability of a class of object-oriented conjunctive queries*, Theoret. Comput. Sci., 134 (1994), pp. 287–309.

[10] E. P. F. Chan, *Containment and minimization of positive conjunctive queries in OODB's*, in Proceedings of the 11th ACM Symposium on Principles of Database Systems, San Diego, CA, 1992, pp. 202–211.

[11] A. Chandra and P. M. Merlin, *Optimal implementation of conjunctive queries in relational databases*, in Proceedings of the 9th ACM Symposium on Theory of Computing, ACM, New York, 1977, pp. 77–90.

[12] S. Cluet and C. Delobel, *A general framework for the optimization of object-oriented queries*, in Proceedings of the ACM SIGMOD, San Diego, CA, 1992, pp. 383–392.

[13] D. H. Fisher, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrette, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan, *IRIS: An object-oriented database management system*, ACM Trans. Office Information Systems, 1 (1987), pp. 48–69.

[14] R. Hull, *A survey of theoretical research on typed complex database objects*, in Databases, J. Paredaens, ed., Academic Press, London, UK, 1987, pp. 193–256.

[15] R. Hull and M. Yoshikawa, *On the equivalence of database restructurings involving object identifiers (extended abstract)*, in Proceedings of the 10th ACM Symposium on Principles of Database Systems, Denver, CO, 1991, pp. 328–340.

[16] M. Kifer, W. Kim, and Y. Sagiv, *Querying object oriented databases*, in Proceedings of the 1992 ACM SIGMOD, San Diego, CA, 1992, pp. 393–402.

[17] W. Kim, *Introduction to Object-Oriented Databases*, MIT Press, Cambridge, MA, 1990.

[18] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, *The ObjectStore database system*, CACM, 34 (1991), pp. 50–63.

[19] R. Lanzelotte, P. Valduriez, M. Ziane, and J-P. Cheiney, *Optimization of nonrecursive queries in OODBs*, in Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, Munich, Germany, Lecture Notes in Comput. Sci. 566, Springer-Verlag, New York, 1991, pp. 1–21.

[20] C. Lecluse and P. Richard, *Manipulation of structured values in object-oriented database system*, in Proceedings of the 2th International Workshop on Database Programming Languages, R. Hull, R. Morrison, and D. Stemple, eds., Morgan Kaufmann, San Mateo, CA, 1990, pp. 113–121.

[21] D. Maier, J. C. Stein, A. Otis, and A. Purdy, *Development of an object-oriented DBMS*, in Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications, Portland, OR, ACM, New York, 1986, pp. 472–482.

[22] *Ontos Object Database Programmer's Guide*, Ontologic Inc., Burlington, MA, 1989.

[23] *The O_2 Programmer Manual*, O_2 Technology, 1991.

[24] Y. Sagiv and M. Yannakakis, *Equivalences among relational expressions with the union and difference operators*, J. ACM, 27 (1980), pp. 633–655.

[25] M. H. Scholl, C. Laasch, and M. Tresch, *Updatable views in object-oriented databases*, in Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, Munich, Germany, Lecture Notes in Comput. Sci. 566, Springer-Verlag, New York, 1991, pp. 189–207.

[26] G. Shaw and S. Zdonik, *OO queries: Equivalence and optimization*, in Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989, W. Kim, J.-M. Nicolas, and S. Nishio, eds., North-Holland–Elsevier, Amsterdam, 1990, pp. 264–278.

[27] D. D. Straube and T. Ozsu, *Queries and query processing in object-oriented database systems*, ACM Trans. Information Systems, 8 (1990), pp. 387–430.

[28] L. J. StockMeyer, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 1–22.

[29] L. J. Stockmeyer and A. R. Meyer, *Word problems requiring exponential time*, in Proceedings of the 5th ACM Symposium on the Theory of Computing, Austin, TX, 1973, pp. 1–9.

[30] J. D. Ullman, *Database theory-past and future*, in Proceedings of the 6th ACM Symposium on Principles of Database Systems, San Diego, CA, 1987, pp. 1–10.

[31] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. I, Computer Science Press, Rockville, MD, 1988.
[32] C. Wrathall, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 23–33.
[33] C. Zaniolo, *The database language GEM*, in Proceedings of the 1983 ACM SIGMOD, San Jose, CA, 1983, pp. 207–218.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1401–1421

ROBUST PLANE SWEEP FOR INTERSECTING SEGMENTS[*]

JEAN-DANIEL BOISSONNAT[†] AND FRANCO P. PREPARATA[‡]

Abstract. In this paper, we reexamine in the framework of robust computation the Bentley–Ottmann algorithm for reporting intersecting pairs of segments in the plane. This algorithm has been reported as being very sensitive to numerical errors. Indeed, a simple analysis reveals that it involves predicates of degree 5, presumably never evaluated exactly in most implementations. Within the exact-computation paradigm we introduce two models of computation aimed at replacing the conventional model of real-number arithmetic. The first model (predicate arithmetic) assumes the exact evaluation of the signs of algebraic expressions of some degree, and the second model (exact arithmetic) assumes the exact computation of the value of such (bounded-degree) expressions. We identify the characteristic geometric property enabling the correct report of all intersections by plane sweeps. Verification of this property involves only predicates of (optimal) degree 2, but its straightforward implementation appears highly inefficient. We then present algorithmic variants that have low degree under these models and achieve the same performance as the original Bentley–Ottmann algorithm. The technique is applicable to a more general case of curved segments.

Key words. computational geometry, segment intersection, plane sweep, robust algorithms

AMS subject classifications. 68Q20, 68Q25, 68Q40, 68U05

PII. S0097539797329373

1. Introduction.

As is well known, computational geometry has traditionally adopted the arithmetic model of exact computation over the real numbers. This model has been extremely productive in terms of algorithmic research, since it has permitted a vast community to focus on the elucidation of the combinatorial (topological) properties of geometric problems, thereby leading to sophisticated and efficient algorithms. Such an approach, however, has a substantial shortcoming, since all computer calculations have finite precision, a feature which affects not only the quality of the results but even the validity of specific algorithms. In other words, in this model, algorithm correctness does not automatically translate into program correctness. In fact, there are several reports of failures of implementations of theoretically correct algorithms (see, e.g., [For87, Hof89]). This state of affairs has engendered a vigorous debate within the research community, as is amply documented in the literature. Several proposals have been made to remedy this unsatisfactory situation. They can be split into two broad categories according to whether they perform exact computations (see, e.g., [BKM$^+$95, FV93, Yap97, She96]) or approximate computations (see, e.g., [Mil88, HHK89, Mil89]).

This paper fine-tunes the exact-computation paradigm. The numerical computations of a geometric algorithm are basically of two types: tests (predicates) and constructions, each with clearly distinct roles. Tests are associated with branching

[*]Received by the editors October 20, 1997; accepted for publication (in revised form) September 10, 1998; published electronically March 15, 2000. This work was partially supported by ESPRIT LTR 21957 (CGAL) and by the U.S. Army Research Office under grant DAAH04-96-1-0013. Part of the work was done while the first author was visiting the Center for Geometric Computing at Brown University.

http://www.siam.org/journals/sicomp/29-5/32937.html

[†]INRIA, 2004 Route des Lucioles, BP 93, 06902 Sophia-Antipolis, France (Jean-Daniel.Boissonnat @sophia.inria.fr).

[‡]Center for Geometric Computing, Computer Science Department, Brown University, 115 Waterman Street, Providence, RI 02912-1910, (franco@cs.brown.edu).

decisions in the algorithm that determine the flow of control, whereas constructions are needed to produce the output data. While approximations in the execution of constructions are often acceptable, approximations in the execution of tests may produce incorrect branching, leading to the inconsistencies which are the object of the criticisms leveled against geometric algorithms. The exact-computation paradigm therefore requires that tests be executed with total accuracy. This will guarantee that the result of a geometric algorithm will be topologically correct albeit geometrically approximate. This also means that robustness is in principle achievable if one is willing to employ the required precision. The reported failures of structurally correct algorithms are entirely attributable to noncompliance with this criterion.

Therefore, geometric algorithms can also be characterized on the basis of the complexity of their predicates. The complexity of a predicate is expressed by the degree of a homogeneous polynomial embodying its evaluation. The degree of an algorithm is the maximum degree of its predicates, and an algorithm is robust if the adopted precision matches the degree requirements.

The "degree criterion" is a design principle aimed at developing low-degree algorithms. This approach involves reexamining under the degree criterion the rich body of geometric algorithms known today, possibly without negatively affecting traditional algorithmic efficiency. A previous paper [LPT96] considered as an illustration of this approach the issue of proximity queries in two and three dimensions. As an additional case of degree-driven algorithm design, in this paper we confront another class of important geometric problems, which have caused considerable difficulties in actual implementations: plane sweep problems for sets of segments. As we shall see, plane sweep applications involve a number of predicates of different degree and algorithmic power. Their analysis not only will lead to new and robust implementations (an outcome of substantial practical interest) but will elucidate on a theoretical level some deeper issues pertaining to the structure of several related problems and the mechanism of plane sweeps.

2. Three problems associated with intersecting segments.

Given is a finite set $\mathcal{S}$ of line segments in the plane. Each segment is defined by the coordinates of its two endpoints. We discuss the three following problems (see Figure 2.1):

- Pb1: Report the pairs of segments of $\mathcal{S}$ that intersect.
- Pb2: Construct the arrangement $\mathcal{A}$ of $\mathcal{S}$, i.e., the incidence structure of the graph obtained interpreting the union of the segments as a planar graph.
- Pb3: Construct the trapezoidal map $\mathcal{T}$ of $\mathcal{S}$. $\mathcal{T}$ is obtained by drawing two vertical line segments (*walls*), one above and one below each endpoint of the segments and each intersection point. The walls are extended either until they meet another segment of $\mathcal{S}$ or to infinity.

Let $S_1, \ldots, S_n$ be the segments of $\mathcal{S}$, and let k be the number of intersecting pairs. We say that the segments are in *general position* if any two intersecting segments intersect in a single point, and all endpoints and intersection points are distinct.

The number of intersection points is no more than the number of intersecting pairs of segments, and both are equal if the segments are in general position. Therefore, the number of vertices of $\mathcal{A}$ is at most k, the number of edges of $\mathcal{A}$ is at most $n+2k$, and the number of vertical walls in $\mathcal{T}$ is at most $2(n+k)$, the bounds being tight when the segments are in general position. Thus the sizes of both $\mathcal{A}$ and $\mathcal{T}$ are $O(n+k)$. We didn't consider here the two-dimensional faces of either $\mathcal{A}$ or $\mathcal{T}$. Including them would not change the problems we address.

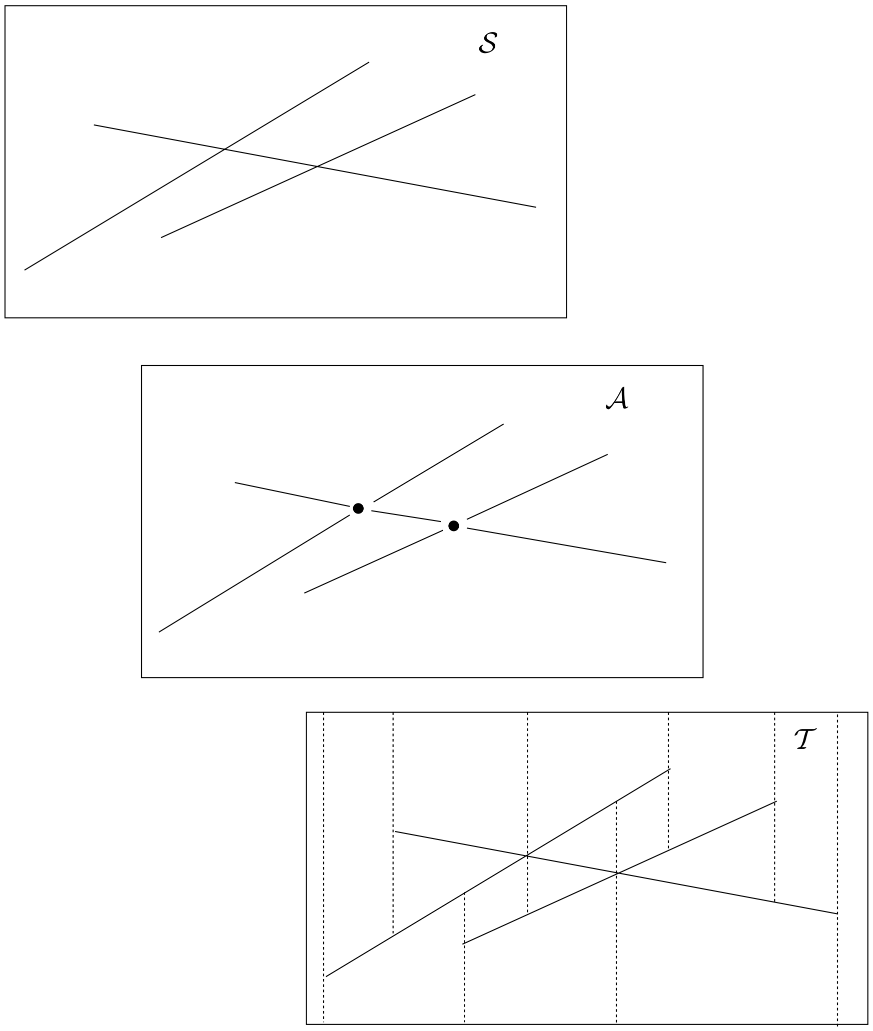


Fig. 2.1. *A set of segments $\mathcal{S}$, the corresponding arrangement $\mathcal{A}$ and its trapezoidal map $\mathcal{T}$.*

3. Algebraic degree and arithmetic models.

It is well known that the efficient algorithms that solve Pb1–Pb3 are very unstable when implemented using nonexact arithmetic, and several frustrating experiences have been reported [For85]. This motivates us to carefully analyze the predicates involved in those algorithms. We first introduce here some terminology borrowed from [LPT96]. See also [Bur96, For93]. We consider each input data (i.e., coordinates of an endpoint of some segment of $\mathcal{S}$) as a *variable.*

An *elementary predicate* is the sign $-$, 0, or $+$ of a homogeneous multivariate polynomial whose arguments are a subset of the input variables. The degree of an elementary predicate is defined as the maximum degree of the irreducible factors (over the rationals) of the polynomials that occur in the predicate and that do not have a constant sign. A *predicate* is more generally a Boolean function of elementary predicates. Its degree is the maximum degree of its elementary predicates.

The *degree of an algorithm* A is defined as the maximum degree of its predicates.

The *degree of a problem* P is defined as the minimum degree of any algorithm that solves P.

In most problems in computational geometry, $d = O(1)$. However, as d affects the speed and/or robustness of an algorithm, it is important to measure d precisely.

In the rest of this paper we consider the degree as an additional measure of algorithmic complexity. Note that, qualitatively, degree and memory requirements are similar, since the arithmetic capabilities demanded by a given degree must be available, albeit they may be never resorted to in an actual run of the algorithm (since the input may be such that predicates may be evaluated reliably with lower precision).

We will consider two arithmetic models. In the first one, called the *predicate arithmetic of degree* d, the only numerical operations that are allowed are the evaluations of predicates of degree at most d. Algorithms of degree d can therefore be implemented exactly in the predicate arithmetic model of degree d. This model is motivated by recent results that show that evaluating the sign of a polynomial expression may be faster than computing its value (see [ABD+97, BY97, BEPP97, Cla92, She96]). This model, however, is very conservative since the nonavailability of the arithmetics required by a predicate is assimilated to an entirely random choice of the value of the predicate.

The second model, called the *exact arithmetic of degree* d, is more demanding. It assumes that values (and not just signs) of polynomials of degree at most d be represented and computed exactly (i.e., roughly as d-fold precision integers). However, higher-degree operations (e.g., a multiplication operation of whose factors is a d-fold precision integer) are appropriately rounded. Typical rounding is rounding to the nearest representable number, but less accurate rounding can also be adequate as will be demonstrated later.

Let P be a predicate (polynomial) of degree d. We ignore for simplicity the size of the coefficients of P, because they are typically small constants; if the input data are all b-bit integers, the size of each monomial in predicate P is upper bounded by 2^{bd}. Moreover, let v be the number of variables that occur in a predicate; for most geometric problems and, in particular, for those considered in this paper, v is a small number. Since the polynomial P is homogeneous, it may contain only highest-degree monomials, whose number is bounded by v^d. It follows that an algorithm of degree d requires precision $p \leq db + d \log v = db + O(1)$ in the exact arithmetic model of degree d.

4. The predicates and the degree of problems Pb1–Pb3. In this section, we analyze the degree of problems Pb1–Pb3 (in subsections 4.1–4.3) and of the standard algorithms for solving these problems (in subsection 4.4).

We use the following notations. The coordinates of point A_i are denoted x_i and y_i. $A_i <_x A_j$ means that the x-coordinate of point A_i is smaller than the x-coordinate of point A_j. The same is true for $<_y$. $[A_i A_j]$ denotes the line segment whose left and right endpoints are, respectively, A_i and A_j, while $(A_i A_j)$ denotes the line containing $[A_i A_j]$. $A_i <_y (A_j A_k)$ means that point A_i lies below line $(A_j A_k)$.

4.1. Predicates. Pb1 requires only that we check if two line segments intersect (Predicate $2'$ below).

Pb2 requires in addition the ability to sort intersection points along a line segment (Predicate 4 below).

Pb3 requires the ability to execute all the predicates listed below.

Predicate 1: $A_0 <_x A_1$.
Predicate 2: $A_0 <_y (A_1A_2)$.
Predicate 2′: $[A_0A_1] \bigcap [A_2A_3] \neq \emptyset$.
Predicate 3: $A_0 <_x [A_1A_2] \bigcap [A_3A_4]$.
Predicate 4: $[A_0A_1] \bigcap [A_2A_3] <_x [A_0A_1] \bigcap [A_4A_5]$.
Predicate 5: $[A_0A_1] \bigcap [A_2A_3] <_x [A_4A_5] \bigcap [A_6A_7]$.

Two other predicates appear in some algorithms that report segment intersections.

Predicate 3′: $(x = x_0) \bigcap [A_1A_2] <_y (x = x_0) \bigcap [A_3A_4]$.
Predicate 4′: $[A_0A_1] \bigcap [A_2A_3] <_y (A_4A_5)$.

4.2. Algebraic degree of the predicates. We now analyze the algebraic degree of the predicates introduced above.

PROPOSITION 4.1. *The degree of Predicates i and i' ($i = 1, \ldots, 5$) is i.*

Proof. We first provide explicit formulae for the predicates.

Evaluating Predicate 2 is equivalent to evaluating the sign of

$$\mathrm{orient}(A_0A_1A_2) = \begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{vmatrix}.$$

Predicate 2′ can be implemented as follows for the case $A_0 <_x A_2$ (otherwise we exchange the roles of $[A_0A_1]$ and $[A_2A_3]$):

```
if $A_1 <_x A_2$ then return false
if $A_3 <_x A_1$
    if orient$(A_0A_1A_2) \times$ orient$(A_0A_1A_3) < 0$ then return true
    else return false
else
    if orient$(A_0A_1A_2) \times$ orient$(A_2A_3A_1) > 0$ then return true
    else return false
```

Therefore, in all cases, Predicate 2′ reduces to Predicate 2.

The intersection point $I = [A_iA_j] \bigcap [A_kA_l]$ is given by

$$I = A_i + (A_j - A_i)\frac{N_I}{D_I} \tag{4.1}$$

with $N_I = \mathrm{orient}(A_iA_kA_l)$ and

$$\begin{aligned} D_I &= \begin{vmatrix} x_j - x_i & x_k - x_l \\ y_j - y_i & y_k - y_l \end{vmatrix} \\ &= \mathrm{orient}(A_iA_jA_k) - \mathrm{orient}(A_iA_jA_l) \\ &\overset{def}{=} \mathrm{Orient}(A_iA_jA_kA_l). \end{aligned}$$

Predicate 4′ reduces to evaluating orient(I, A_0, A_1), where I is the intersection of $[A_2A_3]$ and $[A_4A_5]$. It follows from (4.1) that this is equivalent to evaluating the sign of orient$(A_2A_3A_4A_5)$ and of

$$\mathrm{orient}(A_0A_1A_2) \times \mathrm{Orient}(A_2A_3A_4A_5) - \mathrm{orient}(A_2A_4A_5) \times \mathrm{Orient}(A_0A_1A_2A_3).$$

Predicates 3–5: Explicit formulas for Predicates 3, 4, and 5 can be immediately deduced from the coordinates of the intersection points $I = [A_0A_1] \bigcap [A_2A_3]$ and $J = [A_4A_5] \bigcap [A_6A_7]$ which are given by (4.1). If $A_4A_5 = A_0A_1$, it is clear from (4.1) that $(x_1 - x_0)$ is a common factor of $x_I - x_0$ and $x_J - x_0$.

If $[A_1A_2]$ and $[A_3A_4]$ do not intersect, Predicate $3'$ reduces to Predicate 2. Otherwise, it reduces to Predicate 3.

The above discussion shows that the degree of predicates i and i' is *at most* i. To establish that it is *exactly* i, we have shown in the appendix that the polynomials of Predicates 2, 3, $4'$, and 5, as well as the factor other than $(x_1 - x_0)$ involved in Predicate 4, are irreducible over the rationals.

It follows that the proposition is proved for all predicates. □

Recalling the requirements of the various problems in terms of predicates, we have the following proposition.

PROPOSITION 4.2. *The algebraic degrees of* Pb1, Pb2, *and* Pb3 *are, respectively,* 2*,* 4*, and* 5.

4.3. Implementation of Predicate 3 with exact arithmetic of degree 2. As they will be useful in section 7, in this subsection we present two approximate implementations of Predicate 3 (of degree 3) under the exact arithmetic of degree 2.

From (4.1) we know that Predicate 3 can be written as

$$x_{01}\, D < x_{21}\, N, \tag{4.2}$$

where $x_{01} = x_0 - x_1$, $x_{21} = x_2 - x_1$, $D = |\ \mathrm{Orient}(A_1A_2A_3A_4)|$, and $N = \mathrm{sign}(D)\times$ $\mathrm{orient}(A_1A_3A_4)$.

We stipulate to employ floating-point arithmetic conforming to the IEEE 754 standard [Gol91]. In this standard, simple precision allows us to represent b-bit integers with $b = 24$ and double precision allows us to represent b'-bit integers with $b' = 2b+5 = 53$. The coordinates of the endpoints of the segments are represented in simple precision, and the computations are carried out in double precision. We denote $\oplus$, $\otimes$, and $\oslash$ the rounded arithmetic operations $+, \times$, and $/$. In the IEEE 754 standard, all four arithmetic operations are exactly rounded; i.e., the computed result is the floating-point number that best approximates the exact result.

Since $\mathrm{Orient}(A_1A_2A_3A_4)$ and $\mathrm{orient}(A_1A_3A_4)$ are $(2b+3)$-bit integers, the four terms x_{01}, x_{21}, N, and D in inequality (4.2) can be computed exactly in double precision, and the following monotonicity property is a direct consequence of exact rounding of arithmetic operations.

Monotonicity property 1: $x_{01} \otimes D < x_{21} \otimes N \Longrightarrow x_{01} \times D < x_{21} \times N$.

This implies that the comparison between the two computed expressions $x_{01} \otimes D$ and $x_{21} \otimes N$ evaluates Predicate 3 except when these numbers are equal.

Since in most algorithms, an intersection point is compared with many endpoints, it is more efficient to compute and store the coordinates of each intersection point and to perform comparisons with the computed abscissae rather than to evaluate (4.2) repeatedly. We now illustrate an effective rounding procedure of the x-coordinates of intersection points which gives an alternative approximate implementation of Predicate 3.

LEMMA 4.3. *If the coordinates of the endpoints of the segments are simple precision integers, then the abscissa x_I of an intersection point can be rounded to one of its two nearest simple precision integers using only double precision floating-point arithmetic operations.*

Proof. We assume that the coordinates of the endpoints of the segments are represented as b-bit integers stored as simple precision floating-point numbers. The computations are carried out in double precision.

The rounded value $\tilde{x}_I$ of x_I is given by

$$\tilde{x}_I = \lfloor((x_{21} \otimes N) \oslash D)\rceil \oplus x_1,$$

where $\lfloor X \rceil$ denotes the integer nearest to X (with any tie-breaking rule). If $\varepsilon = 2^{-b'}$ is a strict bound to the modulus of the relative error of all arithmetic operations, $\tilde{X} = (x_{21} \otimes N) \oslash D$ satisfies the following relations:

$$\frac{x_{21}N}{D}(1-2\varepsilon) \approx \frac{x_{21}N}{D}(1-\varepsilon)^2 < \tilde{X} < \frac{x_{21}N}{D}(1+\varepsilon)^2 \approx \frac{x_{21}N}{D}(1+2\varepsilon).$$

As $\frac{x_{21}N}{D} = x_I - x_1 \leq 2^{b+1}$, we obtain

$$|(x_{21} \otimes N) \oslash D - x_{21}N/D)| \lesssim 2^{b+1}22^{-b'} = 2^{-b-3} \ll 1.$$

We round $\tilde{X}$ to the nearest integer $\lfloor \tilde{X} \rceil$. Since $\lfloor \tilde{X} \rceil$ and x_1 are $(b+1)$-bit integers, there is no error in the addition. Therefore, $\tilde{x}_I$ is a $(b+2)$-bit integer and the absolute error on $\tilde{x}_I$ is smaller than 1. □

It follows that, under the hypothesis of the lemma, if E is an endpoint, I is an intersection point, and $\tilde{I}$ is the corresponding rounded point, the following monotonicity property holds.

Monotonicity property 2: $\tilde{I} <_x E \Longrightarrow I <_x E,$
$E <_x \tilde{I} \Longrightarrow E <_x I.$

This implies that the comparison between the x-coordinates of $\tilde{I}$ and E evaluates Predicate 3, except when the abscissae of $\tilde{I}$ and E coincide.

Notice that the monotonicity property does not necessarily hold for two intersection points.

Remark 1. A result similar to Lemma 4.3 has been obtained by Priest [Pri92] for points with floating-point coordinates. More precisely, if the endpoints of the segments are represented as simple precision floating-point numbers, Priest [Pri92] has proposed a rather complicated algorithm that uses double precision floating-point arithmetic and rounds x_I to the nearest simple precision floating-point number. Since it applies to the integers as well, this stronger result also implies the monotonicity property.

4.4. Algebraic degree of the algorithms. The naive algorithm for detecting segment intersections (Pb1) evaluates $\Theta(n^2)$ Predicates $2'$ and thus is of degree 2, which is degree-optimal by the proposition above. Although the time-complexity of the naive algorithm is worst-case optimal, since $0 \leq k \leq \frac{1}{2}n(n-1)$, it is worth looking for an output sensitive algorithm whose complexity depends on both n and k. Chazelle and Edelsbrunner [CE92] have shown that $\Omega(n \log n + k)$ is a lower bound for Pb1 and therefore also for Pb2 and Pb3. A very recent algorithm of Balaban [Bal95] solves Pb1 optimally in $O(n \log n + k)$ time using $O(n)$ space. This algorithm does not solve Pb2 nor Pb3 and, since it uses Predicate $3'$, its degree is 3.

Pb2 can be solved by first solving Pb1 and subsequently sorting the reported intersection points along each segment. This can easily be done in $O((n+k) \log n)$ time by a simple algorithm of degree 4 using $O(n)$ space. A direct (and asymptotically more efficient) solution to Pb2 has been proposed by Chazelle and Edelsbrunner [CE92]. Its time complexity is $O(n \log n + k)$, and it uses $O(n + k)$ space. Their algorithm, which constructs the arrangement of the segments, is of degree 4.

A solution to Pb3 can be deduced from a solution to Pb2 in $O(n + k)$ time using a very complicated algorithm of Chazelle [Cha91]. A deterministic and simple algorithm due to Bentley and Ottmann [BO79] solves Pb3 in $O((n + k) \log n)$ time, which is slightly suboptimal, using $O(n)$ space. This classical algorithm uses the sweep-line paradigm and evaluates $O((n+k)\log n)$ predicates of all types discussed above, and therefore has degree 5. Incremental randomized algorithms [CS89, BDS$^+$92] construct the trapezoidal map of the segments and thus solve Pb3 and have degree 5. Their time complexity and space requirements are optimal (although only as expected performances).

In this paper, we revisit the Bentley–Ottmann algorithm and show that a variant of degree 3 (instead of 5) [Mye85, Sch91] can solve Pb1 with no sacrifice of performance (section 6.1). Although this algorithm is slightly suboptimal with respect to time complexity, it is much simpler than Balaban's algorithm. We also present two variants of the sweep-line algorithm. The first one (section 6.2) uses only predicates of degree at most 2 and applies to the restricted but important special case where the segments belong to two subsets of nonintersecting segments. The second one (section 7) uses the exact arithmetic of degree 2. All these results are based on a (nonefficient) lazy sweep-line algorithm (to be presented in section 5) that solves Pb1 by evaluating predicates of degree at most 2.

Remark 2. When the segments are not in general position, the number s of intersection points can be less than the number k of intersecting pairs. In the extreme, $s = 1$ and $k = n(n-1)/2$. Some algorithms can be adapted so that their time complexities depend on s rather than k [BMS94]. However, a lower bound on the degree of such algorithms is 4 since they must be able to detect if two intersection points are identical, therefore to evaluate Predicate 4′.

5. A lazy sweep-line algorithm.

Let $\mathcal{S}$ be a set of n segments whose endpoints are $E_1, \ldots, E_{2n}$. For a succinct review, the standard algorithm first sorts $E_1, \ldots, E_{2n}$ by increasing x-coordinates and stores the sorted points in a priority queue X, called the *event* schedule. Next, the algorithm begins sweeping the plane with a vertical line L and maintains a data structure Y that represents a subset of the segments of $\mathcal{S}$ (those currently intersected by L, ordered according to the ordinates of their intersections with L). Intersections are detected in correspondence of adjacencies created in Y, either by insertion/deletion of segment endpoints or by order exchanges at intersections. An intersection, upon detection, is inserted into X according to its abscissa. Of course, a given intersection may be detected several times. Multiple detections can be resolved by performing a preliminary membership test for an intersection in X and omitting insertion if the intersection has been previously recorded. An intersection is reported when the sweep-line reaches its abscissa. We stipulate to use another policy to resolve multiple detections, namely to remove from X an intersection point I whose associated segments are no longer adjacent in Y. Event I will be reinserted in X when the segments become again adjacent in Y. This policy has also the advantage of reducing the storage requirement of Bentley–Ottmann's algorithm to $O(n)$ [Bro81].

5.1. Description of the lazy algorithm.

We now describe a modification of the sweep-line algorithm that does not need to process the intersection points by increasing x-coordinates.

First, the algorithm sorts the endpoints of the segments by increasing x-coordinates into an array X. Let $E_1, \ldots, E_{2n}$ be the sorted list of endpoints.

Then the algorithm starts processing *events* and maintains a dictionary Y that stores an ordered subset of the line segments. The events consist of the endpoints and of a subset of the pairs of segments that intersect and are adjacent in Y. Such pairs are called *processable* and will be precisely defined below. The algorithm processes the endpoints by increasing order of their x-coordinates, but, contrary to the standard algorithm, the processable pairs are processed in any order. As a consequence, two intersection points or even an intersection point and an endpoint won't necessarily be processed in the order of their x-coordinates, and Y won't necessarily represent the ordered set of segments intersecting some vertical line L (as in the standard algorithm).

The events are processed in the same way as in the standard algorithm, i.e., processing an endpoint means either the insertion of the corresponding segment into Y or its deletion from it (as appropriate), and processing an intersection means its report and the exchange of the two segments involved. The only difference is that we maintain a set P of processable pairs: each time a pair of intersecting segments become adjacent in Y, we check whether the pair is processable and, in the affirmative, add it to P.

While there are processable pairs, the algorithm extracts any of them from P and processes it. When there are no more processable pairs, the algorithm proceeds to the next endpoint. When there are no more processable pairs and no more endpoints to be processed, the algorithm stops.

To complete the description of the algorithm, we need to define the *processable* pairs. The definition rests on the notions of *active* and *prime* pairs to be given below. We need the following notations. We denote by $L(E_i)$ the vertical line passing through E_i. Slab (E_i, E_{i+1}) denotes the open vertical slab bounded by $L(E_i)$ and $L(E_{i+1})$, and $(E_i, E_{i+1}]$ denotes the semiclosed slab obtained by adjoining line $L(E_{i+1})$ to the open slab (E_i, E_{i+1}). For two segments S_k and S_l, we denote by A_{kl} their rightmost left endpoint, by B_{kl} their leftmost right endpoint, and by I_{kl} their intersection point when they intersect (see Figure 5.1). In addition, W_{kl} denotes the set of segment endpoints that belong to the (closed) region bounded by the vertical lines $L(A_{kl})$ and $L(B_{kl})$ and by the two segments (a trapezoid if the two segments do not intersect, a double-wedge otherwise). We denote by E_{kl} the most recently processed element of W_{kl} and by F_{kl} the element of W_{kl} to be processed next. (Note that E_{kl} and F_{kl} are always defined, since they may respectively coincide with A_{kl} and B_{kl}.) Last, we define sets $W^+_{kl}(E_{kl})$ and $W^-_{kl}(E_{kl})$ as follows. If S_k and S_l do not intersect, $W^+_{kl}(E_{kl}) = \emptyset$ and $W^-_{kl}(E_{kl})$ consists of all points $E \in W_{kl}$, $E_{kl} \leq_x E$. Otherwise, an endpoint $E \in W_{kl}$ belongs to $W^+_{kl}(E_{kl})$ (respectively, to $W^-_{kl}(E_{kl})$) if $E_{kl} \leq_x E$ and if the slab $(E_{kl}, E]$ does (respectively, does not) contain I_{kl}.[1]

DEFINITION 5.1. *Let (S_k, S_l) be a pair of segments, and assume without loss of generality that $S_k \bigcap L(E_{kl}) <_y S_l \bigcap L(E_{kl})$. The pair is said to be* active *if the following conditions are satisfied:*

(1) *S_k and S_l are adjacent in Y,*

(2) *$S_k < S_l$ in Y,*

(3) $F_{kl} \in W^+_{kl}(E_{kl})$ (emptiness condition).

Observe that the emptiness condition implies that the segments intersect (since $W^+_{kl}(E_{kl}) = \emptyset$ if they do not). We now identify a subset of the active pairs, whose processing, as we shall see, has priority.

DEFINITION 5.2. *An active pair of segments (S_k, S_l) is said to be* prime *if the next endpoint to be processed belongs to W_{kl} (i.e., it coincides with $F_{kl} \in W^+_{kl}(E_{kl})$).*

It should be noted that deciding if a pair of intersecting segments is active or prime reduces to the evaluation of Predicates 2 only and the condition $F_{kl} \in W^+_{kl}(E_{kl})$ should

[1]For line segments or even pseudosegments, $W^-_{kl}(E_{kl})$ and $W^+_{kl}(E_{kl})$ do not depend on E_{kl}. However, we keep E_{kl} as a parameter in preparation for the more general case of monotone arcs. See Remark 4 at the end of this section.

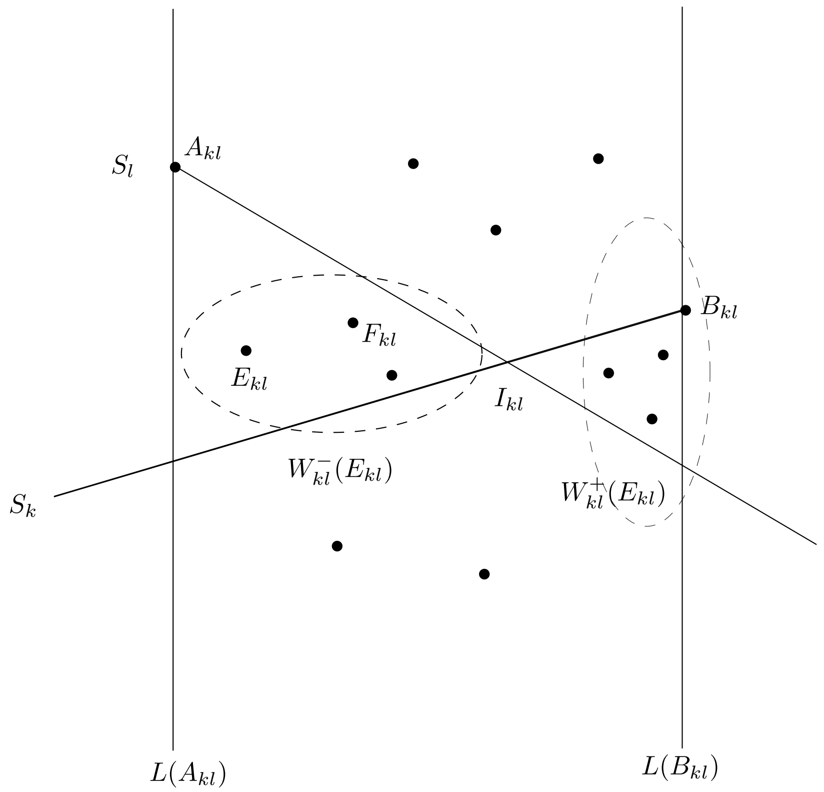


FIG. 5.1. *For the definitions of* $W^-_{kl}(E_{kl})$ *and* $W^+_{kl}(E_{kl})$.

not be construed as implying the comparison of the abscissae of I_{kl} and F_{kl} (Predicate 3). Indeed, if $S_k \bigcap L(A_{kl}) <_y S_l \bigcap L(A_{kl})$ (which can be decided by Predicate 2), then the emptiness condition corresponds to $F_{kl} <_y S_k$ and $S_l <_y F_{kl}$.

For reasons that will be clear below, the set of processable pairs is not specified in the finest detail in this section, since several different implementations are possible. We require only that, at any time, the two following assertions remain true.

(1) All prime pairs are processable.

(2) All processable pairs are active.

In other words, the next endpoint may be processed once there are no more prime pairs, without placing any deadline on the processing of the current active pairs as long as they are not prime.

In the rest of this section, we will simply assume that we have an oracle at our disposal that can decide if a given pair is processable. Clearly, for some instances of the lazy algorithm (e.g., when considering all active pairs as processable), the oracle can be implemented with Predicates 2 only. In such a case, the lazy algorithm involves only Predicates 1 and 2 and is of degree 2 by Proposition 4.1. We will prove its correctness in the next subsection.

The important issue of efficiently detecting the processable pairs will be considered in sections 6 and 7 where several oracles will be introduced. These instances of the lazy algorithm will still be correct but, in some cases, will have a degree higher than 2.

5.2. Correctness of the lazy algorithm. Let $Y^-(E_i)$ and $Y^+(E_i)$ be, respectively, snapshots of the data structure Y immediately before and after processing event E_i, $i = 1, \dots, 2n$. Observe that $Y^-(E_i)$ and $Y^+(E_i)$ differ only by the segment S that has E_i as one of its endpoints. Let $Y(E_i) = Y^-(E_i) \bigcup Y^+(E_i)$. The order relation in Y is denoted by $<$.

THEOREM 5.3. *If Predicates* 1 *and* 2 *are evaluated exactly, the described lazy sweep-line algorithm will detect all pairs of segments that intersect.*

Proof. The algorithm (correctly) sorts the endpoints $E_1, \dots, E_{2n}$ of the segments by increasing x-coordinates into X. Consequently, the set of segments that intersect $L(E)$ and the set of segments in $Y(E)$ coincide for any endpoint E. The proof of the theorem is articulated now as two lemmas and their implications.

LEMMA 5.4. *Two segments have exchanged their positions in Y if and only if they intersect and if the pair has been processed.*

Proof. Let us consider two segments, say S_k and S_l, that do not intersect. Without loss of generality, let $S_k < S_l$ in $Y(A_{kl})$. Assume for a contradiction that $S_l < S_k$ in $Y^-(B_{kl})$. S_k and S_l cannot exchange their positions because they will never form an active pair. Therefore, $S_l < S_k$ in $Y^-(B_{kl})$ can happen only if there exists a segment S_m, $m \neq k, l$, that at some stage in the execution of the algorithm was present in Y together with S_k and S_l and caused one of the following two events to occur.

(1) $S_m > S_l$ and the positions of S_m and S_k are exchanged in Y.

(2) $S_m < S_k$ and the positions of S_m and S_l are exchanged in Y.

In both cases, the segments that exchange their positions are not consecutive in Y, violating condition 1 of Definition 5.1.

Therefore, two segments can exchange their positions in Y only if they intersect and this can happen only when their intersection is processed. Moreover, when the intersection has been processed, the segments are no longer active and cannot exchange their positions a second time. ∎

We say that an endpoint E of S is *correctly placed* if and only if the subset of the segments that are below E (in the plane) coincides with the subset of the segments $< S$ in $Y(E)$, i.e.,

$$\forall S' \in Y(E), \qquad S' < S \Longleftrightarrow S' \bigcap L(E) <_y S \bigcap L(E).$$

Otherwise, E is said to be *misplaced.* (Note that $S \bigcap L(E)$ coincides with E.)

LEMMA 5.5. *If Predicates* 1 *and* 2 *are evaluated exactly, both endpoints of every segment are correctly placed.*

Proof. Assume, for a contradiction, that E of S is the *first* endpoint to be misplaced by the algorithm.

CLAIM 1. *E can be misplaced only if there exist at least two intersecting segments S_k and S_l in $Y^-(E)$ such that E belongs to W_{kl}.*

Proof. First recall that Predicate 2 is the only predicate involved in placing S in Y.

Consider first the case where E is the left endpoint of S. Let (S_k, S_l) be *any* pair of segments in $Y^-(E)$. If $E \notin W_{kl}$, then both S_l and S_k are either above or below E, and their relative order does not affect the placement of E. This establishes that E will be correctly placed in $Y^+(E)$ (i.e., the contrapositive of the necessary condition expressed by the claim).

Suppose now that E is a right endpoint. In this case we shall establish the lemma in its direct form. The left endpoint of S has been correctly placed since it was processed earlier and E is the first one to be misplaced. If E has been misplaced, then, by Lemma 5.4, there exists a segment $S' \in Y^-(E)$ intersecting S to the left of E such that the relative positions of S and S' in $Y^+(A)$ and $Y^-(E)$ are the same (A is the rightmost left endpoint of S and S'), i.e., their order in Y has not been reversed by the algorithm. In this case, $E = B_{kl} \in W_{kl}$ for $S_k = S$ and $S_l = S'$, thus proving that there exists a pair S_l and S_k such that S_l and S_k intersect and $E \in W_{kl}$. ◻

Let S_k and S_l be two segments of $Y^-(E)$ such that $E \in W_{kl}$. Assume without loss of generality that $S_k < S_l$ in $Y(E_{kl})$. Since E is the first endpoint to be misplaced, we have $S_k \bigcap L(E_{kl}) <_y S_l \bigcap L(E_{kl})$. For convenience, we will say that two segments S_p and S_q have been *exchanged between* E' and E'' for two endpoints $E' <_x E''$ if $S_p < S_q$ in $Y^+(E')$ and $S_q < S_p$ in $Y^-(E'')$.

The case where $E \in W^-_{kl}(E_{kl})$ cannot cause any difficulty since S_k and S_l cannot be active between E_{kl} and E, and therefore, S_k and S_l cannot be exchanged between E_{kl} and E, which implies that E is correctly placed with respect to S_k and S_l.

The case where $E \in W^+_{kl}(E_{kl})$ is more difficult. E is not correctly placed only if S_k and S_l are not exchanged between E_{kl} and E, i.e., $S_k < S_l$ in both $Y^+(E_{kl})$ and $Y^-(E)$. We shall prove that this is not possible and therefore conclude that S is correctly placed into Y in this case as well.

Assume, for a contradiction, that S_k and S_l have not been exchanged between E_{kl} and E. As E belongs to $W^+_{kl}(E_{kl})$, S_k and S_l cannot be adjacent in $Y^-(E)$ since otherwise they would constitute a prime pair and they would have been exchanged. Let $(S_k = S_{k_0}, S_{k_1}, \ldots, S_{k_r}, S_{k_{r+1}} = S_l)$ $(r \geq 1)$ be the subsequence of segments of $Y^-(E)$ occurring between S_k and S_l with the further (legitimate) assumption that (S_k, S_l) is a pair of intersecting segments such that $E \in W^+_{kl}(E_{kl})$, for which r is *minimal* (i.e., for which the above subsequence is shortest). For an arbitrary $1 \leq i \leq r$, consider segment S_{k_i} and assume, without loss of generality, that $E <_y S_{k_i}$. As a direct consequence of the fact that $(S_k, S_{k_1}, \ldots, S_{k_r}, S_l)$ is shortest, we observe that E cannot belong to $W^+_{kk_i}(E_{kk_i})$ nor to $W^+_{k_il}(E_{k_il})$. We distinguish two cases.

(i) $E <_y S_{k_i}$ (see Figure 5.2). Clearly, $E \in W_{k_il}$, and, by the above observation, we must have $E \in W^-_{k_il}(E_{k_il})$. It follows that the relative y-orders of S_{k_i} and S_l along $L(E_{k_il})$ and $L(E)$ are the same, hence $S_l \bigcap L(E_{k_il}) <_y S_{k_i} \bigcap L(E_{k_il})$. As E is the first endpoint to be misplaced, the order in $Y^+(E_{k_il})$ agrees with the geometry; i.e., we have $S_l < S_{k_i}$ in $Y^+(E_{k_il})$. Moreover, since the pair (S_{k_i}, S_l) is not active (between E_{k_il} and E, because $E \in W^-_{k_il}(E_{k_il})$) and therefore cannot be exchanged, the same inequality holds in $Y^-(E)$, which contradicts the definition of S_{k_i}.

(ii) $S_{k_i} <_y E$. This case is entirely symmetric to the previous one. It suffices to exchange the roles of S_k and S_l and to reverse the relations $<$ and $<_y$.

Since a contradiction has been reached in both cases, the lemma is proved. ◻

We now complete the proof of the theorem. The previous lemma implies that the endpoints are correctly processed. Indeed let E_i be an endpoint. If E_i is a right endpoint, we simply remove the corresponding segment from Y and update the set of active segments. This can be done exactly since predicates of degree ≤ 2 are evaluated correctly. If E_i is a left endpoint, it is correctly placed in Y on the basis of the previous lemma.

The lemma also implies that all pairs that intersect have been processed. Indeed if S_p and S_q are two intersecting segments such that $S_p \bigcap L(A_{pq}) <_y S_q \bigcap L(A_{pq})$ and $S_q \bigcap L(B_{pq}) <_y S_p \bigcap L(B_{pq})$, the lemma shows that $S_p < S_q$ in $Y^+(A_{pq})$ and $S_p > S_q$ in $Y^-(B_{pq})$, which implies that the pair (S_p, S_q) has been processed (Lemma 5.4).

This concludes the proof of the theorem. ◻

Remark 3. Handling the degenerate cases does not cause any difficulty, and the

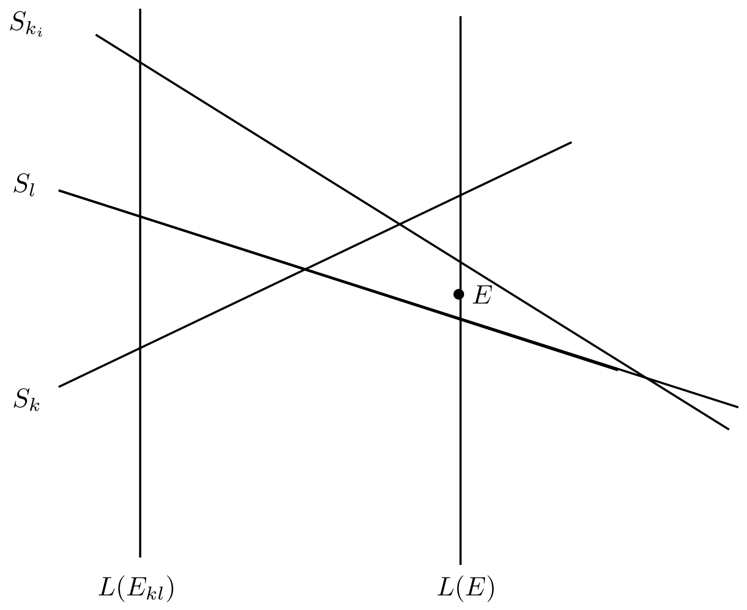


FIG. 5.2. *For the proof of correctness of the lazy algorithm.*

previous algorithm will work with only minor changes. For the initial sorting of the endpoints, we can take any order relation compatible with the order of their x-coordinates, e.g., the lexicographic order.

Remark 4. Theorem 5.3 applies directly to pseudosegments, i.e., curved segments that intersect in at most one point. Lemmas 5.4 and 5.5 also extend to the case of monotone arcs that may intersect at more than one point. To be more precise, in Lemma 5.4, we have to replace "intersect" with "intersect an odd number of times"; Lemma 5.5 and its proof are unchanged provided that we define $W^+_{kl}(E_{kl})$ (respectively, $W^-_{kl}(E_{kl})$) as the subset of W_{kl} consisting of the endpoints E, $E_{kl} \leq_x E$, such that the slab $(E_{kl}, E]$ contains an odd number (respectively, none or an even number) of intersection points. As a consequence, the lazy algorithm (which still uses only Predicates 1, 2, and $2'$) will detect all pairs of arcs that intersect an odd number of times.

Remark 5. For line segments, observe that checking whether a pair of segments is active does not require knowing (and therefore maintaining) E_{kl}. In fact, we can replace condition 3 in the definition of an active pair by the following condition: $S_l <_y F_{kl} <_y S_k$ and $I_{kl} <_x F_{kl}$. If $E_{kl} <_x I_{kl}$, the two definitions are identical and if $I_{kl} <_x E_{kl}$, the pair is not active since, by Lemma 5.5, condition 2 of the definition won't be satisfied.

6. Efficient implementations of the lazy algorithm in the predicate arithmetic model. The difficulty of efficiently implementing the lazy sweep-line algorithm using only predicates of degree at most 2 (i.e., in the predicate arithmetic model of degree 2) is due to verification of the emptiness condition in Definition 5.1 and of the prime-pair condition expressed by Definition 5.2: both conditions require examination of all endpoints that have not been processed yet. One can easily check that various known implementations of the sweep achieve straightforward verification of the emptiness condition by introducing algorithmic complications. The following subsection describes an efficient implementation of the lazy algorithm in the predicate arithmetic model of degree 3. The second subsection improves on this result in a special but important instance of Pb1, namely the case of two sets of nonintersecting segments. The algorithm presented there uses only predicates of degree at most 2.

6.1. Robustness of the standard sweep-line algorithm. We shall run our lazy algorithm under the predicate arithmetic model of degree 3. We then have the capability to correctly compare the abscissae of an intersection and of an endpoint. We refine the lazy algorithm in the following way. Let E_i be the last processed endpoint and let E_{i+1} be the endpoint to be processed next. An active pair (S_k, S_l) that occurs in Y between $Y(E_i)$ and $Y^-(E_{i+1})$ will be processed if and only if its intersection point I_{kl} lies to the right of E_i and not to the right of E_{i+1}. As the slab is free of endpoints in its interior, any pair of adjacent segments encountered in Y (between $Y(E_i)$ and $Y^-(E_{i+1})$) and that intersect within the slab is active. Moreover the intersection points of all prime pairs belong to the slab. It follows that this instance of the lazy algorithm need not explicitly check whether a pair is active or not and therefore is much more efficient than the lazy algorithm of section 5. This algorithm is basically what the original algorithm of Bentley–Ottmann[2] becomes when predicates of degree at most 3 are evaluated. (Recall that the standard algorithm requires the capability to correctly execute predicates of degree up to 5.)

We therefore conclude with the following theorem.

THEOREM 6.1. *If Predicates* 1, 2, *and* 3 *are evaluated exactly, the standard sweep-line algorithm will solve* Pb1 *in* $O((n + k) \log n)$ *time.*

It is now appropriate to briefly comment on the implementation details of the just described modified algorithm. Data structure Y is implemented as usual as a dictionary. Data structure X, however, is even simpler than in the standard algorithm (which uses a priority queue with dictionary access). Here X has a primary component realized as a static search tree on the abscissae of the endpoints $E_1, \ldots, E_{2n}$. Leaf E_j points to a secondary data structure $\mathcal{L}(E_j)$ realized as a conventional linked list, containing (in an arbitrary order) adjacent intersecting pairs in slab $(E_j, E_{j+1}]$. Remember that, when E_j has been processed, all intersecting pairs of $\mathcal{L}(E_j)$ are active. Insertion into $\mathcal{L}(E_j)$ is performed at one of its ends, and so is access for reporting (when the plane sweep reaches slab $(E_j, E_{j+1}]$). Since access to events occurs in data structure X, each pair in X must have pointers to the two corresponding segments in Y, in order to enable the necessary updates. In addition, to effect constant-time removal of a pair (S_h, S_k) due to loss of adjacency, all that is needed is to make bidirectional the mentioned pointers. Notice that the elements of X correspond to pairs of adjacent segments in Y, so that at most two records in X are pointed to by any member of Y. We finally observe that a segment adjacency arising in Y during the execution of the algorithm must be tested for intersection; however, an intersecting pair of adjacent segments is eligible for insertion into X only as long as the plane sweep has not gone beyond the slab containing the intersection in question. As regards the running time, beside the initial sorting of the endpoints and the creation of the corresponding primary tree in time $O(n \log n)$, it is easily seen that each intersection uses $O(\log n)$ time (amortized), thereby achieving the performance of the standard algorithm.

[2]With the policy concerning multiple detections of intersections that is stipulated at the beginning of section 5.

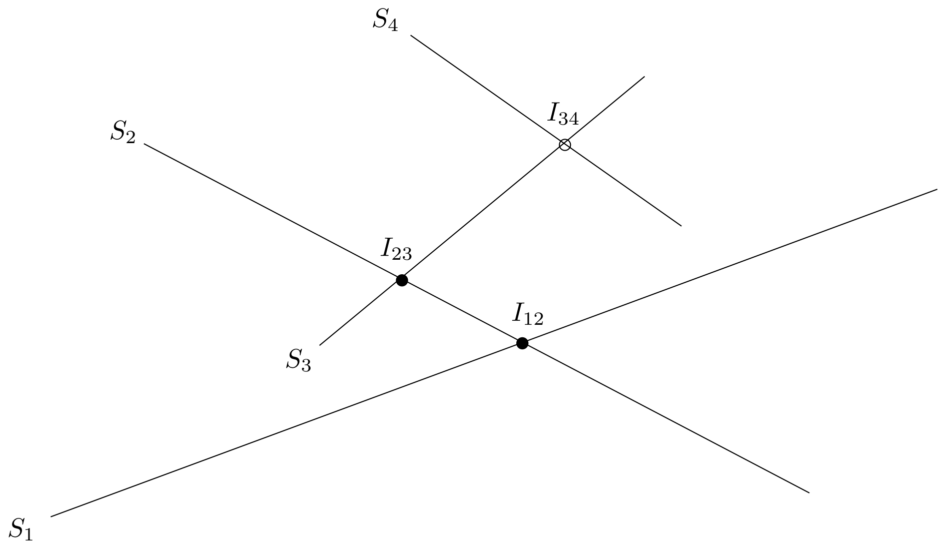


FIG. 6.1. *Assume that the computed x-coordinate of the intersection I_{12} of S_1 and S_2 is (erroneously) found to be smaller than the x-coordinate of the left endpoint of S_3. Y is implemented as a balanced binary search tree. The key associated to a node is the $\lceil \frac{n}{2} \rceil$th element of the corresponding subtree. If all predicates of degree at most 2 are evaluated exactly, S_3 is correctly inserted below S_2, and S_4 is correctly inserted above S_1 and S_2. The different states of Y are (S_1), (S_1, S_2), (S_2, S_1), (S_3, S_2, S_1), and (S_3, S_2, S_1, S_4). Since segments S_3 and S_4 are never adjacent, their intersection I_{34} will not be detected. Observe that the missed intersection point can be arbitrarily far from the intersection point involved in the wrong decision.*

Finally, we note that if only predicates of degree ≤ 2 are evaluated correctly, the algorithm of Bentley–Ottmann may fail to report the set of intersecting pairs of segments. See Figure 6.1 for an example.

Remark 6. The fact that the sweep-line algorithm does not need to sort intersection points had already been observed by several authors including Myers [Mye85], Schorn [Sch91], and Hobby [Hob93]. Myers does not use it for solving robustness problems but for developing an algorithm with an expected running time of $O(n \log n + k)$. Schorn uses this fact to decrease the precision required by the sweep-line algorithm from fivefold to threefold; i.e., Schorn's algorithm uses exact arithmetic of degree 3. Using Theorem 5.3, we will show in section 7 that double precision suffices.

6.2. Reporting intersections between two sets of nonintersecting line segments. In this subsection, we consider two sets of line segments in the plane, $\mathcal{S}_b$ (the blue set) and $\mathcal{S}_r$ (the red set), where no two segments in $\mathcal{S}_b$ (similarly, in $\mathcal{S}_r$) intersect. Such a problem arises in many applications, including the union of two polygons and the merge of two planar maps. We denote by n_b and n_r the cardinalities of $\mathcal{S}_b$ and $\mathcal{S}_r$, respectively, and let $n = n_b + n_r$.

Mairson and Stolfi [MS88] have proposed an algorithm that works for arcs of curve as well as for line segments. Its time complexity is $O(n \log n+k)$, which is optimal, and requires $O(n + k)$ space ($O(n)$ in case of line segments). The same asymptotic time-bound has been obtained by Chazelle et al. [CEGS94] and by Chazelle and Edelsbrunner [CE92]. The latter algorithm is not restricted to two sets of nonintersecting line segments. Other algorithms have been proposed by Nievergelt and Preparata [NP82] and by Guibas and Seidel [GS87] in the case where the segments of $\mathcal{S}_b$ (and $\mathcal{S}_r$) are

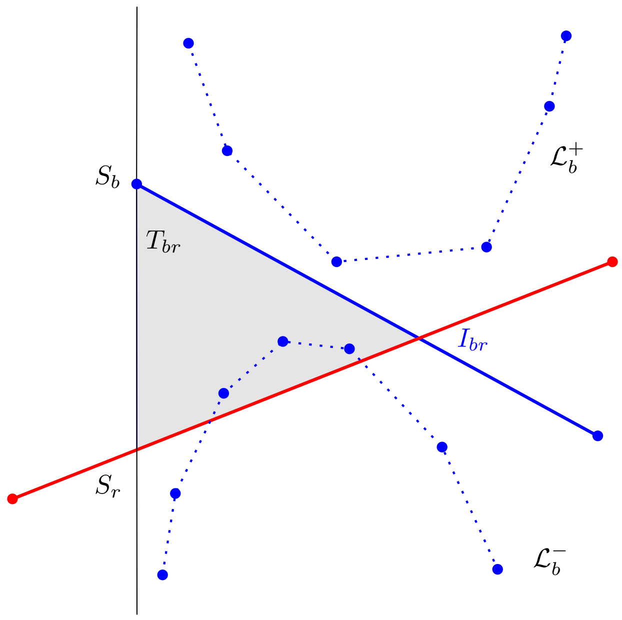


FIG. 6.2. *Notations for the case of two sets of nonintersecting line segments.*

the edges of a subdivision with convex faces. With the exception of the algorithm of Chazelle et al. [CEGS94], all these algorithms construct the resulting arrangement and therefore have degree 4. The algorithm of Chazelle et al. requires comparing the ordinates of the intersections of two segments with a vertical line passing through an endpoint. Therefore it is of degree 3.

We propose here an algorithm that computes all the intersections but not the arrangement. This algorithm uses only predicates of degree ≤ 2 and has time complexity $O((n+k)\log n)$.

Segments are assumed to be nonvertical since intersections with vertical segments can be easily handled with predicates of degree ≤ 2. We say that a point E_i is *vertically visible* from a segment $S_b \in \mathcal{S}_b$ if the vertical line segment joining E_i with S_b does not intersect any other segment in $\mathcal{S}_b$ (the same notion is applicable to $\mathcal{S}_r$). For two intersecting segments $S_b \in \mathcal{S}_b$ and $S_r \in \mathcal{S}_r$, let L be a vertical line to the right of A_{br} such that no other segment intersects L between S_b and S_r (i.e., S_b and S_r are adjacent). We let T_{br} denote the wedge defined by S_b and S_r in the slab between L and $L(I_{br})$ (see Figure 6.2). For a point set $\mathcal{F}$, we let $CH^+(\mathcal{F})$ and $CH^-(\mathcal{F})$ denote its upper and lower convex hulls, respectively.

Our algorithm is based on the following observation.

LEMMA 6.2. *T_{br} contains blue endpoints if and only if it contains a blue endpoint that is vertically visible from S_b. Similarly, T_{br} contains red endpoints if and only if it contains a red endpoint vertically visible from S_r.*

Proof. The sufficient condition is trivial, so we prove only necessity. Assume without loss of generality that $S_r \bigcap L(A_{br}) <_y S_b \bigcap L(A_{br})$. Let $\mathcal{E}$ be the subset of the blue endpoints that belong to T_{br}. Clearly, all vertices of $CH^+(\mathcal{E})$ are vertically visible from S_b. $\square$

Our algorithm has two phases. The second one is the lazy algorithm of section

5. The first one can be considered as a preprocessing step that will help to efficiently find active pairs of segments.

More specifically, our objective is to develop a quick test of the emptiness condition based on the previous lemma. The preprocessing phase is aimed at identifying the candidate endpoints for their potential belonging to wedges formed by intersecting adjacent pairs. Referring to $\mathcal{S}_b$ (and analogously for $\mathcal{S}_r$), we first sweep the segments of $\mathcal{S}_b$ and construct for each blue segment S_b the lists $\mathcal{L}_b^-$ and $\mathcal{L}_b^+$ of blue endpoints that are vertically visible from S_b and lie respectively below and above S_b. The sweep takes time $O(n \log n)$ and the constructed lists are sorted by increasing abscissa. Since there is no intersection point, only predicates of degree ≤ 2 are used. The total size of the lists $\mathcal{L}_b^-$, $\mathcal{L}_b^+$, $\mathcal{L}_r^-$, and $\mathcal{L}_r^+$ is $O(n)$.

As mentioned above, the crucial point is to decide whether or not the wedge T_{br} of a pair of intersecting segments S_b and S_r, adjacent in Y, contains endpoints of other segments. Again, we assume that $S_r < S_b$ in Y. If such endpoints exist, then T_{br} contains either a blue vertex of $CH^+(\mathcal{L}_b^- \cap L^+)$ or a red vertex of $CH^-(\mathcal{L}_b^+ \cap L^+)$. ($L^+$ is the half-plane to the right of line L.)

We will show below that, using predicates of degree ≤ 2, the lists can be preprocessed in time $O(n \log n)$ and that deciding whether or not T_{br} contains endpoints can be done in time $O(\log n)$, using only predicates of degree at most 2.

Assuming for the moment that this primitive is available, we can execute the plane sweep algorithm described earlier. Specifically, we sweep $\mathcal{S}_b$ and $\mathcal{S}_r$ simultaneously, using the lazy sweep-line algorithm of section 5.[3] Each time we detect a pair of (adjacent) intersecting segments S_b and S_r, we can decide in time $O(\log n)$ whether they are "active" or "not active," using only predicates of degree ≤ 2.

We sum up the results of this section in the following theorem.

THEOREM 6.3. *Given n line segments in the plane belonging to two sets $\mathcal{S}_b$ and $\mathcal{S}_r$, where no two segments in $\mathcal{S}_b$ (analogously, in $\mathcal{S}_r$) intersect, there exists an algorithm of optimal degree 2 that reports all intersecting pairs in $O((n+k)\log n)$ time using $O(n)$ storage.*

We now return to the implementation of the primitive described above. Suppose that, for some segment S_i ($i = b$ or r) we have constructed the upper hull $\mathrm{CH}^+(\mathcal{L}_i^- \cap L^+)$. Then we can detect in $O(\log n)$ time if an element of a list, say $\mathcal{L}_b^-$, lies above some segment S_r. More specifically, we first identify among the edges of $CH^+(\mathcal{L}_b^- \cap L^+)$ the two consecutive edges whose slopes are, respectively, smaller and greater than the slope of S_r. This requires only the evaluation of $O(\log |\mathcal{L}_b^-|)$ predicates of degree 2. It then remains to decide whether the common endpoint E of the two reported edges lies above or below the line containing S_r. This can be answered by evaluating the orientation predicate $\mathrm{orient}(E, A_r, B_r)$.

The crucial requirement of the adopted data structure is the ability to efficiently maintain $\mathrm{CH}^+(\mathcal{L}_i^- \cap L^+)$. To this purpose, we propose the following solution.

The data structure associated with a list $\mathcal{L}_i^-$ ($i = b$ or r) represents the upper convex hull $\mathrm{CH}^+(\mathcal{L}_i^-)$ of $\mathcal{L}_i^-$. (Similarly, the data structure associated with a list $\mathcal{L}_i^+$ represents the lower convex hull $\mathrm{CH}^-(\mathcal{L}_i^+)$ of $\mathcal{L}_i^+$.) This implies that a binary search on the convex hull slopes uniquely identifies the test vertex. Since the elements of each list are already sorted by increasing x-coordinates, the data structures can be constructed in time proportional to their sizes and therefore in $O(n)$ time in total. It can be easily checked that only orientation predicates (of degree 2) are involved in this process. To guarantee the availability of $\mathrm{CH}^+(\mathcal{L}_i^- \cap L^+)$, we have to ensure

[3]We can adopt the policy of processing all active pairs before the next endpoint.

that our data structure can efficiently handle the deletion of elements. As elements are deleted in order of increasing abscissa, this can be done in amortized $O(\log |S|)$ time per deletion [HS90, HS96]. It follows that preprocessing all lists takes $O(n)$ time, uses $O(n)$ space, and requires only the evaluation of predicates of degree ≤ 2.

7. An efficient implementation of the lazy algorithm under the exact arithmetic model of degree 2. We shall run the lazy algorithm of section 6.1 under the exact arithmetic model of degree 2, i.e., Predicates 1 and 2 are evaluated exactly but Predicate 3 is implemented with exact arithmetic of degree 2 as explained in section 4.3. Several intersection points may now be found to have the same abscissa as an endpoint. We refine the lazy algorithm in the following way. Let E_i be the last processed endpoint and let E_{i+1} be the endpoint with an abscissa strictly greater than the abscissa of E_i to be processed next. An active pair (S_k, S_l) will be processed if and only if its intersection point is found to lie to the right of E_i and not to the right of E_{i+1}.

We claim that this policy leads to efficient verification of the emptiness condition. Indeed, the intersections of all prime pairs belong to $(E_i, E_{i+1}]$, because $E_{i+1} \in W^+_{kl}(E_{kl}) \Longrightarrow I_{kl} \leq_x E_{i+1}$, and, by the monotonicity property, both implementations of Predicate 3 of section 4.3 will report that $I_{kl} \leq_x E_{i+1}$.

The crucial observation that drastically reduces the time complexity is the following. A pair of adjacent segments (S_k, S_l) encountered in Y between $Y(E_i)$ and $Y(E^-_{i+1})$ whose intersection point is reported to lie in slab $(E_i, E_{i+1}]$ is active if and only if $E_{i+1} \notin W^-_{kl}(E_{kl})$. Indeed, since I_{kl} is found to be $<_x E_{i+2}$, the monotonicity property implies that $I_{kl} <_x E_{i+2}$. Therefore, when checking if a pair is active, it is sufficient to consider just the next endpoint, not all of them.

Theorem 5.3 therefore applies. If no two endpoints have the same x-coordinate, the algorithm is the same as the algorithm in section 6.1 apart from the implementation of Predicate 3. Otherwise, we construct X on the distinct abscissae of the endpoints and store all endpoints with identical x-coordinates in a secondary search structure with endpoints sorted by y-coordinates. This secondary structure will allow us to determine if a pair is active in logarithmic time by binary search. We conclude with the following theorem.

THEOREM 7.1. *Under the exact arithmetic model of degree* 2*, the instance of the lazy algorithm described above solves* $Pb1$ *in* $O((n+k)\log n)$ *time.*

8. Conclusion. Further pursuing our investigations in the context of the exact-computation paradigm, in this paper we have illustrated that important problems on segment sets (such as intersection report, arrangement, and trapezoidal map), which are viewed as equivalent under the Real-RAM model of computation, are distinct if their algebraic degree is taken into account. This sheds new light on robustness issues which are intimately connected with the notion of algebraic degree and illustrates the richness of this new direction of research.

For example, we have shown that the well-known plane sweep algorithm of Bentley–Ottmann uses more machinery than strictly necessary and can be appropriately modified to report segment intersections with arithmetic capabilities very close to optimal and no sacrifice in performance.

Another result of our work is that exact solutions of some problems can be obtained even if approximate (or even random) evaluations of some predicates are performed. More specifically, using less powerful arithmetic than demanded by the application, we have been able to compute the vertices of an arrangement of line segments by constructing an arrangement which may be different from the actual one (and may not even correspond to any set of straight line segments) but still has the same vertex set.

Our work shows that the sweep-line algorithm is more robust than usually believed, proposes practical improvements leading to robust implementations, and provides a better understanding of the sweeping line paradigm. The key to our technique is to relax the horizontal ordering of the sweep. This is one step further after similar attempts, aimed though at different purposes [Mye85, MS88, EG89].

A host of interesting open questions remain. One such question is to devise an output-sensitive algorithm for reporting segment intersections with optimal time complexity and with optimal algebraic degree (that is, 2). It would also be interesting to examine the plane sweep paradigm in general. For example, with regard to the construction of Voronoi diagrams in the plane, one should elucidate the reasons for the apparent gap between the algebraic degrees of Fortune's plane sweep solution and of the (optimal) divide-and-conquer and incremental algorithms.

Appendix. In connection with reducibility of polynomials over a domain of rationality, we choose the rationals as the latter. "Reducible" means "reducible over the rationals."

We first recall that a general determinant is irreducible if its entries are regarded as independent variables [Boc07]. This suffices to prove directly that the degree of Predicate 2 is 2 and will be crucial for completing the proof of irreducibility of the polynomials pertaining to the other predicates. Use will be made of the following theorem.

Let $p(x_1, \ldots, x_n)$ be a multivariate homogeneous polynomial, and let I be a subset of $\{1, \ldots, n\}$, such that for any $j \in I$, x_j is a variable of degree 1 in p. For $i \in \{1, \ldots, n\}$ p can be expressed as

$$p = p_i x_i + p_{i0},$$

where p_i and p_{i0} are polynomials in all variables except x_i. Two polynomials are not considered distinct if they differ just by a multiplicative constant. We have the following.

THEOREM 8.1. *Let $p(x_1, \ldots, x_n)$ be a multivariate homogeneous polynomial with degree at least 3 and $|I| \geq 2$. If for some $i, j \in I$, coefficients p_i and p_j are distinct and irreducible, then p is irreducible.*

Proof. Assume, for a contradiction, that p is reducible. This means that p can be expressed as $p = \phi\psi$, where ϕ and ψ are multivariate homogeneous polynomials over the same variables satisfying $1 \leq degree(\phi), degree(\psi) < degree(p)$. Each degree-1 variable obviously appears in exactly one of the factors. With complete generality, assume that x_i appears in ϕ. We distinguish two cases.

(1) $degree(\phi) > 1$. Expanding ϕ around x_i we obtain $\phi = \phi_i x_i + \phi_{i0}$, so that

$$p = \phi\psi = (\phi_i x_i + \phi_{i0})\psi = \phi_i \psi x_i + \phi_{i0}\psi.$$

This means that $p_i = \phi_i \psi$, with $degree(\phi_i), degree(\psi) \geq 1$, i.e., p_i is reducible, a contradiction.

(2) $degree(\phi) = 1$. Variable x_j appears either in ϕ or in ψ. In the first case, p_i and p_j are both proportional to ψ, i.e., they may differ only by a multiplicative constant and are not distinct, a contradiction.

In the second case, by an argument analogous to that of Case 1, we reach the conclusion that $p_j = \phi\psi_j$ and that $degree(\psi_j) = degree(p) - degree(\phi) - 1 \geq$

$3-1-1=1$. Since $p_j=\phi\psi_j$ and $degree(\phi), degree(\psi_j) \geq 1$, p_j is reducible, another contradiction.

This completes the proof of the theorem. □

The calculation of coefficients p_i, p_j corresponds to taking formal derivatives of p with respect to x_i,x_j. Notice that a derivation reduces the degree by 1. The reader may verify, by explicit and not very enlightening calculations, that the following schedules of derivations lead to irreducible 2×2 determinants. The indices conform with the notation of section 4.1.

Predicate 3: Apply Theorem 8.1 to the pair (x_1, x_2).

Predicate 4: Apply Theorem 8.1 first to (x_2, x_3) and then to (y_6, y_7).

Predicate 4′: Apply Theorem 8.1 first to (x_0, x_1) and then to (x_2, x_3)

Predicate 5: Apply Theorem 8.1 first to (x_1, x_2), next to (x_7, x_8), and finally to (y_5, y_6).

Acknowledgments. We are indebted to H. Brönnimann for having pointed out an error in a previous version of this paper and to O. Devillers for discussions that lead to Lemma 4.3. S. Pion, M. Teillaud, and M. Yvinec are also gratefully acknowledged for their comments on this work. Special thanks are due to two anonymous referees, whose insightful comments have significantly contributed to the quality of the paper.

REFERENCES

[ABD+97] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec, *Evaluating signs of determinants using single-precision arithmetic*, Algorithmica, 17 (1997), pp. 111–132.

[Bal95] I. J. Balaban, *An optimal algorithm for finding segment intersections*, in Proceedings of the 11th Annual ACM Symposium on Computational Geometry, Vancouver, Canada, 1995, pp. 211–219.

[BO79] J. L. Bentley and T. A. Ottmann, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., C-28 (1979), pp. 643–647.

[Boc07] M. Bocher, *Introduction to Higher Algebra*, Macmillan, New York, 1907.

[BDS+92] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec, *Applications of random sampling to on-line algorithms in computational geometry*, Discrete Comput. Geom., 8 (1992), pp. 51–71.

[BEPP97] H. Brönnimann, I. Emiris, V. Pan, and S. Pion, *Computing exact geometric predicates using modular arithmetic with single precision*, in Proceedings of the 13th Annual ACM Symposium on Computational Geometry, Nice, France, 1997, pp. 174–182.

[BY97] H. Brönnimann and M. Yvinec, *Efficient exact evaluation of signs of determinants*, in Proceedings of the 13th Annual ACM Symposium on Computational Geometry, Nice, France, 1997, pp. 166–173.

[Bro81] K. Q. Brown, *Comments on "Algorithms for reporting and counting geometric intersections,"* IEEE Trans. Comput., C-30 (1981), pp. 147–148.

[Bur96] C. Burnikel, *Exact Computation of Voronoi Diagrams and Line Segment Intersections*, Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany, 1996.

[BKM+95] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig, *Exact geometric computation in LEDA*, in Proceedings of the 11th Annual ACM Symposium on Computational Geometry, Vancouver, Canada, 1995, pp. C18–C19.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra, *On degeneracy in geometric computations*, in Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, 1994, pp. 16–23.

[Cha91] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[CE92] B. Chazelle and H. Edelsbrunner, *An optimal algorithm for intersecting line segments in the plane*, J. ACM, 39 (1992), pp. 1–54.

[CEGS94] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, *Algorithms for bichromatic line segment problems and polyhedral terrains*, Algorithmica, 11 (1994), pp. 116–132.

[Cla92] K. L. Clarkson, *Safe and effective determinant evaluation*, in Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 387–395.

[CS89] K. L. Clarkson and P. W. Shor, *Applications of random sampling in computational geometry,* II. Discrete Comput. Geom., 4 (1989), pp. 387–421.

[EG89] H. Edelsbrunner and L. J. Guibas, *Topologically sweeping an arrangement*, J. Comput. System Sci., 38 (1989), pp. 165–194. Corrigendum in 42 (1991), pp. 249–251.

[For85] A. R. Forrest, *Computational geometry in practice*, in Fundamental Algorithms for Computer Graphics, NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci. 17, R. A. Earnshaw, ed., Springer-Verlag, Berlin, 1985, pp. 707–724.

[For87] A. R. Forrest, *Computational geometry and software engineering: Towards a geometric computing environment*, in Techniques for Computer Graphics, D. F. Rogers and R. A. Earnshaw, eds., Springer-Verlag, Berlin, 1987, pp. 23–37.

[For93] S. Fortune, *Progress in computational geometry*, in Directions in Computational Geometry, R. Martin, ed., Information Geometers, Winchester, UK, 1993, pp. 81–128.

[FV93] S. Fortune and C. J. Van Wyk, *Efficient exact arithmetic for computational geometry*, in Proceedings of the 9th Annual ACM Symposium on Computational Geometry, San Diego, CA, 1993, pp. 163–172.

[Gol91] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surv., 23 (1991), pp. 5–48.

[GS87] L. J. Guibas and R. Seidel, *Computing convolutions by reciprocal search*, Discrete Comput. Geom., 2 (1987), pp. 175–193.

[HS90] J. Hershberger and S. Suri, *Applications of a semi-dynamic convex hull algorithm*, In Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (Bergen, 1990), Lecture Notes Comput. Sci. 447, Springer-Verlag, Berlin, 1990, pp. 380–392.

[HS96] J. Hershberger and S. Suri, *Off-line maintenance of planar configurations*, J. Algorithms, 21 (1996), pp. 453–475.

[Hob93] J. Hobby, *Practical Segment Intersection with Finite Precision Output*, Technical Report 93/2-27, Bell Laboratories, Murray Hill, NJ, 1993.

[Hof89] C. M. Hoffmann, *The problems of accuracy and robustness in geometric computation*, IEEE Computer, 22 (1989), pp. 31–41.

[HHK89] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick, *Robust set operations on polyhedral solids*, IEEE Comput. Graph. Appl., 9 (1989), pp. 50–59.

[LPT96] G. Liotta, F. P. Preparata, and R. Tamassia, *Robust proximity queries: An illustration of degree-driven algorithm design*, SIAM J. Comput., 28 (1999), pp. 864–889.

[MS88] H. G. Mairson and J. Stolfi, *Reporting and counting intersections between two sets of line segments*, in Theoretical Foundations of Computer Graphics and CAD, NATO Adv. Sci. Inst. Ser. F. Comput. Systems Sci. 40, R. A. Earnshaw, ed., Springer-Verlag, Berlin, 1988, pp. 307–325.

[Mil88] V. J. Milenkovic, *Verifiable implementations of geometric algorithms using finite precision arithmetic*, Artificial Intelligence, 37 (1988), pp. 377–401.

[Mil89] V. Milenkovic, *Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic*, in Proceedings of the 30th Annual IEEE Symposium on Foundations Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 500–505.

[Mye85] E. W. Myers, *An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem*, SIAM J. Comput., 14 (1985), pp. 625–637.

[NP82] J. Nievergelt and F. P. Preparata, *Plane-sweep algorithms for intersecting geometric figures*, Commun. ACM, 25 (1982), pp. 739–747.

[Pri92] D. Priest, *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, Ph.D. thesis, Dept. of Mathematics, Univ. of California at Berkeley, 1992.

[Sch91] P. Schorn, *Robust Algorithms in a Program Library for Geometric Computation*, volume 32 of Informatik-Dissertationen ETH Zürich, Verlag der Fachvereine, Zürich, Switzerland, 1991.

[She96] J. R. Shewchuk, *Robust adaptive floating-point geometric predicates*, in Proceedings of the 12th Annual ACM Symposium on Computational Geometry, Philadelphia, PA, 1996, pp. 141–150.

[Yap97] C. Yap, *Towards exact geometric computation*, Comput. Geom., 7 (1997), pp. 3–23.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1422–1448

BINARY SPACE PARTITIONS FOR FAT RECTANGLES[*]

PANKAJ K. AGARWAL[†], EDWARD F. GROVE[‡], T. M. MURALI[§], AND JEFFREY SCOTT VITTER[¶]

Abstract. We consider the practical problem of constructing binary space partitions (BSPs) for a set S of n orthogonal, nonintersecting, two-dimensional rectangles in $\mathbb{R}^3$ such that the aspect ratio of each rectangle in S is at most α, for some constant $\alpha \geq 1$. We present an $n2^{O(\sqrt{\log n})}$-time algorithm to build a binary space partition of size $n2^{O(\sqrt{\log n})}$ for S. We also show that if m of the n rectangles in S have aspect ratios greater than α, we can construct a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ for S in $n\sqrt{m}2^{O(\sqrt{\log n})}$ time. The constants of proportionality in the big-oh terms are linear in $\log \alpha$. We extend these results to cases in which the input contains nonorthogonal or intersecting objects.

Key words. binary space partitions, rectangles, aspect ratio, computational geometry, computer graphics, solid modelling

AMS subject classifications. 65Y25, 68P05, 68Q20, 68Q25, 68U05

PII. S0097539797320578

1. Introduction.

Rendering a set of opaque or partially transparent objects in $\mathbb{R}^3$ quickly and in a visually realistic way is a fundamental problem in computer graphics [15]. A central component of this problem is *hidden-surface removal*: given a set of objects, a viewpoint, and an image plane, compute the scene visible from the viewpoint as projected onto the image plane. Because of its importance, the hidden-surface removal problem has been studied extensively in both the computer graphics and the computational geometry communities [14, 15, 28]. One of the conceptually simplest solutions to this problem is the z-buffer algorithm [8, 15]. This algorithm sequentially processes the objects; for each object, it updates the pixels of the image plane covered by the object, based on the distance information stored in the z-buffer. A very fast hidden-surface removal algorithm can be obtained by implementing the z-buffer in hardware. However, only special-purpose and costly graphics engines contain fast z-buffers, and z-buffers implemented in software are generally inefficient. Even

*Received by the editors May 2, 1997; accepted for publication (in revised form) March 30, 1999; published electronically March 15, 2000. A preliminary version of this paper appeared in the *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1996.

http://www.siam.org/journals/sicomp/29-5/32057.html

†Box 90129, Department of Computer Science, Duke University, Durham, NC 27708–0129 (pankaj@cs.duke.edu). Support of this author was provided by National Science Foundation research grant CCR–93–01259, Army Research Office MURI grant DAAH04–96–1–0013, a Sloan fellowship, a National Science Foundation NYI award with matching funds from the Xerox Corporation, and a grant from the U.S.–Israeli Binational Science Foundation.

‡946 Tamarack Lane #13, Sunnyvale, CA 94086 (efg@cs.duke.edu). Support of this author was provided by Army Research Office grant DAAH04–93–G–0076. This work was partially done when the author was at Duke University.

§Computer Science Department, Stanford University, Gates 133, 353 Serra Mall, Stanford, CA 94305 (murali@cs.stanford.edu). Support of this author was provided in part by National Science Foundation research grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013. This author performed this research when he was affiliated with Brown University and was a visiting scholar at Duke University.

¶Box 90129, Department of Computer Science, Duke University, Durham, NC 27708–0129 (jsv@cs.duke.edu). Support of this author was provided in part by National Science Foundation research grant CCR–9522047, by Army Research Office grant DAAH04–93–G–0076, and by Army Research Office MURI grant DAAH04–96–1–0013.

when fast hardware z-buffers are present, they are not fast enough to handle the huge models (containing hundreds of millions of polygons) that often have to be displayed in real time. As a result, other methods have to be developed either to "cull away" a large subset of invisible polygons so as to decrease the rendering load on the graphics pipeline (when models are large; see, e.g., [29]) or to completely solve the hidden-surface removal problem (when there are very slow or no z-buffers).

One technique to handle both of these problems is the *binary space partition* (BSP), a data structure introduced by Fuchs, Kedem, and Naylor [16] that is based on work by Schumacker et al. [27]. Fuchs, Kedem, and Naylor use the BSP to implement the so-called "painter's algorithm" for hidden-surface removal; the painter's algorithm draws the objects to be displayed on the screen in a back-to-front order (in which no object is occluded by any object earlier in the order). In general, it is not possible to find a back-to-front order from a given viewpoint for an arbitrary set of objects. By fragmenting the objects, the BSP ensures that a back-to-front order from *any* viewpoint can be determined for the fragments [16].

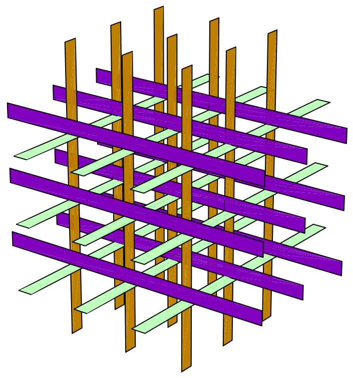
The BSPs have subsequently proven to be versatile, with applications in many other problems—global illumination [6], shadow generation [10, 11], visibility problems [4, 29], solid modeling [22, 24, 30], geometric data repair [19], ray tracing [21], robotics [5], approximation algorithms for network design [17], and surface simplification [3]. Algorithms have also been developed to construct BSPs for moving objects [1, 2, 12, 23, 31].

Informally, a BSP $\mathcal{B}$ for a set of $(d-1)$-dimensional objects in $\mathbb{R}^d$ is a binary tree. Each node v of $\mathcal{B}$ is associated with a convex region $\mathcal{R}_v$. The regions associated with the children of v are obtained by splitting $\mathcal{R}_v$ with a hyperplane. If v is a leaf of $\mathcal{B}$, then the interior of $\mathcal{R}_v$ does not intersect any object. The regions associated with the leaves of the tree form a convex decomposition of $\mathbb{R}^d$. The $(d-1)$-dimensional faces of the cells of this decomposition intersect the objects and divide them into fragments; these fragments are stored at appropriate nodes of the BSP.

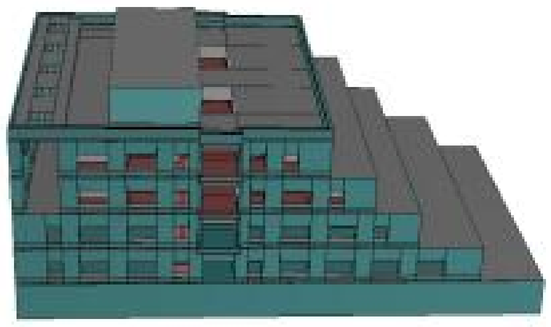
The efficiency of most algorithms that use BSPs depends on the number of nodes in the BSP. As a result, there has been a lot of effort to construct BSPs of small size. Although several simple heuristics have been developed for constructing BSPs of reasonable sizes [4, 7, 16, 20, 29, 30], provable bounds were first obtained by Paterson and Yao. They show that a BSP of size $O(n \log n)$ can be constructed for n disjoint segments in $\mathbb{R}^2$; they also show that a BSP of size $O(n^2)$ can be constructed for n disjoint triangles in $\mathbb{R}^3$, which is optimal in the worst case [25]. But in graphics-related applications, many common environments like buildings are composed largely of orthogonal rectangles, and nonorthogonal objects are approximated by their orthogonal bounding boxes [15]. Paterson and Yao [26] prove that a BSP of size $O(n)$ exists for n nonintersecting, orthogonal segments in $\mathbb{R}^2$, and a BSP of size $O(n\sqrt{n})$ exists for n nonintersecting, orthogonal rectangles in $\mathbb{R}^3$. These bounds are optimal in the worst case.

In all known lower bound examples of orthogonal rectangles in $\mathbb{R}^3$ requiring BSPs of size $\Omega(n\sqrt{n})$, most of the rectangles are "thin." For example, the lower bound proof presented by Paterson and Yao uses a configuration of $\Theta(n)$ orthogonal rectangles, arranged in a $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ grid, for which any BSP has size $\Omega(n\sqrt{n})$ (see Figure 1.1).All rectangles in their construction have aspect ratio $\Omega(\sqrt{n})$. Such configurations of thin rectangles rarely occur in practice. Many real databases consist mainly of "fat" rectangles; i.e., the aspect ratios of these rectangles are bounded by a constant.

It is natural to ask whether BSPs of near-linear size can be constructed if most



(a)



(b)

FIG. 1.1. (a) *Lower bound for orthogonal rectangles.* (b) *Model of a building—*85% *of the rectangles have aspect ratio at most* 25.

of the rectangles are "fat." We call a rectangle *fat* if its aspect ratio (the ratio of the longer side to the shorter side) is bounded by a fixed constant; for specificity, we use $\alpha \geq 1$ to denote this constant. A rectangle is said to be *thin* if its aspect ratio is greater than α. In this paper, we consider the following problem:

> *Given a set S of n nonintersecting, orthogonal, two-dimensional rectangles in $\mathbb{R}^3$, of which m are thin and the remaining $n-m$ are fat, construct a BSP for S.*

We first show how to construct a BSP of size $n2^{O(\sqrt{\log n})}$ for n fat rectangles in $\mathbb{R}^3$ (i.e., when $m = 0$). We then show that if $m > 0$, a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ can be built. We also prove a lower bound of $\Omega(n\sqrt{m})$ on the size of such a BSP.

We finally prove two important extensions to these results. If p of the n input objects are nonorthogonal, we show that an $np2^{O(\sqrt{\log n})}$-size BSP exists. Unlike in the case of orthogonal objects, fatness does not help in reducing the worst-case size of BSPs for nonorthogonal objects. In particular, we prove that there exists a set of n fat triangles in $\mathbb{R}^3$ for which any BSP has $\Omega(n^2)$ size. However, nonorthogonal objects can be approximated by orthogonal bounding boxes. The resulting bounding boxes might intersect each other. Motivated by this observation, we also consider the problem in which n fat rectangles contain k intersecting pairs of rectangles, and we show that we can construct a BSP of size $(n+k)\sqrt{k}2^{O(\sqrt{\log n})}$.

In all cases, the constant of proportionality in the big-oh terms is linear in $\log \alpha$, where α is the maximum aspect ratio of the fat rectangles. Our algorithms to construct these BSPs run in time proportional to the size of the BSPs they build. Experiments demonstrate that our algorithms work well in practice and construct BSPs of near-linear size when most of the rectangles are fat, and perform better than most known algorithms for constructing BSPs for orthogonal rectangles [18].

As far as we are aware, ours is the first work to consider BSPs for the practical and common case of (two-dimensional) fat polygons in $\mathbb{R}^3$. De Berg considers a weaker model, the case of (three-dimensional) fat polyhedra in $\mathbb{R}^3$ (a polyhedron is said to be *fat* if its volume is at least a constant fraction of the volume of the smallest sphere enclosing it), although his results extend to higher dimensions [13].

One of the main ingredients of our algorithm is the construction of an $O(n \log n)$-size BSP for a set of n fat rectangles that are "long" with respect to a box B; i.e., none of the vertices of the rectangles lie in the interior of B. To prove this result, we crucially use the fatness of the rectangles. We then develop an algorithm to construct a BSP of size $n2^{O(\sqrt{\log n})}$ for n fat rectangles by simultaneously simulating the algorithm for long rectangles and partitioning the vertices of rectangles in S in a clever manner.

The rest of the paper is organized as follows: section 2 gives some preliminary definitions. In section 3, we show how to build an $O(n \log n)$-size BSP for n long rectangles. Sections 4 and 5 present and analyze our algorithm to construct a BSP of size $n2^{O(\sqrt{\log n})}$ for n fat rectangles. We extend this result in section 6 to construct BSPs for cases in which some objects in the input are thin or nonorthogonal. We conclude in section 7 with some open problems.

2. Geometric preliminaries. A *binary space partition* $\mathcal{B}$ for a set S of pairwise-disjoint, $(d-1)$-dimensional, polyhedral objects in $\mathbb{R}^d$ is a tree defined as follows: Each node v in $\mathcal{B}$ represents a convex polytope $\mathcal{R}_v$ and a set of objects $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The region associated with the root is $\mathbb{R}^d$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$. Otherwise, we partition $\mathcal{R}_v$ into two convex polytopes by a *cutting hyperplane* H_v. At v, we store the equation of H_v and $\{s \mid s \in S_v, s \subseteq H_v\}$, the subset of objects in S_v that lie in H_v. If we let H_v^+ be the positive halfspace and H_v^- the negative halfspace bounded by H_v, the polytopes associated with the left and right children of v are $\mathcal{R}_v \cap H_v^-$ and $\mathcal{R}_v \cap H_v^+$, respectively. The left subtree of v is a BSP for the set of objects $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of objects $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of nodes in $\mathcal{B}$ and the total number of faces of all dimensions of the objects stored at all the nodes in $\mathcal{B}$.

In our case, S is a set of orthogonal rectangles in $\mathbb{R}^3$. In our algorithms, we will use orthogonal cutting planes. Therefore, the region $\mathcal{R}_v$ associated with each node v in $\mathcal{B}$ is a *box* (rectangular parallelepiped). We say that a rectangle r is *long* with respect to a box B if none of the vertices of r lie in the interior of B. Otherwise, r is said to be *short* (see Figure 2.1). A long rectangle is *free* if none of its edges lies in the interior of B; otherwise it is *nonfree*. A *free cut* is a cutting plane that does not cross any rectangle in S and that either divides S into two nonempty sets or contains

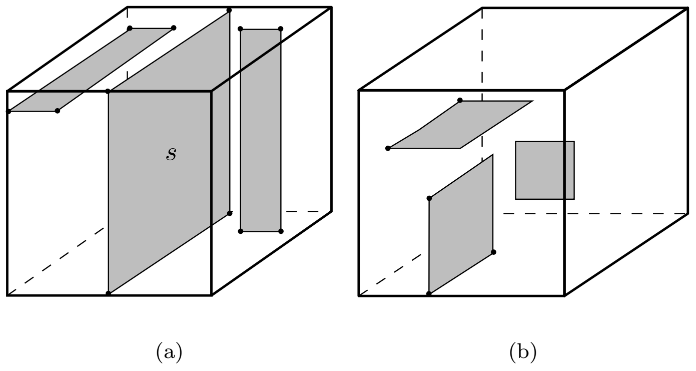


FIG. 2.1. (a) *Long and* (b) *short rectangles. Heavy dots indicate the vertices of these rectangles that lie on the boundary of the box. Rectangle s is a free rectangle.*

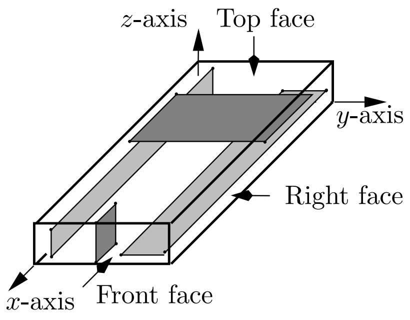


FIG. 2.2. *Different classes of rectangles.*

a rectangle in S. Note that the plane containing a free rectangle is a free cut. Free cuts play a critical role in preventing excessive fragmentation of the rectangles in S.

We will often focus on a box B and construct a BSP for the rectangles intersecting it. Given a set of rectangles R, let

$$R_B = \{s \cap B \mid s \in R\}$$

be the set of rectangles obtained by clipping the rectangles in R within B. For a set of points P, let P_B be the subset of P lying in the interior of B.

A box B has six faces: *top, bottom, front, back, right,* and *left*, as shown in Figure 2.2. We assume, without loss of generality, that the back, bottom, left corner of B is the origin (i.e., the back face of B lies on the yz-plane). A rectangle s that is long with respect to B belongs to the *top class* if two parallel edges of $s \cap B$ are contained in the top and bottom faces of B. We similarly define the *front* and *right* classes. A long rectangle belongs to at least one of these three classes; a nonfree long rectangle belongs to a unique class. See Figure 2.2 for examples of rectangles belonging to different classes.

Although a BSP is a tree, we will often discuss just how to partition the box represented by a node into two boxes. We will not explicitly detail the associated construction of the actual tree itself, since the construction is straightforward once we specify the cutting plane. Sometimes we will abuse notation and use B to refer to

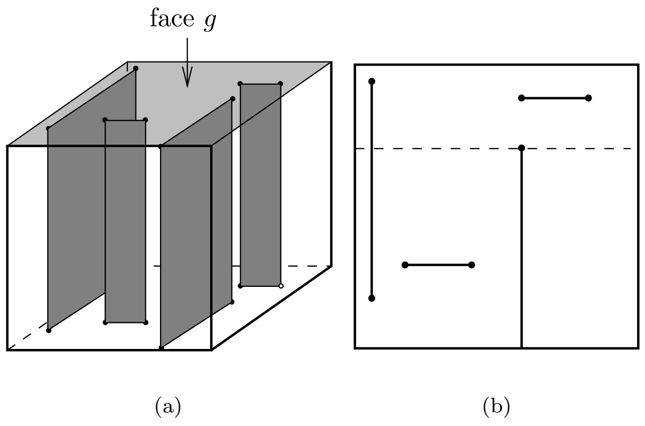


FIG. 2.3. (a) *Long rectangles in the top class.* (b) *Projections of the rectangles in* (a) *onto the top face g; heavy dots indicate the vertices of these rectangles that lie in the interior of g. The dashed line is the cut satisfying* (2.1).

the corresponding node in the BSP as well.

In the rest of the paper, we assume that the vertices of the rectangles in S are sorted by x-, y-, and z-coordinates, and that for each axis, the rectangles perpendicular to that axis are sorted by intercept. The cost of this sort will not affect the asymptotic running times of our algorithms.

We now state two preliminary lemmas that we will use in our algorithms. The first lemma characterizes a set of rectangles that are long with respect to a box and belong to one class. The second lemma applies to two classes of long rectangles.

LEMMA 2.1. *Let C be a box, P a set of points in the interior of C, R a set of rectangles long with respect to C, and $w \geq 1$ a real number. If the rectangles in R_C belong to one class, then the following two conditions hold (see Figure* 2.3*):*

(i) *There exists a face g of the box C that contains exactly one of the edges of each rectangle in R_C. Let V be the set of those vertices of the rectangles in R_C that lie in the interior of g.*

(ii) *We can find a plane that partitions C into two boxes C_1 and C_2 so that for $i = 1, 2$,*

$$|V \cap C_i| + w|P_{C_i}| \leq \frac{|V| + w|P|}{2}. \tag{2.1}$$

If the rectangles in R_C and the points in P are sorted along each of the three axes, the partitioning plane can be computed in $O(|R_C| + |P|)$ time.

Proof. (i) follows from the definition of a class. To prove part (ii) of the lemma, let P^* be the set of projections of the points in P onto g. Assume g is the top face of C. If we associate a weight of 1 with each point in V and a weight w with each point in P^*, the total weight of the points in $V \cup P^*$ is $|V| + w|P|$. By sweeping g, we can find a line l lying in g and parallel to the x-axis that contains a point in $V \cup P^*$ and divides $V \cup P^*$ into two sets, each with weight at most $(|V| + w|P|)/2$. We split C into two boxes C_1 and C_2 using the plane containing l that is orthogonal to g. By construction, C_1 and C_2 satisfy (2.1). The time bound follows easily. ∎

LEMMA 2.2. *Let C be a box, P a set of points in the interior of C, R a set of* *rectangles long with respect to C, and $w \geq 1$ a real number. If the rectangles in R_C belong to two classes, then one of the following two conditions holds (see Figure 2.4):*

(i) *We can find one free cut that partitions C into two boxes C_1 and C_2 so that*

$$|R_{C_i}| + w|P_{C_i}| \leq \frac{2\left(|R_C| + w|P|\right)}{3} \tag{2.2}$$

for $i = 1, 2$.

(ii) *We can find two parallel free cuts that divide C into three boxes C_1, C_2, and C_3 such that all rectangles in R_{C_2} belong to the same class and such that*

$$|R_{C_2}| + w|P_{C_2}| \geq \frac{|R_C| + w|P|}{3}. \tag{2.3}$$

If the rectangles in R_C and the points in P are sorted along each of the three axes, these free cuts can be computed in $O(|R_C| + |P|)$ time.

Proof. We assume without loss of generality that the rectangles in R_C belong to the top and right classes. Let $\bar{r}$ denote the projection of a rectangle $r \in R_C$ onto the x-axis: $\bar{r}$ is either a point or an interval. Similarly, let $\bar{p}$ denote the projection of a point $p \in P$ onto the x-axis. Set

$$U = \left\{ \left(\bigcup_{r \in R_C} \bar{r} \right) \cup \left(\bigcup_{p \in P} \bar{p} \right) \right\}.$$

The set U is a collection of disjoint intervals, some of which may be single points. Let r_1 (resp., r_2) be a rectangle in R_C belonging to the top (resp., right) class. Since the rectangles in R_C are disjoint, it is easily seen that $\bar{r}_1$ and $\bar{r}_2$ are also disjoint. Hence, each connected component of U contains the projections of rectangles belonging to at most one class. For any connected component I of U define

$$\mu(I) = |\{r \in R_C \mid \bar{r} \subseteq I\}| + w|\{p \in P \mid \bar{p} \subseteq I\}|.$$

Set $W = |R_C| + w|P|$. If U contains a connected component $I = [\beta, \gamma]$ with $\mu(I) > W/3$, then the two free cuts are $x = \beta$ and $x = \gamma$. The cuts partition the box C into three boxes C_1, C_2, and C_3, where C_2 denotes the middle box. By construction, all rectangles in R_{C_2} belong to at most one class. Hence, condition (ii) holds. [1]

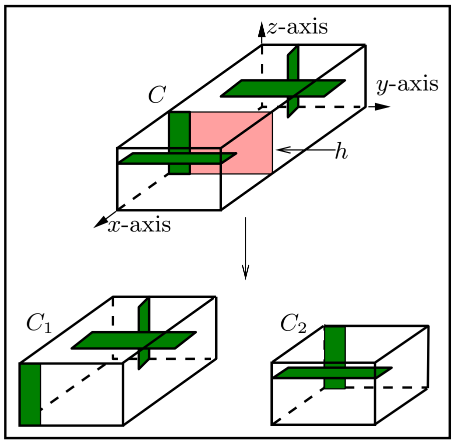
If there is no such connected component of U, then let $I = [\beta, \gamma]$ be the leftmost connected component of U with $\sum_{I' \leq I} \mu(I') > W/3$ (we say that $I' \leq I$ if I' lies to the left of I). Since $\mu(I) \leq W/3$ and $\sum_{I' < I} \mu(I') < W/3$,

$$\sum_{I' \leq I} \mu(I') \leq 2W/3.$$

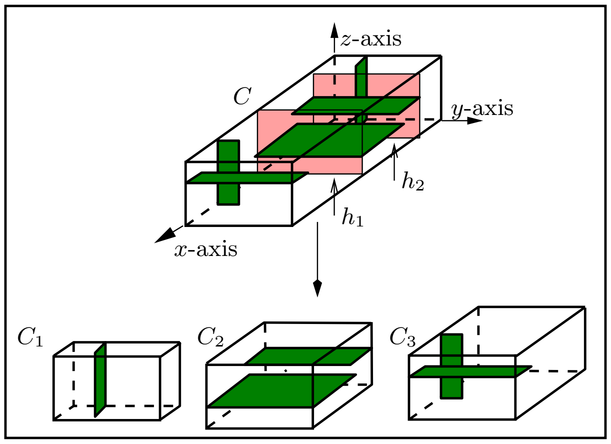
We partition C into two boxes C_1 and C_2 using the cut $x = \gamma$. In this case, condition (i) holds.

If the rectangles in R_C and the points in P are sorted along the x-axis, it is clear that the components in U can be computed and sorted in $O(|R_C| + |P|)$ time. The free cut(s) used to partition C can be found in the same time by sweeping the components of U. ◻

[1]If $\beta = \gamma$, i.e., I is a point, then C_2 is regarded as a degenerate box. If I is the first (resp., last) connected component of U, then C_1 (resp., C_3) may be a degenerate box.



(a)



(b)

FIG. 2.4. (a) *Free cut h partitions box C into two boxes C_1 and C_2.* (b) *Two parallel free cuts h_1 and h_2 partition C into three boxes: C_1, C_2, and C_3.*

3. BSPs for long fat rectangles. Let S be a set of fat rectangles. Assume that all the rectangles in S are long with respect to a box B. In this section, we show how to build a BSP for S_B, the set of rectangles clipped within B. In general, S_B can have all three classes of rectangles. We first exploit the fatness of the rectangles to prove that whenever all three classes are present in S_B, a small number of cuts can divide B into boxes, each of which has only two classes of rectangles. Then we describe an algorithm that constructs a BSP for rectangles belonging to only two classes.

3.1. Reducing three classes to two classes. Assume, without loss of generality, that the longest edge of B is parallel to the x-axis. The rectangles in S_B that

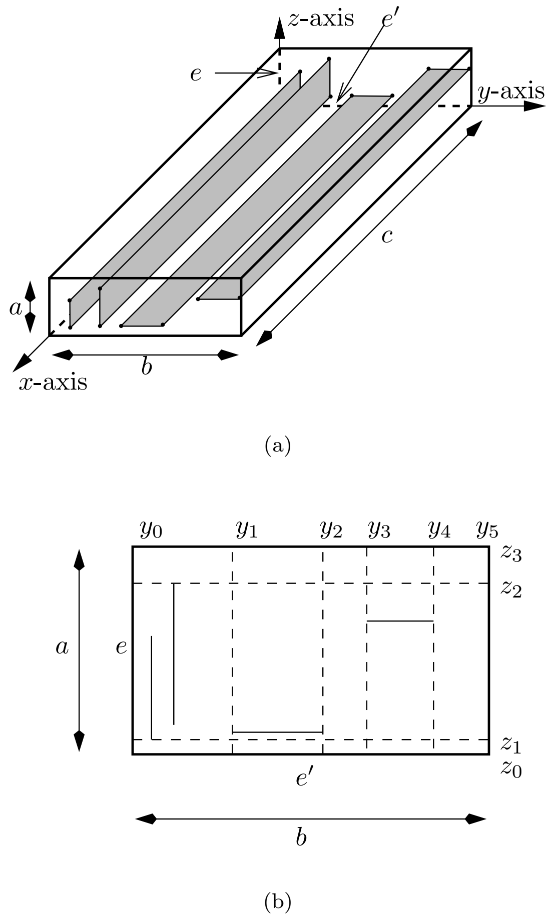


FIG. 3.1. (a) *Rectangles belonging to the sets R and T.* (b) *The back face of B; dashed lines are intersections of the back face with the α-cuts.*

belong to the front class can be partitioned into two subsets: the set R of rectangles that are vertical (and parallel to the right face of B) and the set T of rectangles that are horizontal (and parallel to the top face of B); see Figure 3.1. Let e be the edge of B that lies on the z-axis, and let e' be the edge of B that lies on the y-axis. The intersection of each rectangle in R with the back face of B is a segment parallel to the z-axis. Let $\bar{r}$ denote the projection of such a segment r onto the z-axis, and let $\bar{R} = \{\bar{r} \mid r \in R\}$. Let $z_1 < z_2 < \cdots < z_{k-1}$ be the endpoints of intervals in $\bar{R}$ that lie in the interior of e but not in the interior of any interval of $\bar{R}$. Note that $k-1$ may be less than $2|R|$, as in Figure 3.1, if some of the projected segments overlap. If no two intervals in $\bar{R}$ share an endpoint, then $\{z_1, z_2, \ldots, z_{k-1}\}$ is the set of vertices of the union of the intervals in $\bar{R}$; otherwise, $\{z_1, z_2, \ldots, z_{k-1}\}$ includes endpoints common to two intervals in $\bar{R}$ and not lying in the interior of any other interval in $\bar{R}$. Similarly, for each rectangle t in the set T, let $\bar{t}$ be the projection of t onto the y-axis, and let $\bar{T} = \{\bar{t} \mid t \in T\}$. Let $y_1 < y_2 < \cdots < y_{l-1}$ be the endpoints of intervals in $\bar{T}$ that lie in the interior of e' but not in the interior of any interval of $\bar{T}$.

We divide B into kl boxes by drawing the planes $z = z_i$ for $1 \leq i < k$ and the planes $y = y_j$ for $1 \leq j < l$; see Figure 3.1. This decomposition of B into kl boxes can easily be constructed in a treelike fashion by performing $(k-1)(l-1)$ cuts. We refer to these cuts as *α-cuts*. If any resulting box has a free rectangle, we divide that box into two boxes by applying the free cut along the free rectangle. Let $\mathcal{C}$ be the set of boxes into which B is partitioned in this manner. We can prove the following lemma about the decomposition of B into $\mathcal{C}$.

LEMMA 3.1. *The set $\mathcal{C}$ of boxes formed by the above process satisfies the following properties:*

(i) *Each box C in $\mathcal{C}$ has only two classes of rectangles,*
(ii) *there are at most $26\alpha^2 n$ boxes in $\mathcal{C}$, and*
(iii) $\sum_{C \in \mathcal{C}} |S_C| \leq 16\alpha n$.

Proof. Let z_0 and z_k, where $z_0 < z_k$, be the endpoints of e, the edge of the box B that lies on the z-axis. Similarly, define y_0 and y_l, where $y_0 < y_l$, to be the endpoints of the edge of B that lies on the y-axis.

(i) Let C be a box in $\mathcal{C}$. If C does not contain a rectangle from $T \cup R$, the claim is obvious since the rectangles in T and R together constitute the front class. Suppose C contains rectangles from the set R. Rectangles in R belong to the front class and are parallel to the right face of B. We claim that C cannot have any rectangles from the right class. Indeed, consider an edge of C parallel to the z-axis. The endpoints of this edge have z-coordinates z_i and z_{i+1}, for some $0 \leq i < k$. Since C contains a rectangle from R, by construction, the interval $z_i z_{i+1}$ must be covered by projections of rectangles in R (onto the z-axis). If C also contains a rectangle r belonging to the right class, then let $z_i < z < z_{i+1}$ be the z-coordinate of a point in $r \cap C$. Let r' be a rectangle in R whose projection on the z-axis contains z. Since both r and r' are long with respect to C, the interiors of r and r' intersect, which contradicts the fact that the rectangles in S are nonintersecting. A similar proof shows that if C contains rectangles from T, then C does not contain any rectangle in the top class.

(ii) We first show that both k and l are at most $2\lfloor\alpha\rfloor + 3$. Let a (resp., b, c) denote the length of the edges of B parallel to the z-axis (resp., y-axis, x-axis). By assumption, $a, b \leq c$. Let $r \in R$ be a rectangle with dimensions β and γ, where $\beta \leq \gamma$. Consider $\bar{r}$, the projection of r onto the z-axis. Suppose that $\bar{r} \subseteq z_i z_{i+1}$, for some $0 < i < k-1$, i.e., $\bar{r}$ lies in the interior of the edge e of B lying on the z-axis. Since r is a rectangle in the front class and is parallel to the right face of B, we have $\beta \leq a \leq c = \gamma$. If $\hat{r}$, the rectangle supporting r in the set S, has dimensions $\hat{\beta}$ and $\hat{\gamma}$, where $\hat{\beta} \leq \hat{\gamma} \leq \alpha\hat{\beta}$, we have $\beta = \hat{\beta}$ (since $\bar{r} \subseteq \mathrm{int}(e)$) and $\gamma \leq \hat{\gamma}$. (If i is 0 or $k-1$, we cannot claim that $\beta = \hat{\beta}$; in these cases, it is possible that $\beta \ll \hat{\beta}$.) See Figure 3.2. Thus, we obtain

$$a \leq c = \gamma \leq \hat{\gamma} \leq \alpha\hat{\beta} = \alpha\beta.$$

It follows that the length of the interval $\bar{r}$, and hence the length of $z_i z_{i+1}$, is at least a/α. Since every alternate interval $z_i z_{i+1}, 0 < i < k-1$ contains the projection of at least one rectangle of R, we have $k \leq 2\lfloor\alpha\rfloor + 3$. In a similar manner, $l \leq 2\lfloor\alpha\rfloor + 3$. Hence, the planes $z = z_i, 1 \leq i \leq k-1$ and the planes $y = y_j, 1 \leq j \leq l-1$ partition B into at most $kl \leq (2\lfloor\alpha\rfloor + 3)^2$ boxes. Each such box C can contain at most n rectangles. Hence, at most n free cuts can be made inside C. The free cuts can divide C into at most $n+1$ boxes. This implies that the set $\mathcal{C}$ has at most $kl(n+1) \leq 26\alpha^2 n$ boxes.

(iii) Each rectangle r in S_B is cut into at most kl pieces. The edges of these pieces

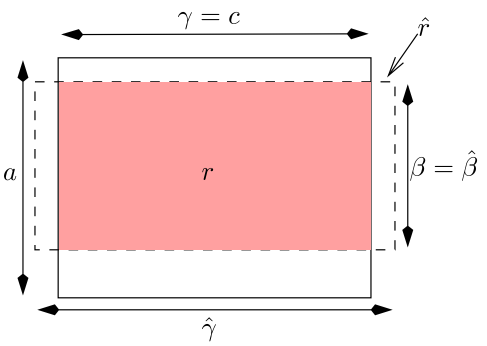


FIG. 3.2. *Projections of $\hat{r}$ (the dashed rectangle), $r = \hat{r} \cap B$ (the shaded rectangle), and the right face of B onto the zx-plane.*

form an arrangement on r. Each face of the arrangement is one of the at most kl rectangles that r is partitioned into. Only $2(k + l - 2)$ faces of the arrangement have an edge on the boundary of r. All other faces can be used as free cuts. Hence, after all possible free cuts are made in the boxes into which B is divided by the $(k-1)(l-1)$ cuts, only $2(k + l - 2)$ pieces of each rectangle in S_B survive. This proves that $\sum_{C \in \mathcal{C}} |S_C| \leq 16\alpha n$. ◻

Remark. The only place in the whole algorithm where we use the fatness of the rectangles in S is in the proof of Lemma 3.1. If the rectangles in S are thin, then Lemma 3.1(ii) is not true; both k and l can be $\Omega(n)$.

If S_B contains a short rectangle, the α-cuts partition the short rectangle into a constant number of pieces. Hence, Lemma 3.1(ii) and 3.1(iii) hold even when S_B contains short rectangles.

3.2. BSPs for two classes of long rectangles. Let C be one of the boxes into which B is partitioned by the α-cuts. We now present an algorithm for constructing a BSP for the set of clipped rectangles S_C, which has only two classes of long rectangles. We recursively apply the following steps to each of the boxes produced by the algorithm until no box contains a rectangle.

1. If S_C has a free rectangle, we use the free cut containing that rectangle to split C into two boxes.
2. If S_C has two classes of rectangles, we use Lemma 2.2 (with $R = S$ and $P = \emptyset$) to split C into at most three boxes, using at most two parallel free cuts.
3. If S_C has only one class of rectangles, we split C into two by a plane, using Lemma 2.1 (with $R = S$ and $P = \emptyset$).

In steps 2 and 3, since $P = \emptyset$, we can use any value of w in Lemmas 2.2 and 2.1.

We first analyze the algorithm for two classes of long rectangles. The BSP produced has the following structure: If step 3 is executed at a node v, then step 2 is not invoked at any descendant of v. Note that the cutting planes used in steps 1 and 2 do not intersect any rectangle of S_C, so only the cuts made in step 3 increase the number of rectangles. Hence, repeated execution of steps 1 or 2 on S_C constructs a top subtree $\mathcal{T}_C$ of the BSP with $O(|S_C|)$ nodes such that each leaf in $\mathcal{T}_C$ has only one class of rectangles and the total number of rectangles in all the leaves is at most $|S_C|$. The operations at a box D in $\mathcal{T}_C$ involve determining the cuts to be made at D, partitioning D according to these cuts, identifying the resulting free rectangles and applying free cuts containing them, and partitioning the rectangles in S_D into the new boxes.

Since we assume that we have sorted the vertices of the rectangles in S at the very beginning, Lemmas 2.1 and 2.2 imply that the cuts to be made at B can be determined in $O(|S_D|)$ time. The free cuts resulting after partitioning D according to these cuts are parallel to each other. Hence, all free cuts can be applied in $O(|S_D|)$ time. Further, the number of rectangles at each child of D is at most $2|S_D|/3$. Hence, $\mathcal{T}_C$ can be constructed in $O(|S_C|\log|S_C|)$ time. At each leaf v of the tree $\mathcal{T}_C$, recursive invocations of steps 1 and 3 build a BSP of size $O(|S_v|\log|S_v|)$ in $O(|S_v|\log|S_v|)$ time (see [25] for details). Since $\sum_v S_v \le |S_C|$, where the sum is taken over all leaves v of $\mathcal{T}_C$, the total size of the BSP constructed inside C is $O(|S_C|\log|S_C|)$. It also follows that the BSP inside C can be constructed in $O(|S_C|\log|S_C|)$ time.

We now analyze the overall algorithm for long rectangles. The algorithm first applies the α-cuts to the rectangles in S_B, as described in section 3.1. Consider the set of boxes $\mathcal{C}$ produced by the α-cuts. Each of the boxes in $\mathcal{C}$ contains only two classes of rectangles (by Lemma 3.1(i)). In view of the above discussion, for each box $C \in \mathcal{C}$, we can construct a BSP for S_C of size $O(|S_C|\log|S_C|)$ in time $O(|S_C|\log|S_C|)$. Lemma 3.1(ii) and (iii) imply that the total size of the BSP is

$$O(n)+\sum_{C\in\mathcal{C}} O(|S_C|\log|S_C|) = O(n\log n).$$

The time spent in building the BSP is also $O(n\log n)$. We can now state the following theorem.

THEOREM 3.2. *Let S be a set of n fat rectangles and B a box so that all rectangles in S are long with respect to B. An $O(n\log n)$-size BSP for the clipped rectangles S_B can be constructed in $O(n\log n)$ time. The constants of proportionality in the big-oh terms are linear in α^2, where α is the maximum aspect ratio of the rectangles in S.*

Remark. We can show that the height of the BSP constructed by the above algorithm is $O(\log n)$. We can also modify our algorithm to construct a BSP of size $O(n)$ for n long rectangles as follows: If a box C has two classes of long rectangles, we apply step 1 or 2 of the previous algorithm. If the rectangles in C belong to one class, we use the algorithm of Paterson and Yao for constructing BSPs for orthogonal segments in the plane [26] to construct a BSP of linear size for S_C. However, the height of the BSP can now be $\Omega(n)$ in the worst case.

4. BSPs for fat rectangles.

We now describe our main algorithm for constructing a BSP for a set S of n fat nonintersecting rectangles, in which we simultaneously simulate the algorithm for long fat rectangles presented in section 3 and partition the vertices of the rectangles in S. The algorithm proceeds in rounds. Each round simulates a few steps of the algorithm for long rectangles and partitions the vertices of the rectangles in S into a small number of sets of approximately equal size. At the beginning of the ith round, for $i > 0$, the algorithm has a top subtree $\mathcal{B}_i$ of the BSP for S. Let Q_i be the set of boxes associated with the leaves of $\mathcal{B}_i$ containing at least one rectangle. The initial tree $\mathcal{B}_1$ consists of one node and Q_1 consists of one box that contains all the input rectangles. Our algorithm maintains the invariant that for each box $B \in Q_i$, all long rectangles in S_B are nonfree. If Q_i is empty, we are done. Otherwise, in the ith round, for each box $B \in Q_i$, we construct a top subtree $\mathcal{T}_B$ of the BSP for the set S_B and attach it to the corresponding leaf of $\mathcal{B}_i$. This gives us the new top subtree $\mathcal{B}_{i+1}$. Thus, it suffices to describe how to build the tree $\mathcal{T}_B$ on a box B during a round.

Let $F \subseteq S_B$ be the set of rectangles that are long with respect to B. Set $f = |F|$, and let k be the number of vertices of rectangles in S_B that lie in the interior of B

(note that each such vertex is a vertex of an original rectangle in the input set S). By assumption, all rectangles in F are nonfree. We choose a parameter a, which remains fixed throughout the round. We pick

$$a = 2^{\sqrt{\log(f+k)}}$$

to optimize the size of the BSP that the algorithm creates. We now describe the ith round in detail. See Figure 4.1 for an outline of $\mathcal{B}$'s structure.

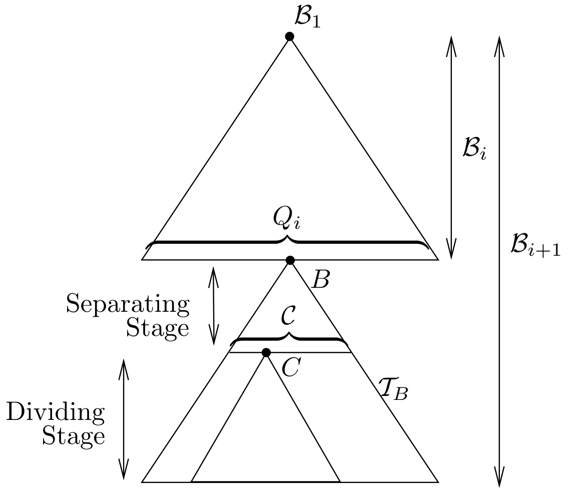


FIG. 4.1. *Overall structure of $\mathcal{B}$.*

If $k = 0$ (i.e., if all rectangles in S_B are long), we use Theorem 3.2 to construct a BSP for S_B. Otherwise, we perform a sequence of cuts in two stages that partition B as follows:

Separating stage. We apply the α-cuts, as described in section 3.1. We make these cuts with respect to the rectangles in F, i.e., we consider only those rectangles of S_B that are long with respect to B. Let $\mathcal{C}$ be the set of boxes into which B is partitioned by the α-cuts.

Dividing stage. We refine each box C in $\mathcal{C}$ by applying cuts similar to the ones made in section 3.2, as described below. Let k_C denote the number of vertices of rectangles in S_C that lie in the interior of C. Recall that F_C is the set of rectangles in F that are clipped within C. We recursively invoke the dividing stage until $|F_C| + 2ak_C \leq (f + ak)/a$ and S_C does not contain any free rectangles.

1. If C has any free rectangle, we use the free cut containing that rectangle to split C into two boxes.
2. If the rectangles in F_C belong to two classes, let P_C denote the set of vertices of the rectangles in S_C that lie in the interior of C. We apply at most two parallel free cuts that satisfy Lemma 2.2, with $R = F, P = P_C$, and $w = 2a$.
3. If the rectangles in F_C belong to just one class, we apply one cut using Lemma 2.1, with $R = F, P = P_C$, and $w = 2a$.

The cuts introduced during the dividing stage can be made in a treelike fashion. At the end of the dividing stage, we have a set of boxes so that for each box D in this set, S_D does not contain any free rectangle and $|F_D| + ak_D \leq (f + ak)/a$. Notice that as we apply cuts in C and in the resulting boxes, rectangles that are short with respect to C may become long with respect to the new boxes. We ignore these new long rectangles until the next round, unless they induce a free cut.

5. Analysis of the algorithm.

We now analyze the size of the BSP constructed by the algorithm and the time complexity of the algorithm. In a round, the algorithm constructs a top subtree $\mathcal{T}_B$ of the BSP for the set of clipped rectangles S_B. Recall that F is the set of rectangles that are long with respect to B, $f = |F|$, and k is the number of vertices of rectangles in S_B that lie in the interior of B. For a node C in $\mathcal{T}_B$, recall that k_C denotes the number of vertices of rectangles in S_C that lie in the interior of C.

We now define some more notation that we need for the analysis. For a node C in $\mathcal{T}_B$, let $\mathcal{T}_C$ be the subtree of $\mathcal{T}_B$ rooted at C, L_C be the set of leaves in $\mathcal{T}_C$, ϕ_C be the number of long rectangles in F_C (recall that F_C is the set of rectangles in F that intersect C and are clipped within C), and ν_C be the number of long rectangles in $S_C \setminus F_C$ (recall that a rectangle in $S_C \setminus F_C$ is a portion of a rectangle in S_B that is short with respect to B). For a box D corresponding to a leaf of $\mathcal{T}_B$, let f_D be the number of long rectangles in S_D. Note that f_D counts both the "old" long rectangles in F_D (pieces of rectangles that were long with respect to B) and the "new" long rectangles in $S_D \setminus F_D$ (pieces of rectangles that were short with respect to B, but became long with respect to D due to the cuts made during the round); $f_D = \phi_D + \nu_D$.

In a round, the separating stage first splits B into a set of boxes $\mathcal{C}$. For each box $C \in \mathcal{C}$, F_C has only two classes of long rectangles. The algorithm then executes the dividing stage on each such box C. As in the case of the algorithm for long rectangles (see section 3), the subtree constructed in C has the following property: if step 4 is executed at a node v, then step 4 is not executed at any descendent of v. In Lemma 5.1, we prove a bound on the total number of long rectangles at each leaf of $\mathcal{T}_B$. For a box $C \in \mathcal{C}$, we bound the number of long rectangles at the leaves of $\mathcal{T}_C$ in Lemmas 5.2 and 5.3. In Lemma 5.4, we prove a bound on the size of the tree $\mathcal{T}_B$. Finally, we use these lemmas to establish bounds on the size of the BSP constructed by our algorithm and the running time of our algorithm (see Theorem 5.5).

LEMMA 5.1. *For a box D associated with a leaf of $\mathcal{T}_B$,*

$$f_D + 2ak_D \leq \frac{f + 2ak}{a}.$$

Proof. We know that ν_D is at most k (since a rectangle in $S_D \setminus F_D$ must be a piece of a rectangle short with respect to B, and there are at most k short rectangles in B). Since $f_D + 2ak_D \leq \phi_D + 2ak_D + \nu_D$ and $\phi_D + 2ak_D \leq (f + ak)/a$ (by construction), the lemma follows. □

LEMMA 5.2. *Let C be a box associated with a node in $\mathcal{T}_B$. If all rectangles in F_C belong to one class, then*

$$\sum_{D \in L_C} f_D \leq 2\phi_C + 2\nu_C \max\left\{\frac{2(\phi_C + ak_C)}{\mu}, 1\right\} + 4k_C\left(\frac{\phi_C + ak_C}{\mu}\right),$$

where $\mu = (f + ak)/a$.

Proof. Assume, without loss of generality, that all rectangles in F_C belong to the top class, and let g be the top face of C. By Lemma 2.1(i), g contains an edge of every rectangle in F_C. Let ρ_C be the number of vertices of the nonfree long rectangles in F_C that lie in the interior of g; obviously, $\phi_C \leq \rho_C \leq 2\phi_C$. Set

$$\Phi(\rho_C, \nu_C, k_C) = \max \sum_{D \in L_C} f_D,$$

where the maximum is taken over all boxes C and over all sets S of rectangles with ρ_C vertices of rectangles in F_C lying in the interior of the top face of C, ν_C long rectangles in $S_C \setminus F_C$, and k_C vertices in the interior of C. We claim that

$$\Phi(\rho_C, \nu_C, k_C) \leq \rho_C + 2\nu_C \max\left\{\frac{\rho_C + 2ak_C}{\mu}, 1\right\} + 2k_C \left(\frac{\rho_C + 2ak_C}{\mu}\right), \tag{5.1}$$

which implies the lemma, because $\rho_C \leq 2\phi_C$.

Note that if S_C contains $m \geq 1$ free rectangles, we apply the free cuts containing these rectangles to partition C (by repeatedly invoking step 4 of the dividing stage) until the resulting boxes do not contain any free rectangle. The free cuts partition C into a set $\mathcal{E}$ of $m+1$ boxes. Since we have created the boxes in $\mathcal{E}$ using free cuts,

$$\rho_E + 2ak_E \leq \rho_C + 2ak_C, \qquad \text{for any box } E \text{ in } \mathcal{E}, \tag{5.2}$$

$$\sum_{E \in \mathcal{E}} \nu_E \leq \nu_C, \qquad \sum_{E \in \mathcal{E}} k_E \leq k_C, \qquad \sum_{E \in \mathcal{E}} \rho_E \leq \rho_C. \tag{5.3}$$

These inequalities imply that if (5.1) holds for each box in $\mathcal{E}$, then (5.1) holds for C as well. Therefore, we prove (5.1) for all boxes C such that F_C contains only one class of rectangles and S_C does not contain any free rectangle. We proceed by induction on $\rho_C + 2ak_C$.

Base case. $0 \leq \rho_C + 2ak_C \leq \mu$. Since $0 \leq \rho_C + 2ak_C \leq \mu$ and S_C does not contain any free rectangle, C is a leaf of $\mathcal{T}_B$, i.e., $L_C = \{C\}$. We have

$$\Phi(\rho_C, \nu_C, k_C) = \sum_{D \in L_C} f_D = f_C = \phi_C + \nu_C \leq \rho_C + \nu_C, \tag{5.4}$$

which implies (5.1).

Induction step. $\rho_C + 2ak_C > \mu$. In this case, C is split into two subboxes C_1 and C_2 by a cutting plane h. Since $\sum_{D \in L_C} f_D = \sum_{D \in L_{C_1}} f_D + \sum_{D \in L_{C_2}} f_D$,

$$\Phi(\rho_C, \nu_C, k_C) = \Phi(\rho_{C_1}, \nu_{C_1}, k_{C_1}) + \Phi(\rho_{C_2}, \nu_{C_2}, k_{C_2}),$$

where $k_{C_1} + k_{C_2} \leq k_C$ and $\rho_{C_1} + \rho_{C_2} \leq \rho_C$.

Note that h does not contain a free rectangle. For $i = 1, 2$, each long rectangle in $S_{C_i} \setminus F_{C_i}$ is contained either in a long rectangle in $S_C \setminus F_C$ or in a short rectangle in S_C. Since h intersects each rectangle in S_C at most once and a short rectangle intersected by h is divided into one short and one long rectangle,

$$\nu_{C_1} + \nu_{C_2} \leq 2\nu_C + k_C. \tag{5.5}$$

By Lemma 2.1(ii), we have

$$\rho_{C_i} + 2ak_{C_i} \leq \frac{\rho_C + 2ak_C}{2}, \qquad \text{for } i = 1, 2. \tag{5.6}$$

Let $\mathcal{E}_1$ (resp., $\mathcal{E}_2$) be the set of boxes obtained by applying all the free cuts in S_{C_1} (resp., S_{C_2}) in step 4 of the dividing stage. Clearly,

$$\Phi(\rho_C, \nu_C, k_C) = \sum_{E \in \mathcal{E}_1} \Phi(\rho_E, \nu_E, k_E) + \sum_{E \in \mathcal{E}_2} \Phi(\rho_E, \nu_E, k_E).$$

We consider two cases.

Case (i). $\mu < \rho_C + 2ak_C \leq 2\mu$. For each $i = 1, 2$

$$\rho_{C_i} + 2ak_{C_i} \leq \frac{\rho_C + 2ak_C}{2} \leq \mu.$$

As a result, all boxes in $\mathcal{E}_1$ and $\mathcal{E}_2$ are leaves of $\mathcal{T}_B$. Using (5.3) and (5.4), we obtain

$$\begin{aligned}\Phi(\rho_C, \nu_C, k_C) &\leq \sum_{E \in \mathcal{E}_1} f_E + \sum_{E \in \mathcal{E}_2} f_E \leq \sum_{E \in \mathcal{E}_1} (\rho_E + \nu_E) + \sum_{E \in \mathcal{E}_2} (\rho_E + \nu_E) \\ &\leq \rho_{C_1} + \nu_{C_1} + \rho_{C_2} + \nu_{C_2}.\end{aligned}$$

It now follows from (5.5) that

$$\Phi(\rho_C, \nu_C, k_C) \leq \rho_C + 2\nu_C + k_C, \tag{5.7}$$

which implies (5.1), because $\rho_C + 2ak_C > \mu$.

Case (ii). $\rho_C + 2ak_C > 2\mu$. For any box E in $\mathcal{E}_1 \cup \mathcal{E}_2$, by (5.2) and (5.6),

$$\max\left\{\frac{\rho_E + 2ak_E}{\mu}, 1\right\} \leq \max\left\{\frac{\rho_C + 2ak_C}{2\mu}, 1\right\} = \frac{\rho_C + 2ak_C}{2\mu}.$$

By (5.2) and the induction hypothesis,

$$\begin{aligned}\Phi(\rho_C, \nu_C, k_C) \leq& \sum_{E \in \mathcal{E}_1} \left(\rho_E + 2\nu_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right)\right) \\ &+ \sum_{E \in \mathcal{E}_2} \left(\rho_E + 2\nu_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_E \left(\frac{\rho_C + 2ak_C}{2\mu}\right)\right) \\ \leq& \left(\rho_{C_1} + \rho_{C_2}\right) + 2\left(\nu_{C_1} + \nu_{C_2}\right)\left(\frac{\rho_C + 2ak_C}{2\mu}\right) \\ &+ 2\left(k_{C_1} + k_{C_2}\right)\left(\frac{\rho_C + 2ak_C}{2\mu}\right).\end{aligned}$$

Using (5.5), we obtain

$$\begin{aligned}\Phi(\rho_C, \nu_C, k_C) &\leq \rho_C + 2\left(2\nu_C + k_C\right)\left(\frac{\rho_C + 2ak_C}{2\mu}\right) + 2k_C\left(\frac{\rho_C + 2ak_C}{2\mu}\right) \\ &= \rho_C + 2\nu_C\left(\frac{\rho_C + 2ak_C}{\mu}\right) + 2k_C\left(\frac{\rho_C + 2ak_C}{\mu}\right),\end{aligned}$$

which implies (5.1). ∎

LEMMA 5.3. *Let C be a box associated with a node in $\mathcal{T}_B$. If all rectangles in F_C belong to two classes, then*

$$\sum_{D \in L_C} f_D \leq O\left(\phi_C + (\nu_C + k_C)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right),$$

where $\mu = (f + ak)/a$.

Proof. Similar to the proof of Lemma 5.2. See the Appendix for details. ∎

LEMMA 5.4. *The tree $\mathcal{T}_B$ constructed on box B in a round has the following properties:*

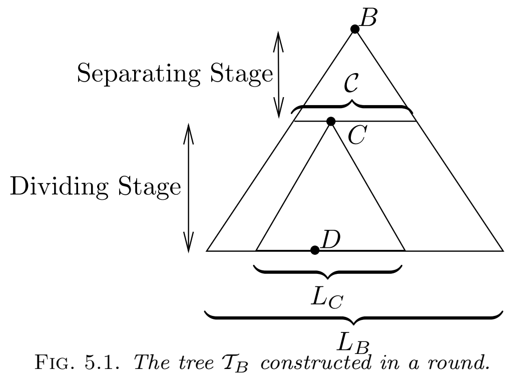


FIG. 5.1. *The tree $\mathcal{T}_B$ constructed in a round.*

(i) $\sum_{D \in L_B} k_D \leq k$,

(ii) $\sum_{D \in L_B} f_D = O(f + a^3 k)$, *and*

(iii) $|\mathcal{T}_B| = O((f + a^3 k) \log a)$.

Proof. The bound on $\sum_{D \in L_B} k_D$ is obvious, since each vertex in the interior of S_B lies in the interior of at most one box of L_B. Next, we use Lemma 5.3 to prove a bound on $\sum_{D \in L_B} f_D$.

Let $\mathcal{C}$ be the set of boxes into which B is partitioned by the separating stage; see Figure 5.1. Obviously, $\sum_{D \in L_B} f_D = \sum_{C \in \mathcal{C}} \sum_{D \in L_C} f_D$. For each box $C \in \mathcal{C}$, Lemma 3.1(i) implies that all rectangles in F_C belong to at most two classes. Hence, by Lemma 5.3,

$$\sum_{D \in L_B} f_D \leq \sum_{C \in \mathcal{C}} O\left(\phi_C + (\nu_C + k_C)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right).$$

Arguing as in the proof of Lemma 3.1(3), we can show that $\sum_{C \in \mathcal{C}} \phi_C = O(f)$ and that $\sum_{C \in \mathcal{C}} \nu_C = O(k)$. We also know that $\sum_{C \in \mathcal{C}} k_C \leq k$ and $\mu = (f + ak)/a$. Therefore,

$$\sum_{D \in L_B} f_D = O\left(f + k\left(\frac{f + 2ak}{\mu}\right)^3\right) = O\left(f + a^3 k\right).$$

We now sketch the proof that $|\mathcal{T}_B| = O((f + a^3 k) \log a)$. Following the same argument as in the proof of Lemma 3.1(2), we can show that the separating stage creates a tree with $O(f + k)$ nodes. We now count the number of nodes in $\mathcal{T}_B$ that are created by the dividing stage. Let $D \in \mathcal{T}_B$ be such a node. If D is partitioned by a cut containing a free rectangle (i.e., step 4 of the dividing stage is invoked at D), we charge D to its nearest ancestor $C \in \mathcal{T}_B$ such that C is not partitioned by a free cut (i.e., step 4 or 4 of the dividing stage is executed at C). Otherwise, we charge a cost of 1 to D itself. Let C be a node in $\mathcal{T}_B$ that is not partitioned by a free cut. Since a free rectangle can be created only by partitioning a long rectangle, the cut used to partition C creates $O(\phi_C + \nu_C)$ free rectangles, which implies that C is charged $O(\phi_C + \nu_C + 1)$ times by the above argument. Hence,

$$|\mathcal{T}_B| = O\left(\sum_C (\phi_C + \nu_C + 1)\right),$$

where C ranges over all nodes in $\mathcal{T}_B$ where step 4 or 4 of the dividing stage is executed. By following an inductive argument similar to the ones used to prove Lemmas 5.2 and 5.3, we can show that $|\mathcal{T}_B| = O((f + a^3k)\log a)$. Informally, the charging scheme compresses $\mathcal{T}_B$ by collapsing all nodes that are split by free cuts. Lemma 2.1 and 2.2 imply that the height of the compressed tree is $O(\log a)$. We can show that $\sum_C(\phi_C + \nu_C + 1)$ is roughly the product of the height of the tree and $\sum_D f_D$, the total number of long rectangles intersecting the leaves of the tree. $\square$

We now present our main result regarding the performance of our algorithm.

THEOREM 5.5. *Given a set S of n rectangles in $\mathbb{R}^3$ such that the aspect ratio of each rectangle in S is bounded by a constant $\alpha \geq 1$, we can construct a BSP of size $n2^{O(\sqrt{\log n})}$ for S in time $n2^{O(\sqrt{\log n})}$. The constants of proportionality in the big-oh terms are linear in* $\log \alpha$.

Proof. We first bound the size of the BSP constructed by the algorithm. Let $S(f, k)$ denote the maximum size of the BSP produced by the algorithm for a box B that contains f long rectangles and k vertices in its interior. If $k = 0$, Theorem 3.2 implies that $S(f, k) = O(f \log f)$. For $k > 0$, by Lemma 5.4(iii), we construct the subtree $\mathcal{T}_B$ on B of size $O((f + a^3k)\log a)$ in one round, and recursively construct subtrees for each box in the set of leaves L_B. Therefore, there exist constants $c_1, c_2, c_3 > 0$ so that the size $S(f, k)$ satisfies the following recurrence:

$$S(f,k) \leq \begin{cases} c_1 f \log f & \text{for} \quad k = 0, \\ \displaystyle\sum_{D \in L_B} S(f_D, k_D) + c_2(f + a^3k)\log a & \text{for} \quad k > 0, \end{cases}$$

where

$$f_D + 2ak_D \leq \frac{f + 2ak}{a}$$

for every box D in L_B (by Lemma 5.1), and

$$\sum_D k_D \leq k, \qquad \sum_D f_D \leq c_3(f + a^3k)$$

(by Lemma 5.4(i) and 5.4(ii). Using induction on $f + 2ak$, we can prove that the solution to the above recurrence is

$$S(f,k) = (f + k)2^{O(\sqrt{\log(f+k)})},$$

where the constant of proportionality is linear in $\log \alpha$. Intuitively, the algorithm constructs the BSP for B in $O(\log a) = O(\sqrt{\log(f + k)})$ rounds, and the total number of long rectangles increases roughly by a constant factor in each round. The $n2^{O(\sqrt{\log n})}$ bound on the size of the BSP constructed by the algorithm follows, since $f \leq n$ and $k \leq 4n$ at the beginning of the first round.

We now bound the running time of our algorithm. Recall that we initially sorted the vertices of the rectangles in S by x-, y-, and z-coordinates. Suppose S_B does not contain a free rectangle. By Lemmas 2.1 and 2.2, the cuts to be made at B can be determined in $O(|S_B|)$ time. Suppose C is a box obtained by partitioning B according to these cuts; we can easily obtain the sorted order of the vertices of the rectangles in S_C from the sorted order in B. Let $\mathcal{E}$ be the set of boxes obtained by applying C using all the free rectangles in S_C. Since the free rectangles in S_C are parallel to each other, we can partition the rectangles in S_C among the boxes in $\mathcal{E}$ in $O(|S_C|)$ time. Therefore, we can construct the tree representing the partition of B into the set of boxes E in $O(|S_B|)$ time. Hence, we obtain the same $n2^{O(\sqrt{\log n})}$ bound for the running time of the algorithm. $\square$

Remark. We can modify our algorithm to prove that the height of the BSP constructed is $O(\log n)$: if S_B contains free rectangles, we assign appropriate weights to the free rectangles, and partition B using the weighted median of the free rectangles. We leave the details to the reader. Paterson and Yao [25] use a similar idea to bound the height of BSP they construct for segments in the plane.

6. Extensions.

In this section, we extend the algorithm of section 4 to the following two cases: (i) some of the input rectangles are thin and (ii) some of the input polygons are triangles.

6.1. Fat and thin rectangles.

Let us assume that the input $S = F \cup T$ has n rectangles, consisting of $m \geq 1$ thin rectangles in T and $n - m$ fat rectangles in F. We first describe our algorithm and then construct a set of rectangles for which any BSP has size $\Omega(n\sqrt{m})$. The algorithm we use now is very similar to the algorithm for fat rectangles. Given a box B, let f be the number of long rectangles in F_B, k the number of vertices of rectangles in F_B that lie in the interior of B, and t the number of rectangles in T_B. We fix a parameter $a = 2^{\sqrt{\log(f+k)}}$ and perform the following steps:

1. If S_B contains a free rectangle, we use the corresponding free cut to split B into two boxes.
2. If $k = t = 0$, we use the algorithm for long rectangles to construct a BSP for the set of clipped rectangles S_B.
3. If $t \geq (f + k)$, we use the algorithm by Paterson and Yao for orthogonal rectangles in $\mathbb{R}^3$ to construct a BSP for S_B [26].
4. If $(f + k) > t$, we perform one round of the algorithm described in section 4, with the difference that we also use thin rectangles to make free cuts.

This algorithm is recursively invoked on all the resulting subboxes. Let $S(f, k, t)$ be the maximum size of the BSP produced by this algorithm for a box B with k vertices in its interior, f long rectangles in F_B, and t thin rectangles in T_B. Note that in section 5, during the analysis of a round, we did not use the fact that the short rectangles were fat. As a result, Lemmas 5.2 and 5.3 hold for step 6.1 above, with $\nu_C + k_C + t_C$ replacing the term $\nu_C + k_C$. We can similarly extend Lemma 5.4 to obtain the following recurrence for $S(f, k, t)$. Here D ranges over all the boxes that B is divided into by a round of cuts, as described above in step 6.1.

$$S(f,k,t) = \begin{cases} O(f \log f), & \text{for} \quad k = t = 0, \\ O(t\sqrt{t}), & \text{for} \quad t \geq f + k, \\ \sum_D S(f_D, k_D, t_D) + O(f \log a + a^3 k + a^3 t), & \text{for} \quad f + k > t, \end{cases}$$

where $\sum_D k_D \leq k$, $f_D + 2ak_D \leq (f + 2ak)/a$, $\sum_D f_D = O(f + a^3 k)$, and $\sum_D t_D = O(a^3 t)$.

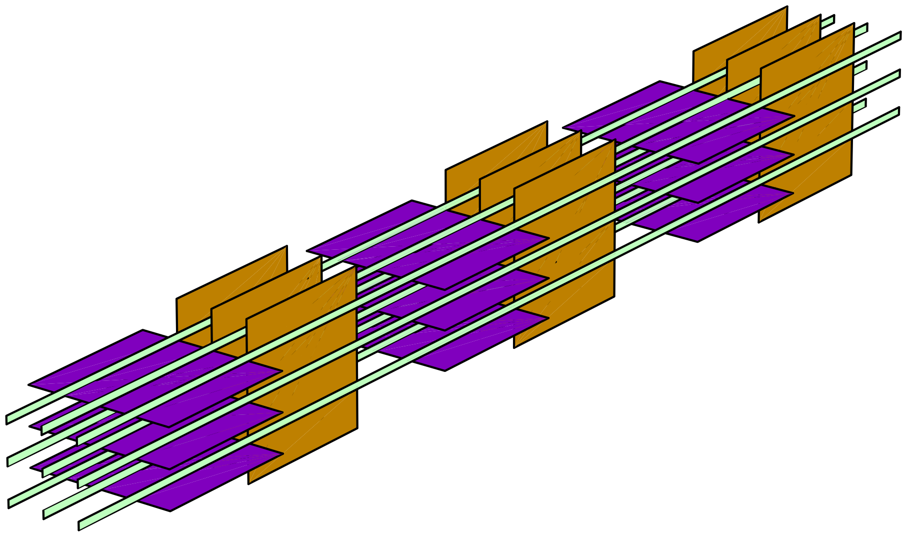


FIG. 6.1. *Lower bound construction for thin and fat rectangles.*

We can analyze this recurrence as in section 5 and show that its solution is

$$S(f,k,t) = (f+k)\sqrt{t}2^{O(\sqrt{\log(f+k)})},$$

where the constant of proportionality is linear in $\log \alpha$. The following theorem is immediate.

THEOREM 6.1. *Let S be a set of n rectangles in $\mathbb{R}^3$, of which $m \geq 1$ are thin. A BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ for S can be constructed in $n\sqrt{m}2^{O(\sqrt{\log n})}$ time. The constants of proportionality in the big-oh terms are linear in $\log \alpha$, where α is the maximum aspect ratio of the fat rectangles.*

We can show that Theorem 6.1 is almost optimal by constructing a set of n rectangles of which m are thin, for which any BSP has size $\Omega(n\sqrt{m})$. Recall that there exists a set of m thin rectangles in $\mathbb{R}^3$ for which any BSP has size $\Omega(m\sqrt{m})$ [26]. To complete the proof of the lower bound, we now exhibit a set $S = T \cup F$ of n rectangles, where T is a set of m thin rectangles and F is a set of $n-m$ fat rectangles, for which any BSP has size $\Omega((n-m)\sqrt{m})$. The rectangles in T are arranged in a $\sqrt{m}\times\sqrt{m}$ grid; each rectangle in T is a segment of length $n-m+5(n-m)/(2\sqrt{m})+1$ perpendicular to the yz-plane. The rectangles in F are divided into $(n-m)/(2\sqrt{m})$ sets, each consisting of $2\sqrt{m}$ squares with side length $\sqrt{m}+2$. Each set consists of $\sqrt{m}$ squares parallel to the xy-plane and $\sqrt{m}$ squares parallel to the xz-plane so that the following three conditions are satisfied (see Figure 6.1).

1. In each set, a square parallel to the xy-plane is at a distance of 2ε from any square parallel to the xz-plane,
2. For any square, the closest square in a different set is at a distance of $1-2\varepsilon$, and
3. For any square, the closest thin rectangle is at a distance of ε.

We can show that there are $(n-m)\sqrt{m}/2$ points such that a cube of side 2ε centered at any such point intersects a thin rectangle t of T and two squares r and s of F. For each such cube ψ, we can show that at least one edge of r, s, or t is crossed in the interior of ψ by a cutting plane of $\mathcal{B}$, which implies that the cutting planes in $\mathcal{B}$ and the edges of the rectangles in S cross at $\Omega((n-m)\sqrt{m})$ points, thus proving the lower bound on the size of $\mathcal{B}$.

6.2. Fat rectangles and triangles. Suppose that $p \geq 1$ polygons in the input S are (nonorthogonal) triangles and that the rest are fat rectangles. To construct a BSP for S, we use nonorthogonal cutting planes; hence, each region is a convex polytope and the intersection of a triangle with a region is a polygon, possibly with more than three edges. We can extend the algorithm of section 6.1 to this case as follows: In step 6.1, we check whether we can make free cuts through the nonorthogonal polygons too. In step 6.1, if the number of triangles at a node is greater than the number of fat rectangles, we use the algorithm of Agarwal et al. for triangles in $\mathbb{R}^3$ to construct a BSP of size quadratic in the number of triangles in near-quadratic time [2]. Proceeding as in the previous section, we can prove the following theorem.

THEOREM 6.2. *A BSP of size* $np2^{O(\sqrt{\log n})}$ *can be constructed in* $np2^{O(\sqrt{\log n})}$ *time for* n *polygons in* $\mathbb{R}^3$*, of which* $p \geq 1$ *are nonorthogonal and the rest are fat rectangles. The constants of proportionality in the big-oh terms are linear in* $\log \alpha$*, where* α *is the maximum aspect ratio of the fat rectangles.*

Unlike the case of rectangles, the fatness assumption does not help in constructing BSPs of small size for triangles. More specifically, we can show that there exists a set of n fat triangles in $\mathbb{R}^3$ such that any BSP for these triangles has $\Omega(n^2)$ size by modifying Chazelle's construction for proving a quadratic lower bound on the size of convex decompositions of polyhedra in $\mathbb{R}^3$ [9].

7. Conclusions. In this paper, we have studied the problem of constructing BSPs for orthogonal rectangles under the natural assumption that most rectangles are fat. Our result shows that this assumption allows a smaller worst-case size of BSPs. Our algorithm constructs a BSP for any set of orthogonal rectangles; it is only the analysis of the algorithm that depends on the fatness of the input rectangles. We have implemented a variant of our algorithm and compared its performance to that of other known algorithms [18]. Our algorithm is indeed practical: it constructs a BSP of near-linear size on real data sets. It performs better than not only Paterson and Yao's algorithm [26] but also most heuristics described in the literature [4, 16, 30].

We now briefly mention another extension to our algorithms. If the n fat rectangles contain $k \geq 1$ crossing pairs, we can construct a BSP of size $(n+k)\sqrt{k}2^{O(\sqrt{\log n})}$ for these rectangles as follows: for each crossing pair r and s, we partition one of the rectangles (say, r) into two smaller rectangles that do not intersect s. We construct a BSP for the resulting $n + O(k)$ rectangles by invoking our algorithm for a set of fat and thin triangles. We can also construct a set of n fat rectangles with k crossing pairs for which any BSP has size $\Omega(n + k\sqrt{k})$.

It seems very probable that BSPs of a size smaller than $n2^{O(\sqrt{\log n})}$ can be built for n fat rectangles in $\mathbb{R}^3$. The only lower bound we have is the trivial $\Omega(n)$ bound. It would be interesting to see if simple heuristics (e.g., choose the next splitting plane to be one that intersects the smallest number of rectangles) can be proven to construct BSPs of (close to) optimal size. An even more challenging open problem is determining the right assumptions that should be made about the input objects and the graphics display hardware so that provably fast and *practically efficient* algorithms can be developed for doing hidden-surface elimination of these objects.

Appendix. Proof of Lemma 5.3.

Lemma 5.3. *Let C be a box associated with a node in $\mathcal{T}_B$. If all rectangles in F_C belong to two classes, then*

$$\sum_{D \in L_C} f_D \le O\left(\phi_C + (\nu_C + k_C)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right),$$

where $\mu = (f + ak)/a$.

Proof. Let

$$\Psi(\phi_C, \nu_C, k_C) = \max \sum_{D \in L_C} f_D,$$

where the maximum is taken over all boxes C and over all sets S of rectangles with ϕ_C long rectangles in F_C, ν_C long rectangles in $S_C \setminus F_C$, and k_C vertices in the interior of C. The rectangles in F_C belong to at most two classes. We claim that

(A.1)
$$\Psi(\phi_C, \nu_C, k_C) \le 2\phi_C + 5\nu_C \max\left\{\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3, 1\right\} + 6k_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3,$$

which proves the lemma.

If S_C contains $m \ge 1$ free rectangles, we apply the free cuts containing these rectangles to partition C (by repeatedly invoking step 4 of the dividing stage) until the resulting boxes do not contain any free rectangles. Let $\mathcal{E}$ be the set of boxes into which C is so partitioned. Then

(A.2) $$\phi_E + 2ak_E \le \phi_C + 2ak_C, \qquad \text{for any box } E \text{ in } \mathcal{E},$$

(A.3) $$\sum_{E \in \mathcal{E}} \phi_E \le \phi_C \qquad \sum_{E \in \mathcal{E}} \nu_E \le \nu_C \qquad \sum_{E \in \mathcal{E}} k_E \le k_C.$$

These inequalities imply that if (A.1) holds for each box in $\mathcal{E}$, then (A.1) holds for C as well. Therefore, we prove (A.1) for all boxes C such that F_C contains at most two classes of rectangles and S_C does not contain any free rectangles. We proceed by induction on $\phi_C + 2ak_C$.

Base case. $\phi_C + 2ak_C \le \mu$. Since $\phi_C + 2ak_C \le \mu$ and S_C does not contain any free rectangles, C is a leaf of $\mathcal{T}_B$. We have

(A.4) $$\Psi(\phi_C, \nu_C, k_C) = \sum_{D \in L_C} f_D = f_C = \phi_C + \nu_C,$$

which implies (A.1).

Induction step. $\phi_C + 2ak_C > \mu$. The cuts made in step 4 of the dividing stage fall into one of two categories (see Lemma 2.2). Note that none of these cuts contains a free rectangle.

Case (i). We divide C into two boxes, C_1 and C_2, using a plane h that does not cross any rectangle in F_C. As a result,

$$\phi_{C_1} + \phi_{C_2} \le \phi_C \qquad \text{and} \qquad k_{C_1} + k_{C_2} \le k_C.$$

Since h intersects each rectangle in S_C at most once, (5.5) holds in this case too; i.e.,

(A.5) $$\nu_{C_1} + \nu_{C_2} \le 2\nu_C + k_C.$$

Lemma 2.2 implies that

$$\phi_{C_i} + 2ak_{C_i} \leq \frac{2(\phi_C + 2ak_C)}{3} \qquad \text{for } i = 1, 2. \tag{A.6}$$

Let $\mathcal{E}_1$ (resp., $\mathcal{E}_2$) be the set of boxes obtained by applying all the free cuts in S_{C_1} (resp., S_{C_2}) in step 4 of the dividing stage. Clearly,

$$\Psi(\phi_C, \nu_C, k_C) \leq \sum_{E \in \mathcal{E}_1} \Psi(\phi_E, \nu_E, k_E) + \sum_{E \in \mathcal{E}_2} \Psi(\phi_E, \nu_E, k_E).$$

We consider two cases.

(a) $\mu \leq \phi_C + 2ak_C \leq 3\mu/2$. In this case, by (A.2) and (A.6),

$$\phi_E + 2ak_E \leq \frac{2(\phi_C + 2ak_C)}{3} \leq \mu,$$

for each box E in $\mathcal{E}_1$ and $\mathcal{E}_2$. Since E does not contain a free rectangle, E is a leaf of $\mathcal{T}_B$. Using (A.5), (A.3), and (A.4), we obtain

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) &\leq \sum_{E \in \mathcal{E}_1} f_E + \sum_{E \in \mathcal{E}_2} f_E \\ &\leq \sum_{E \in \mathcal{E}_1} (\phi_E + \nu_E) + \sum_{E \in \mathcal{E}_2} (\phi_E + \nu_E) \\ &\leq \phi_{C_1} + \nu_{C_1} + \phi_{C_2} + \nu_{C_2} \\ &\leq \phi_C + 2\nu_C + k_C,\end{aligned}$$

which implies (A.1), because $\phi_C + 2ak_C > \mu$.

(b) $\phi_C + 2ak_C > 3\mu/2$. For a box E in $\mathcal{E}_1 \cup \mathcal{E}_2$, by (A.2) and (A.6), we have

$$\max\left\{\left(\frac{\phi_E + 2ak_E}{\mu}\right)^3, 1\right\} \leq \max\left\{\left(\frac{2}{3}\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right)^3, 1\right\} = \frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

By the induction hypothesis, and by using (A.3) and (A.5),

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) \leq &\sum_{E \in \mathcal{E}_1} \left(2\phi_E + 5\nu_E \left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right) + 6k_E \left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right)\right) \\ &+ \sum_{E \in \mathcal{E}_2} \left(2\phi_E + 5\nu_E \left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right) + 6k_E \left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right)\right) \\ \leq &\; 2\left(\phi_{C_1} + \phi_{C_2}\right) + 5\left(\nu_{C_1} + \nu_{C_2}\right)\left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right) \\ &\quad + 6\left(k_{C_1} + k_{C_2}\right)\left(\frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right) \\ \leq &\; 2\phi_C + 5 \cdot \frac{8}{27}\left(2\nu_C + k_C\right)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6 \cdot \frac{8}{27} k_C \left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 \\ \leq &\; 2\phi_C + 5\nu_C \left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6k_C \left(\frac{\phi_C + 2ak_C}{\mu}\right)^3,\end{aligned}$$

which implies (A.1).

Case (ii). We find two parallel planes h_1 and h_2 that divide C into three boxes C_1, C_2, and C_3 (in this order) so that all rectangles in F_{C_2} belong to one class. Lemma 2.2 implies that the rectangles in F_C are partitioned among C_1, C_2, and C_3. Moreover, h_1 and h_2 partition the vertices in the interior of C. Thus,

$$\phi_{C_1} + \phi_{C_2} + \phi_{C_3} \leq \phi_C \qquad \text{and} \qquad k_{C_1} + k_{C_2} + k_{C_3} \leq k_C.$$

The planes h_1 and h_2 can intersect the rectangles in $S_C \setminus F_C$. Each long rectangle in $S_C \setminus F_C$ is partitioned into at most three long rectangles. Each short rectangle in S_C is partitioned into at most three rectangles; if two of these rectangles are long, then one of the long rectangles must be in S_{C_2}, since C_2 is sandwiched between C_1 and C_3 (see the proof of Lemma 2.2). In other words,

$$\nu_{C_1} + \nu_{C_3} \leq 2\nu_C + k_C \qquad \text{and} \qquad \nu_{C_2} \leq \nu_C + k_C. \tag{A.7}$$

Lemma 2.2 also implies that

(A.8)
$$\phi_{C_i} + 2ak_{C_i} \leq \frac{2(\phi_C + 2ak_C)}{3}, \qquad \text{for } i = 1, 3, \text{ and } \phi_{C_2} + 2ak_{C_2} \leq \phi_C + 2ak_C.$$

Let $\mathcal{E}_i, 1 \leq i \leq 3$ be the set of boxes obtained by applying all the free cuts in S_{C_i} in step 4 of the dividing stage. Note that for each box $E \in \mathcal{E}_2$, F_E contains only one class of rectangles. It is clear that

$$\Psi(\phi_C, \nu_C, k_C) \leq \sum_{E \in \mathcal{E}_1} \Psi(\phi_E, \nu_E, k_E) + \sum_{E \in \mathcal{E}_2} \Phi(2\phi_E, \nu_E, k_E) + \sum_{E \in \mathcal{E}_3} \Psi(\phi_E, \nu_E, k_E),$$

where $\Phi(\,)$ is as defined in the proof of Lemma 5.2. We again consider two cases.

(a) $\mu < \phi_C + 2ak_C \leq 3\mu/2$. By (A.8), each box in $\mathcal{E}_1$ and $\mathcal{E}_3$ is a leaf of $\mathcal{T}_B$. Therefore, by (A.4),

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) &\leq \sum_{E \in \mathcal{E}_1} f_E + \sum_{E \in \mathcal{E}_2} \Phi\left(2\phi_E, \nu_E, k_E\right) + \sum_{E \in \mathcal{E}_3} f_E \\ &\leq \sum_{E \in \mathcal{E}_1} (\phi_E + \nu_E) + \sum_{E \in \mathcal{E}_3} (\phi_E + \nu_E) + \sum_{E \in \mathcal{E}_2} \Phi\left(2\phi_E, \nu_E, k_E\right) \\ &\leq (\phi_{C_1} + \phi_{C_3}) + (\nu_{C_1} + \nu_{C_3}) + \sum_{E \in \mathcal{E}_2} \Phi\left(2\phi_E, \nu_E, k_E\right).\end{aligned}$$

For each box $E \in \mathcal{E}_2$, $\phi_E + 2ak_E \leq 3\mu/2$. Hence, by (5.7), we have

$$\Phi\left(2\phi_E, \nu_E, k_E\right) \leq 2\phi_E + 2\nu_E + k_E.$$

As a result,

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) &\leq (\phi_{C_1} + \phi_{C_3}) + (\nu_{C_1} + \nu_{C_3}) + \sum_{E \in \mathcal{E}_2} (2\phi_E + 2\nu_E + k_E) \\ &\leq (\phi_{C_1} + \phi_{C_3}) + (\nu_{C_1} + \nu_{C_3}) + (2\phi_{C_2} + 2\nu_{C_2} + k_{C_2}) \\ &\leq 2\phi_C + 4\nu_C + 4k_C,\end{aligned}$$

where the last inequality follows from (A.7). This inequality implies (A.1).

(b) $\phi_C + 2ak_C > 3\mu/2$. For a box $E \in \mathcal{E}_1 \cup \mathcal{E}_3$,

$$\max\left\{\left(\frac{\phi_E + 2ak_E}{\mu}\right)^3, 1\right\} \le \max\left\{\left(\frac{2}{3}\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right)^3, 1\right\} = \frac{8}{27}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

Similarly, for a box $E \in \mathcal{E}_2$,

$$\max\left\{2\left(\frac{\phi_E + ak_E}{\mu}\right), 1\right\} \le 2\left(\frac{\phi_C + 2ak_C}{\mu}\right).$$

By the induction hypothesis and (5.1),

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) \le &\sum_{E \in \mathcal{E}_1}\left(2\phi_E + 5 \cdot \frac{8}{27}\nu_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6 \cdot \frac{8}{27}k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right)\\ &+ \sum_{E \in \mathcal{E}_2}\left(2\phi_E + 4\nu_E\left(\frac{\phi_C + 2ak_C}{\mu}\right) + 4k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)\right)\\ &+ \sum_{E \in \mathcal{E}_3}\left(2\phi_E + 5 \cdot \frac{8}{27}\nu_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6 \cdot \frac{8}{27}k_E\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\right).\end{aligned}$$

Since $\phi_C + 2ak_C > 3\mu/2$,

$$\frac{\phi_C + 2ak_C}{\mu} \le \frac{4}{9}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3.$$

Therefore, using (A.7), we have

$$\begin{aligned}\Psi(\phi_C, \nu_C, k_C) &\le 2\left(\phi_{C_1} + \phi_{C_2} + \phi_{C_3}\right) + 5 \cdot \frac{8}{27}\left(\nu_{C_1} + \nu_{C_3}\right)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\\ &\quad + \frac{16}{9}\nu_{C_2}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6 \cdot \frac{8}{27}\left(k_{C_1} + k_{C_3}\right)\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\\ &\quad + \frac{16}{9}k_{C_2}\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3\\ &\le 2\phi_C + 5\nu_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3 + 6k_C\left(\frac{\phi_C + 2ak_C}{\mu}\right)^3,\end{aligned}$$

which implies (A.1). ☐

Acknowledgments. We would like to thank Seth Teller for providing us with the Soda Hall dataset (shown in Figure 1.1) created at the Department of Computer Science, University of California at Berkeley.

REFERENCES

[1] P. K. Agarwal, J. Erickson, and L. J. Guibas, *Kinetic binary space partitions for intersecting segments and disjoint triangles*, in Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1998, pp. 107–116.

[2] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter, *Cylindrical static and kinetic binary space partitions*, in Proceedings of the 13th ACM Symposium on Comput. Geom., ACM, New York, 1997, pp. 39–48.

[3] P. K. Agarwal and S. Suri, *Surface approximation and geometric partitions*, in Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, 1994, pp. 24–33.

[4] J. M. Airey, *Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations*, Ph.D. thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, NC, 1990.

[5] C. Ballieux, *Motion Planning Using Binary Space Partitions*, Technical Report inf/src/93-25, Utrecht University, Utrecht, The Netherlands, 1993.

[6] A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, TX 1991.

[7] T. Cassen, K. R. Subramanian, and Z. Michalewicz, *Near-optimal construction of partitioning trees by evolutionary techniques*, in Proceedings of Graphics Interface '95, Quebec, Canada, Canadian Human-Computer Communications Society, 1995, pp. 263–271.

[8] E. Catmull, *A subdivision algorithm for computer display of curved surfaces*, Technical Report UTEC-CSC-74-133, Department Computer Sciences, University of Utah, Salt Lake City, UT, 1974.

[9] B. Chazelle, *Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm*, SIAM J. Comput., 13 (1984), pp. 488–507.

[10] N. Chin and S. Feiner, *Near real-time shadow generation using BSP trees*, in Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques, Boston, MA, 1989, pp. 99–106.

[11] N. Chin and S. Feiner, *Fast object-precision shadow generation for area light sources using BSP trees*, in Proceedings of the ACM Symposium on Interactive 3D Graphics, Cambridge, MA, 1992, pp. 21–30.

[12] Y. Chrysanthou, *Shadow Computation for 3D Interaction and Animation*, Ph.D. thesis, Queen Mary and Westfield College, University of London, London, UK, 1996.

[13] M. de Berg, *Linear size binary space partitions for fat objects*, in Proceedings of the 3rd Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 979, Springer-Verlag, Berlin, 1995, pp. 252–263.

[14] S. E. Dorward, *A survey of object-space hidden surface removal*, Internat. J. Comput. Geom. Appl., 4 (1994), pp. 325–362.

[15] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA, 1990.

[16] H. Fuchs, Z. M. Kedem, and B. Naylor, *On visible surface generation by a priori tree structures*, in Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques, ACM, New York, 1980, pp. 124–133.

[17] C. Mata and J. S. B. Mitchell, *Approximation algorithms for geometric tour and network design problems*, in Proceedings of the 11th Annual ACM Symposium on Comput. Geom., ACM, New York, 1995, pp. 360–369.

[18] T. M. Murali, P. K. Agarwal, and J. S. Vitter, *Constructing binary space partitions for orthogonal rectangles in practice*, in Proceedings of the 6th Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 1461, Springer, Berlin, 1998, pp. 211–222.

[19] T. M. Murali and T. A. Funkhouser, *Consistent solid and boundary representations from arbitrary polygonal data*, in Proceedings of the ACM Symposium on Interactive 3D Graphics, Providence, RI, 1997, pp. 155–162.

[20] B. Naylor, *Constructing good partition trees*, in Proceedings of Graphics Interface, Toronto, Ontario, Canada, Canadian Human-Computer Communications Society, 1993, pp. 181–191.

[21] B. Naylor and W. Thibault, *Application of BSP Trees to Ray-Tracing and CSG Evaluation*, Technical Report GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[22] B. F. Naylor, *SCULPT: An interactive solid modeling tool*, in Proceedings of Graphics Interface, Halifax, Nova Scotia, Canada, Canadian Human-Computer Communications Society, 1990, pp. 138–148.

[23] B. F. Naylor, *Interactive solid geometry via partitioning trees*, in Proceedings of Graphics Interface, Vancouver, B.C., Canada, Canadian Human-Computer Communications Society, 1992, pp. 11–18.

[24] B. F. Naylor, J. Amanatides, and W. C. Thibault, *Merging BSP trees yields polyhedral set operations*, in Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques, Dallas, TX, 1990, pp. 115–124.

[25] M. S. Paterson and F. F. Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete Comput. Geom., 5 (1990), pp. 485–503.

[26] M. S. Paterson and F. F. Yao, *Optimal binary space partitions for orthogonal objects*, J.

Algorithms, 13 (1992), pp. 99–113.

[27] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, *Study for Applying Computer-Generated Images to Visual Simulation*, Technical Report AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, Brooks Air Force Base, San Antonio, TX, 1969.

[28] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, *A characterization of ten hidden-surface algorithms*, ACM Comput. Surv., 6 (1974), pp. 1–55.

[29] S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. thesis, Department of Computer Science, University of California at Berkeley, Berkeley, CA 1992.

[30] W. C. Thibault and B. F. Naylor, *Set operations on polyhedra using binary space partitioning trees*, in Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques, Anaheim, CA, 1987, pp. 153–162.

[31] E. Torres, *Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes*, in Proceedings of Eurographics, European Association for Computer Graphics, Aire-la-Ville, Switzerland, 1990, pp. 507–518.

SIAM J. Comput.
Vol. 29, No. 5, pp. 1449–1483

WAIT-FREE k-SET AGREEMENT IS IMPOSSIBLE: THE TOPOLOGY OF PUBLIC KNOWLEDGE*

MICHAEL SAKS† AND FOTIOS ZAHAROGLOU‡

Abstract. In the classical consensus problem, each of n processors receives a private input value and produces a decision value which is one of the original input values, with the requirement that all processors decide the same value. A central result in distributed computing is that, in several standard models including the asynchronous shared-memory model, this problem has no deterministic solution. The k-set agreement problem is a generalization of the classical consensus proposed by Chaudhuri [*Inform. and Comput.*, 105 (1993), pp. 132–158], where the agreement condition is weakened so that the decision values produced may be different, as long as the number of distinct values is at most k. For $n > k \geq 2$ it was not known whether this problem is solvable deterministically in the asynchronous shared memory model. In this paper, we resolve this question by showing that for any $k < n$, there is no deterministic wait-free protocol for n processors that solves the k-set agreement problem. The proof technique is new: it is based on the development of a topological structure on the set of possible processor schedules of a protocol. This topological structure has a natural interpretation in terms of the knowledge of the processors of the state of the system. This structure reveals a close analogy between the impossibility of wait-free k-set agreement and the Brouwer fixed point theorem for the k-dimensional ball.

Key words. distributed computing, consensus, Sperner's lemma, wait-free

AMS subject classifications. 68Q22, 68R10, 54A99

PII. S0097539796307698

1. Introduction.

1.1. Wait-free algorithms and the k-set agreement problem.

In totally asynchronous multiprocessor systems without global clocks, the execution speed of each processor may fluctuate widely. A highly desirable property for protocols in such a system is that no processor ever wait indefinitely for an action by another processor, that is, unless a processor fails (stops running) it is guaranteed to complete its task regardless of the relative speeds of the other processors, even if other processors stop participating. Protocols with this property are said to be *wait-free*.

We are interested in the standard model of shared-memory distributed systems with atomic registers [20]; an essentially equivalent model has been studied as *asynchronous parallel random access machines (PRAMs)* (e.g., [12, 22]). We restrict consideration to the case where each processor is deterministic. Informally such a system consists of a set of processors each with its own local memory accessible only to itself, and a set of shared registers. Each shared register supports atomic read and write operations, which means that (1) if two processors access a register simultaneously, the register automatically serializes the accesses, so there are no collisions, and (2)

*Received by the editors August 2, 1996; accepted for publication (in revised form) June 9, 1999; published electronically March 15, 2000. This work was supported in part by NSF grants CCR-8911388, CCR-939215293, and CCR-9700239 and by DIMACS. A preliminary version appeared in the Proceedings of the 24th annual ACM Symposium on Theory of Computing. The material in this paper appeared in somewhat different form in the second author's Ph.D. dissertation, completed at University of California San Diego, May 1993.
http://www.siam.org/journals/sicomp/29-5/30769.html

†Department of Mathematics, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019 (saks@math.rutgers.edu).

‡Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA 92093. This work was performed while the second author was visiting DIMACS.

a processor cannot simultaneously both read and write to a register. A protocol is defined by a set of programs, one for each processor, where each program involves "private computations," together with reads and writes to the shared memory. The system is completely asynchronous and the protocol makes no reference to a clock.

In executing a particular protocol the system may exhibit a wide range of behavior, depending on the relative speeds of the processors. The execution thus depends on the *schedule* of the processors, i.e., the way in which the program steps of the individual processors are interleaved. For a protocol to be correct, it should be correct for *all* schedules. A useful way to think about this requirement is to view the schedule as being chosen by an adversary who seeks to force the protocol to behave incorrectly.

Effective computation in such systems requires some coordination among the processors. The *consensus* problem was introduced as an abstraction of one coordination problem. In this problem, each processor p receives a private input x_p and must produce as output a *decision value* d_p, subject to the following requirements:

Validity. Each decision value is the input value of some processor.

Consistency. All processors that decide must decide the same value.

A fundamental result for the deterministic shared memory model outlined above is that there is no wait-free protocol that solves the consensus problem. This result was proven for this model by Herlihy [16] and independently by Loui and Abu-Amara [21] by adapting the proof of the seminal impossibility result for consensus in message passing systems with one failing processor proved by Fischer, Lynch, and Paterson [15].

This impossibility result spawned considerable activity along several fronts. One direction is to strengthen the model (i.e., introduce randomization, strengthen the shared memory primitives) so as to make consensus achievable. A second direction, pioneered by Herlihy, is the classification of data objects according to the number of processors that can achieve consensus using this data object [16]. A third direction has been to understand fully what can and can't be done in the deterministic atomic register shared-memory model.

As a step toward this third goal, it is natural to consider a weaker version of the consensus problem called the *k-set agreement problem.* This problem is identical to the consensus problem except that the *consistency* condition is replaced by a weaker condition:

k-Consistency. The set of decision values produced by the processors has cardinality at most k.

This problem was proposed and studied by Chaudhuri [10], who considered it in the message passing model and obtained some results relating the difficulty of this problem to various other related problems. In the shared atomic register model the main question is, For which values of n and k is there a wait-free algorithm for n processors to achieve k-set agreement in the shared-memory model? Trivially such an algorithm is possible for $n \leq k$ and (by the result for consensus) is impossible for $k = 1$ and $n > 1$. Chaudhuri conjectured that k-set agreement is impossible for any $n > k$. To appreciate the deceptive difficulty of the problem, the reader may consider the first previously unsolved case $k = 2$ and $n = 3$. This seemingly elementary brain teaser is already quite challenging.

1.2. Main results.

In this paper we prove the following theorem.

THEOREM 1.1. *For $k < n$, there is no deterministic wait-free protocol in the shared atomic registers model which solves the k-set agreement problem in a system of n processors.*

The first step in the proof is the formulation of a new formal model for shared-memory atomic register systems, called the *weakly synchronous model.* For our purposes, this model is at least as powerful as the standard ones (so impossibility proofs for this model apply to those) but it has the advantage of a particularly simple combinatorial structure. This allows us to reformulate the given problem in purely combinatorial terms.

We then develop a new approach for reasoning about computability issues in distributed systems. The basis of this approach is to shift focus from the structure of protocols for a distributed system to the structure of the set of possible schedules of a distributed system. To accomplish this shift for a given protocol, we fix ("hardwire") a particular set of input values to the processors and observe that having done this, the processor schedule now completely determines the output values of the processors and thus can be viewed as the "input" to the system. We introduce two key notions: (i) two schedules are "indistinguishable" if for any protocol they exhibit the same output behavior, (ii) a set S of schedules is "knowable" if there is a protocol which "recognizes" it, in the sense that for some specified output symbol, the protocol produces that symbol during an execution if and only if the execution proceeds according to some schedule from S.

These two concepts lead naturally to the definition of a topology on the set of schedules, and Theorem 1.1 is proved by analyzing this topology. Our approach reveals and exploits a close analogy between the impossibility of wait-free k-set agreement and a lemma of Knaster, Kuratowski, and Mazurkiewicz (KKM lemma) [1], which is equivalent to the fixed point theorem for the closed unit ball B_m in m-dimensional Euclidean space: if f is a continuous map from B_m to itself, then there exists a point $x \in B_m$ such that $f(x) = x$. Very roughly, f corresponds to a distributed protocol Π, and the fixed point x corresponds to the schedule for which Π fails to solve the k-set agreement. The increase in difficulty of the k-set agreement proof in going from the case $k = 1$ to the case $k > 1$ corresponds to the increase in difficulty in going from the fixed point theorem for the interval $[-1, 1]$, which is very simple, to the theorem for balls in higher dimension, which, while elementary, is considerably harder. An additional obstacle in our work is that, while the topological structure of B_m is well understood, we must develop the topological structure for the set of schedules from scratch.

While the explicit use of topology can be avoided, we have retained the topological structure of the proof, because this is what drove the proof and it provides important insight into what is going on. Our topological structure has an intuitive interpretation in terms of the information about an execution which is "public knowledge." We believe that it will be worthwhile to explore the connection with the formal theory of distributed knowledge [14].

The inspiration for the topological approach came from Chaudhuri's work [10], in which the combinatorial properties of triangulations in $\mathbf{R}^k$ were used to obtain certain reductions among various decision problems. There is a considerable literature concerning topologies underlying computation in general [27] and distributed computing in particular [26]. The main body of this work seems to center on the use of topology as a tool for describing various semantic constructs in distributing computing, rather than as a tool for proving impossibility results.

Two other research teams—Borowsky and Gafni [7] and Herlihy and Shavit [17]—independently discovered the topological approach to proving impossibility theorems and proved Theorem 1.1. Borowsky and Gafni [7] considered a class of protocols that is similar to our *weakly synchronous* model, and Herlihy and Shavit [17] considered *full information* protocols that bear similarities to both. The proof of Borowsky and Gafni [7] is similar to ours but does not make the explicit connection to topology. By an additional computational reduction they extend the result to prove the impossibility of k-set agreement for n processors in the presence of k faults. Herlihy and Shavit [17] developed a more general technique for proving impossibility results in this model based on simplicial homology theory. Subsequent developments along these lines include the papers [8, 11, 18].

This paper is organized as follows. In section 2, we formalize the model, the problem, and the above-mentioned notions of indistinguishable schedules and knowable sets. We also present a proof of the impossibility of (1-set) consensus using the new terminology. In section 3, we use this notation to reformulate Theorem 1.1, and we observe a close analogy between this reformulation and the aforementioned KKM lemma. In section 4, we provide an explicit bijection between the set of 2-processor schedules and the points of the closed unit interval that provides an alternative, albeit more complicated, proof of the impossibility of 2-processor consensus, and we sketch an (unproved) correspondence between n-processor schedules and the n-vertex simplex. This informal sketch motivates the combinatorial constructions discussed later. Section 5 contains an outline of the steps that we will follow to emulate the proof of the KKM lemma. Section 6 contains the proof of the main theorem. Some additional facts about the underlying topological structure in our proof are given in an appendix.

2. Definitions and preliminary results.

2.1. Input-output problems and *k*-set agreement. We fix, once and for all, the set $P = \{1, 2, \ldots, n\}$ of processors. The k-set agreement problem for P is a special case of a larger class of *input-output* problems [23, 6, 5]. In an input-output problem, each processor receives an input value from some set I and must produce an output value from some set D, where the output values must satisfy certain restrictions depending on the input. Formally, such a problem is specified by the input set I, output set D, and a relation $R \subset I^n \times D^n$.

For the k-set agreement problem, we take $I = D = \mathbf{N}$, the set of natural numbers, and the relation consists of pairs $(\vec{x}, \vec{d})$ satisfying the following: The set of values appearing in $\vec{d}$ has size at most k and is a subset of the set of values appearing in $\vec{x}$.

2.2. Informal description of the model. The results in this work will be proved for a specific formalization of the general model of asynchronous distributed computing described in the introduction. For reasons that will be apparent, we call this model the *weakly synchronous model.* This formalization was chosen because it is technically simple and is well suited to formal impossibility proofs. Informally the features of the model are as follows.

WS.1 Each register is a single-writer–multiple-reader register. The unique processor who may write to a register is referred to as the *owner* of the register.

WS.2 We assume that each process can simultaneously write to all of the registers it owns in a single atomic step. We model this by having each processor own only one register, whose set of allowed values is an arbitrary infinite set. Thus an arbitrary amount of information can be encoded in a single write.

WS.3 Each register holds an ordered list of values. When a value v is written to the register it is appended to the list rather than overwriting the existing values. Thus the register contains a record of all the writes ever done to it. At all times, the length of this list is equal to the total number of write operations that have been performed by the owner.

WS.4 When a processor performs a read operation, it reads the entire shared memory in one atomic step.

WS.5 Each processor's program consists of an infinite loop. Each iteration of the loop consists of a write to its shared register, followed by a read of the entire shared memory, followed by some arbitrary private computation.

WS.6 In solving a decision problem each processor p starts with a private initial input x_p from some input domain I. Each processor must write its decision value in the shared register that it owns. If D is the set of possible decision values, then the first element of D (if any) that the processor writes is considered to be its decision.

WS.7 The system satisfies a weak synchronicity condition. The processors execute their programs in a sequence of synchronous rounds. For each round the scheduling adversary chooses an arbitrary nonempty subset of processors to be active and each active processor executes one iteration of its loop. Since the round is synchronous all active processor writes are completed before any read begins.

We make the informal claim that our model is at least as powerful as other shared-memory models (e.g., [20, 16, 5]). It has been shown in [25] and [19, 13] that restricting to single-writer registers does not reduce the power of the model. Intuitively features 2, 3, and 4 provide more power than the standard models. Features 5 and 6 provide a "normal form" for protocols that solve input-output problems; a program in some other model can easily be converted to one of this form. Feature 7 restricts the power of the adversary by limiting the possible behaviors of the system. Note that this makes it *easier* for a protocol to be correct and thus *harder* for there to be an impossibility proof. We will not present a formal justification of these claims; the interested reader can do this for a favorite model.

2.3. Formal description of the model. A *protocol* Π for a processor set $P = \{1, \dots, n\}$ and an input domain I is referred to as a (P, I)-protocol and it is specified by a tuple (S, V, e, w, u), where

(1) S is an arbitrary set. S corresponds to the set of possible states for each processor.

(2) V is an arbitrary set. V corresponds to the set of possible values that a processor can write in a register. Thus by WS.2 each register holds an element of V^*, which is the set of finite lists of elements of V.

(3) e is a map from $I \times P$ to S. It determines the initial state of each processor from the processor's input value and the processor's ID.

(4) w is a map from S to V. It determines the value a processor will write in its register on the next step.

(5) u is a map from $S \times (V^*)^n$ to S. It determines the next state of a processor after executing a read operation. The evaluation of u corresponds to an arbitrary private computation by a processor.

A *system configuration* is a pair $(\vec{s}; \vec{l})$, where $\vec{s} = (s_1, \dots, s_n) \in S^n$ is called the *state configuration* and $\vec{l} = (l_1, \dots, l_n) \in (V^*)^n$ is called the *memory configuration*. Here s_i is the state of the ith processor and $l_i = v_1; \dots; v_k \in V^*$ is the list stored in the ith register.

A *block* J is a nonempty subset of P. The *configuration update operator* $\lhd = \lhd_\Pi$ takes as operands a system configuration and a block $J \subset P$ and produces a system configuration as follows:

$$(\vec{t}, \vec{m}) = (\vec{s}, \vec{l}) \lhd J,$$

where

$$\begin{array}{rcll} m_i &=& l_i & \text{if } i \notin J, \\ m_i &=& l_i, w(s_i) & \text{if } i \in J, \\ t_i &=& s_i & \text{if } i \notin J, \\ t_i &=& u(s_i, \vec{m}) & \text{if } i \in J. \end{array}$$

The operator corresponds to the modification of the system configuration which occurs after the synchronous execution of a program loop by the set J of processors. This corresponds to condition WS.7 in the informal description.

A *schedule* is an infinite sequence of blocks $\sigma = \sigma_1\sigma_2\sigma_3\dots$. For a nonempty $J \subseteq P$, $\Sigma(J) = \{\sigma = \sigma_1\sigma_2\dots | \sigma_i \subseteq J, \sigma_i \neq \emptyset\ \}$ denotes the set of all schedules whose blocks are subsets of J. If $p \in \sigma_i$ we say that processor p takes a step at time i. If $p \in P$ and σ is a schedule, then $steps_p(\sigma)$ is equal to the (possibly infinite) number of steps that p takes in σ. We say that p is

- *active* in σ if $steps_p(\sigma) \geq 1$, i.e., p appears in at least one block. Active(σ) is the set of active processors.
- *inactive* in σ if $steps_p(\sigma) = 0$, i.e., p appears in no block. Inactive(σ) is the set of inactive processors.
- *nonfaulty* in σ if $steps_p(\sigma)$ is infinite. Nonfaulty(σ) is the set of nonfaulty processors.
- *faulty* in σ if $steps_p(\sigma)$ is finite. Faulty(σ) is the set of faulty processors.

Observe that $\Sigma(J)$ consists of those schedules whose set of active processors is a subset of J.

A schedule is always an infinite sequence of blocks. A finite sequence of blocks is called a *fragment*. $\Phi(J)$ denotes the set of fragments whose blocks are subsets of J. The above definitions of $steps_p(\sigma)$, active, and inactive can be extended to fragments, but faulty and nonfaulty make sense only for schedules. If τ is a fragment and ϕ is a schedule or fragment, then $\tau\phi$ represents their concatenation and is a schedule or fragment. If $\sigma = \tau\phi$ we say that τ is a *prefix* of σ or that σ is an *extension* of τ.

If J is a subset of P, we denote by $[J]$ the schedule whose blocks are all equal to J. If J is equal to the singleton set $\{p\}$, we typically write $[p]$ for $[\{p\}]$.

A *run* (resp., *partial run*) is a triple $(\Pi, \vec{x}, \sigma)$, where Π is a (P, I) protocol, $\vec{x} \in I^n$ is the input, and σ is a schedule (resp., a fragment). The *execution* (resp., *partial execution*) $E = E(\Pi, \vec{x}, \sigma)$ associated to a run (resp., partial run) is defined as the infinite (resp., finite) sequence of configurations $C_0C_1C_2\dots$, where $C_0 = C_0(\Pi, \vec{x}) = (\vec{s}^{\,0}, \vec{l}^{\,0})$ is the *initial configuration* defined by $\vec{s}^{\,0} = (e(i_1, 1), \dots, e(i_n, n))$ and $\vec{l}^{\,0} = (\perp, \dots, \perp)$, where $\perp$ denotes the empty list, and $C_{i+1} = C_i \lhd \sigma_{i+1}$. The *public record* of the run or partial run $(\Pi, \vec{x}, \sigma)$ is a vector

$$Pub(\Pi, \vec{x}, \sigma) \ = \ (Pub_1(\Pi, \vec{x}, \sigma), \dots, Pub_n(\Pi, \vec{x}, \sigma)),$$

where $Pub_p(\Pi, \vec{x}, \sigma)$ is the (possibly infinite) list of all writes performed by p in the execution. Note that $Pub_p(\Pi, \vec{x}, \sigma)$ is infinite if and only if σ is a schedule and p is nonfaulty in σ. If $Pub_p(\Pi, \vec{x}, \sigma)$ is finite, then its length is the number of steps that p takes in σ.

Next, we define what it means for a protocol to *compute* a relation $R \subseteq I^n \times D^n$ for some arbitrary set D. The *D-decision value* of p on the run $(\Pi, \vec{x}, \sigma)$, denoted

$d_p^D(\Pi, \vec{x}, \sigma)$, is the first element $d \in D$ that appears on its list, or Λ(null) if none exists. If the element $d \in D$ first appears in the sth item of its list we say that p *D-decides* at step s. The vector of the D-decision values is the *D-decision vector* $\vec{d}^D(\Pi, \vec{x}, \sigma)$.

A vector $\vec{b} \in D^n$ is *compatible* with $\vec{d}$ if it can be obtained from $\vec{d}$ by replacing each Λ by some element of D. An input $\vec{x}$ is *R-permissible* if there is at least one vector $\vec{d} \in D^n$ such that $(\vec{x}, \vec{d}) \in R$. Protocol Π *computes* the relation R on schedule σ if for all R-permissible inputs $\vec{x}$

(1) the D-decision value of every nonfaulty processor is not null,

(2) there is a vector $\vec{b} \in D^n$ with $(\vec{x}, \vec{b}) \in R$ which is compatible with the decision vector $\vec{d}^D(\Pi, \vec{x}, \sigma)$.

A protocol Π is an *f-fault tolerant protocol* for R if it computes R on σ for all σ such that $|\text{Faulty}(\sigma)| \leq f$. A protocol that is $(n-1)$-fault tolerant, i.e., one that computes R on every schedule σ, is said to be *wait-free.* A protocol Π is a *bounded wait-free* protocol for R if there is a B such that for every run, each processor that takes at least B steps D-decides. It is easy to see (and is well known) that a wait-free protocol is bounded wait-free. It is also known (see [9] and the remark following Lemma 6.1 below) that for k-set consensus the existence of a wait-free protocol implies the existence of a bounded wait-free protocol.

Finally, a (P, I) protocol Π is *input-free* if the initial state of each processor depends on the processor ID only, that is, e is a map from P to S instead of from $I \times P$ to S. For an input-free protocol Π we write (Π, σ) for the run or partial run, $E(\Pi, \sigma)$ for the execution or partial execution, and $Pub(\Pi, \sigma)$ for the public record. Intuitively, input-free protocols are obtained from arbitrary ones by "hardwiring" a specific set of inputs to the individual processors.

PROPOSITION 2.1. *Let Π be a (P, I) protocol and let $\vec{x} \in I^n$ be some fixed input vector. Then there exists an input-free protocol Π' such that for all $\sigma \in \Sigma(P) \cup \Phi(P)$,*

$$E(\Pi, \vec{x}, \sigma) = E(\Pi', \sigma)$$

2.4. Impossibility of consensus. For illustration purposes we show how to adapt the proof of the consensus impossibility result [15, 16, 21] to prove that wait-free consensus is impossible in the weakly synchronous model. The previous proofs in the literature give a stronger result: there is no consensus protocol that is even 1-fault tolerant. It is possible to strengthen the following proof to give this result, but we do not do this here. This section is not needed for the development of the rest of the paper.

THEOREM 2.2. *In the weakly synchronous model, for $n > 1$ there is no deterministic wait-free protocol for n-processor consensus.*

Proof. Assume the set of possible inputs is $I = \{a, b\}$. Suppose, for contradiction, that Π is a protocol that solves consensus for all $\vec{x} \in I^n$ and all schedules σ. Each run $(\Pi, \vec{x}, \sigma)$ may be classified uniquely as a-deciding or b-deciding depending on the decision value.

PROPOSITION 2.3. *There is an input vector $\vec{y}$ and two schedules σ and ϕ such that $(\Pi, \vec{y}, \sigma)$ is a-deciding and $(\Pi, \vec{y}, \phi)$ is b-deciding.*

Proof. Let $\vec{y} \in I^n$ be an input vector with the minimum number of a's such that there exists a schedule σ such that $(\Pi, \vec{y}, \sigma)$ is a a-deciding; $\vec{y}$ has at least one a in it, say, in coordinate p. Define ϕ to be the schedule with repeated blocks $P - \{p\}$. We claim that $(\Pi, \vec{y}, \phi)$ is b-deciding. Let $\vec{w}$ be the vector obtained from $\vec{y}$ by changing the entry in coordinate p to b. Both of the runs $(\Pi, \vec{y}, \phi)$ and $(\Pi, \vec{w}, \phi)$ have the same public record. Furthermore the latter is b-deciding since $\vec{w}$ has fewer a's from $\vec{y}$ which was selected to have the minimum possible such number. Therefore $(\Pi, \vec{y}, \phi)$ is also b-deciding. $\square$

We restrict our attention to runs with input $\vec{y}$ given by Proposition 2.3. We say that a fragment τ is a-valent (resp., b-valent) if for every schedule ρ, $(\Pi, \vec{y}, \tau\rho)$ is a-deciding (resp., b-deciding). It is *bivalent* if it is neither a-valent nor b-valent, i.e., if there are schedules ρ and μ such that $(\Pi, \vec{y}, \tau\rho)$ is an a-deciding and $(\Pi, \vec{y}, \tau\mu)$ is a b-deciding. Clearly, no processor reaches a decision on the partial execution corresponding to a bivalent fragment. By choice of $\vec{y}$, the empty fragment is *bivalent.*

LEMMA 2.4. *For any bivalent fragment τ there exists a block J such that the fragment τJ is bivalent.*

Proof. Let τ be a bivalent fragment. Assume for contradiction that τJ is not bivalent for any block J. Thus for each J, τJ is either a- or b-valent. Without loss of generality suppose that τP (where P is the set of all processors) is a-valent. We will show that τJ is a-valent for every J in P which would imply τ is a-valent, a contradiction. Let J be arbitrary and $\bar{J} = P - J$. It is easy to see that the state of the shared memory and the internal states of processors in $\bar{J}$ are identical for the partial executions $PE(\Pi, \vec{y}, \tau P)$ and $PE(\Pi, \vec{y}, \tau J\bar{J})$. (Note, however, that the internal states of processors in J may differ between the two partial executions.) Recall that $[\bar{J}]$ denotes the schedule consisting of repeated $\bar{J}$ blocks. Then schedules $\tau J\bar{J}[\bar{J}]$ and $\tau P[\bar{J}]$ have identical public records. Since by our assumption τJ is not bivalent it must be a-valent, giving the desired contradiction. $\square$

By using the above lemma, one can construct an (infinite) schedule such that any prefix is bivalent, which contradicts that Π is a wait-free algorithm for consensus. $\square$

2.5. Tallies and the counting protocol. We now introduce a specific protocol, called the *counting protocol,* which will play a special role in our analysis. This is an input-free protocol which we denote by Γ.

To describe this protocol, we need to introduce the notion of a *tally vector*, which is a vector indexed by P whose entries are nonnegative integers. For a tally vector v, and $p \in P$, we write $v[p]$ for the $[p]$ entry of v. We define the partial order on tally vectors with $v \leq w$ if $v[p] \leq w[p]$ for all $p \in P$. Two tally vectors that are comparable under this ordering are said to be *noncrossing* and they are *crossing* otherwise. If τ is a fragment, the *tally* of τ, denoted $t(\tau)$, is the tally vector such that $t(\tau)[q]$ is equal to the number of steps q takes in τ. If $\sigma = \sigma_1, \sigma_2, \ldots$ is a schedule or fragment, the *tally sequence* associated with σ, denoted $\mathrm{T}(\sigma)$, is the sequence $T_0, T_1, T_2, \ldots$ of tally vectors, where T_i is the tally of the fragment $\sigma_1\sigma_2\ldots\sigma_i$. For example, the fragment $\{1,2\}\{3\}\{1,3\}\{1,2,3\}\{1\}\{2\}$ has tally sequence $(0,0,0), (1,1,0), (1,1,1), (2,1,2), (3,2,3), (4,2,3), (4,3,3)$. Observe that the tally vectors in the tally sequence of σ are noncrossing and that σ is trivially determined by $\mathrm{T}(\sigma)$.

We can now define the counting protocol Γ. As stated before, it takes no input. The state of each processor p is a tally vector $tally_p$. Initially all entries of $tally_p$ are 0. Each time p executes a step, p writes (appends) $tally_p$ to its public register. It then reads all of the shared registers and sets $tally_p$ so that $tally_p[q]$ is equal to the number of steps that have been taken by processor q (which is equal to the length of the list in processor q's public register). We define $Count(\sigma)$ to be the public record, $Pub(\Gamma, \sigma)$, of the counting protocol on schedule or fragment σ and call this the *public tally* of σ. Thus for each $p \in P$, $Count_p(\sigma)$ is a (possibly infinite) list of length $steps_p(\sigma)$, where each element of the list is itself a vector indexed by P. For $i \leq \mathit{steps}_p(\sigma)$, we denote by $\mathit{Count}_{p,i}(\sigma)$ the vector corresponding to the ith write by p, and for $q \in P$, $\mathit{Count}_{p,i}(\sigma)[q]$ is the value of this vector in position q. For example, on the fragment $\{1,2\}\{3\}\{1,3\}\{1,2,3\}\{1\}\{2\}$, we have

$$\begin{aligned}
\mathit{Count}_1(\sigma) &= (0,0,0);(1,1,0);(2,1,2);(3,2,3),\\
\mathit{Count}_2(\sigma) &= (0,0,0);(1,1,0);(3,2,3),\\
\mathit{Count}_3(\sigma) &= (0,0,0);(1,1,1);(2,1,2).
\end{aligned}$$

Note that the ith write by processor p is $t(\tau)$, where τ is the prefix up to the $(i-1)$st step of p. Note also that the final values of the internal states of the processors, tally_1, tally_2, and tally_3, are, respectively, $(4,2,3)$, $(4,3,3)$, and $(3,2,3)$, which do not appear in the public record.

By definition, each tally that appears in the public tally of σ also appears in its tally sequence $\mathrm{T}(\sigma)$. In particular, the set of all tally vectors that appear in the public tally is noncrossing. We will need the following lemma.

Lemma 2.5. *Let σ and ϕ be two schedules such that $\mathit{Count}(\sigma) \neq \mathit{Count}(\phi)$. Then at least one of the following holds:*

(1) *There exists a processor p and an integer i such that p takes at least i steps in both σ and ϕ and the tally vectors $\mathit{Count}_{p,i}(\sigma)$ and $C_{p,i}(\phi)$ are different.*

(2) *There exists a pair of crossing tally vectors v and w so that v appears in $\mathit{Count}(\sigma)$ and w appears in $\mathit{Count}(\phi)$.*

Proof. First consider the case that each processor takes exactly the same number of steps in σ as in ϕ. Then since $\mathit{Count}(\sigma) \neq \mathit{Count}(\phi)$, the first conclusion must hold.

Next, suppose that some processor p takes a total of i steps in σ and takes at least $i+1$ steps in ϕ. Let r be a processor that takes infinitely many steps in ϕ and let w be a tally vector written by r on schedule ϕ such that $w(p) \geq i+1$. Such a w must exist since p takes at least $i+1$ steps and r takes infinitely many steps. Let q be a processor that takes infinitely many steps in σ and let v be the tally vector written by q on schedule σ at its $w(q)+2$ step. Then $v(q) = w(q)+1$ and $v(p) \leq i < w(p)$, and so v and w are crossing tally vectors. ◻

2.6. Indistinguishable schedules. Two schedules or fragments σ and τ are *publicly indistinguishable* if for any protocol Π and input vector $\vec{x}$, the public records $\mathit{Pub}(\Pi, \vec{x}, \sigma)$ and $\mathit{Pub}(\Pi, \vec{x}, \tau)$ are the same. Intuitively, this says that there is no protocol Π and input $\vec{x}$ that will enable an "outside observer" looking at the "final" lists in the registers to distinguish between the schedules σ and τ. This is clearly an equivalence relation on the set all schedules and fragments. The structure of this equivalence relation is fundamental to the proofs of our results.

If σ and τ are publicly indistinguishable, then they must have the same public tallies, i.e., $\mathit{Count}(\sigma) = \mathit{Count}(\tau)$. As we will see in Theorem 2.12, this condition is also sufficient for public indistinguishability. This theorem will also provide another combinatorial characterization based on a notion called *compression*. To introduce this notion we will need some additional definitions.

If τ is a fragment, its length $|\tau|$ is the number of blocks in it, and its *weight*, $w(\tau)$, is the sum of the block sizes. Associated to each schedule or fragment σ is its *site sequence* $s(\sigma) = (s_1, s_2, \ldots)$ of length $|\sigma|$, where s_i is the weight of the first i blocks. We will also refer to s_i as the *site* of the ith block of σ.

Example 2.6. *$P = \{1,2,3\}$ and τ is the fragment $\{1,2,3\}\{1,2\}\{2,3\}\{2\}\{3\}$. Then $|\tau| = 5$, $w(\tau) = 9$, and $s(\tau) = (3,5,7,8,9)$.*

Let σ be a schedule or fragment. Then a block σ_i of σ is *hidden* if σ_i is not the last block and no processor in σ_i appears in any later block. Note that any two hidden blocks are necessarily disjoint and that whether σ is a schedule or a fragment, at least one processor belongs to no hidden block. Thus we have the following proposition.

PROPOSITION 2.7. *A schedule or fragment σ has at most $|P|-1$ hidden blocks.*

A schedule or fragment that has no hidden blocks is said to be *compressed*.

We now define operators for "eliminating" hidden blocks. For a positive integer s, the *merge* operator M_s is defined as follows. If σ is a schedule or fragment, $M_s(\sigma)$ is the schedule or fragment obtained as follows: if there is a block σ_i at site s and the block is hidden, then replace σ_i and σ_{i+1} by their union; otherwise, $M_s(\sigma) = \sigma$. The following facts are easy to verify.

PROPOSITION 2.8.

(1) *If σ is compressed, then $M_s(\sigma) = \sigma$ for all s.*
(2) *The operators M_s and M_r commute for all integers r and s.*
(3) *If σ is a schedule or fragment and $r_1, r_2, \ldots, r_k$ are the sites of its hidden blocks, then $M_{r_1}M_{r_2}\ldots M_{r_k}(\sigma)$ is a compressed sequence.*

The compressed sequence obtained from σ in the third part of Proposition 2.8 is called the *compression* of σ and is denoted $\hat{\sigma}$. More generally, a sequence τ which can be obtained from σ by application of some sequence of merge operators is said to be a *partial compression* or σ. An easy consequence of Proposition 2.8 is the following.

COROLLARY 2.9. *If τ is a partial compression of σ, then $\hat{\tau} = \hat{\sigma}$. Thus $\hat{\sigma}$ is the unique compressed sequence that can be obtained from σ by applying merge operators.*

The compression map $\sigma \longrightarrow \hat{\sigma}$ defines an equivalence relation on $\Sigma(P) \cup \Phi(P)$: σ and τ are *compression equivalent* if $\hat{\sigma} = \hat{\tau}$. The equivalence class of σ is called the *compression class* of σ and is denoted $\langle\sigma\rangle$.

Let σ be a compressed schedule, let χ be the smallest prefix of σ containing all faulty processors, and write $\sigma = \chi\phi$. Then any schedule that compresses to σ is of the form $\tau\phi$, where τ is a fragment of the same weight as χ. In particular, this implies the following.

PROPOSITION 2.10. *The compression class $\langle\sigma\rangle$ of any schedule is finite.*

EXAMPLE 2.11. *Suppose $P = \{1,2,3,4\}$ and let σ be the compressed schedule $\{1,2\}\{1,3,4\}\{1,2,3\}[3] = \{1,2\}\{1,3,4\}\{1,2,3\}\{3\}\{3\}\ldots$. There are eleven uncompressed schedules whose compression is σ:*

$$\begin{aligned}
\sigma^1 &= \{1,2\}\{1,3,4\}\{1\}\{2,3\}[3],\\
\sigma^2 &= \{1,2\}\{1,3,4\}\{2\}\{1,3\}[3],\\
\sigma^3 &= \{1,2\}\{1,3,4\}\{1\}\{2\}[3],\\
\sigma^4 &= \{1,2\}\{1,3,4\}\{2\}\{1\}[3],\\
\sigma^5 &= \{1,2\}\{1,3,4\}\{12\}[3],\\
\sigma^6 &= \{1,2\}\{4\}\{1,3\}\{1,2,3\}[3],\\
\sigma^7 &= \{1,2\}\{4\}\{1,3\}\{1\}\{2,3\}[3],\\
\sigma^8 &= \{1,2\}\{4\}\{1,3\}\{2\}\{1,3\}[3],\\
\sigma^9 &= \{1,2\}\{4\}\{1,3\}\{1\}\{2\}[3],\\
\sigma^{10} &= \{1,2\}\{4\}\{1,3\}\{2\}\{1\}[3],\\
\sigma^{11} &= \{1,2\}\{4\}\{1,3\}\{1,2\}[3].
\end{aligned}$$

In σ^{10}, for instance, the hidden blocks are at sites 3, 6, *and* 7. *Applying M_6 yields σ^{11}, and then applying M_7 yields σ^6, and applying M_3 yields σ.*

The following result characterizes public indistinguishability. Recall the definition of the public tally vector $Count(\sigma)$ from the previous section as the public record of the counting protocol Γ.

THEOREM 2.12. *Let σ and τ be schedules or fragments. Then the following are equivalent:*

(1) *σ is publicly indistinguishable from τ,*

(2) *$Count(\sigma) = Count(\tau)$,*

(3) *$\hat{\sigma} = \hat{\tau}$.*

Proof. (3) $\Rightarrow$ (1). Assume that $\hat{\sigma} = \hat{\tau}$; we will show that σ and τ are publicly indistinguishable.

LEMMA 2.13. *σ is publicly indistinguishable from $M_s(\sigma)$ for all s.*

Proof. The result is trivial if $\sigma = M_s(\sigma)$, so assume they are distinct. Then σ has a hidden block σ_k at site s and

$$\begin{array}{rcll} \phi_i & = & \sigma_i & \text{if } i < k, \\ \phi_k & = & \sigma_k \cup \sigma_{k+1}, & \\ \phi_i & = & \sigma_{i+1} & \text{if } i > k. \end{array}$$

Let $C(\sigma, i)$ be the configuration of the protocol Π on schedule σ after the execution of i blocks. Clearly $C(\phi, i) = C(\sigma, i)$ for every $i < k$. The configuration $C(\sigma, k+1)$ and $C(\phi, k)$ have exactly the same memory configuration but they may differ on the state configuration of the processors in the set σ_k. But these processors will not execute another step later in any of the schedules. Therefore, for $i > k$, $C(\phi, i)$ differs from $C(\sigma, i+1)$ only in the state of the processors in σ_k. Therefore σ and ϕ are publicly indistinguishable. □

COROLLARY 2.14. *σ is publicly indistinguishable from $\hat{\sigma}$.*

Proof. Let $r_1, r_2, \ldots, r_k$ be the sites of the hidden blocks of σ. Then by proposition 2.8, $\hat{\sigma} = M_{r_1} M_{r_2} \ldots M_{r_k}(\sigma)$ and the result follows by applying the previous lemma and induction. □

Therefore σ is publicly indistinguishable from $\hat{\sigma}$ and τ is publicly indistinguishable from $\hat{\tau}$. Since $\hat{\sigma} = \hat{\tau}$, then σ and τ are publicly indistinguishable.

(1) $\Rightarrow$ (2). This is trivial since if σ is publicly indistinguishable from τ, then by definition every protocol, including Γ, has the same public record on σ as τ.

(2) $\Rightarrow$ (3). Let σ and τ be schedules such that $Count(\sigma) = Count(\tau)$. From Corollary 2.14 and the fact that (1) $\Rightarrow$ (2) we have $Count(\sigma) = Count(\hat{\sigma})$ and $Count(\tau) = Count(\hat{\tau})$, hence $Count(\hat{\sigma}) = Count(\hat{\tau})$.

Now suppose for contradiction that $\hat{\sigma} \neq \hat{\tau}$. Write $\phi = \hat{\sigma}$ and $\mu = \hat{\tau}$ and consider the least k such that $\phi_k \neq \mu_k$. Let ϕ' and μ' be the prefixes ending with ϕ_k and μ_k, respectively. Then the tally vectors $t(\phi')$ and $t(\mu')$ are different; we may assume that either $t(\phi') < t(\mu)$ or $t(\phi')$ and $t(\mu')$ are crossing vectors (defined in section 2.5). Then the vector $t(\phi')$ does not appear in the tally sequence of μ and does not appear in $Count(\mu)$. On the other hand, since ϕ is compressed, we may choose a processor q in ϕ_k that writes again. Its next write in the counting protocol will be $t(\phi')$, contradicting that $Count(\phi) = Count(\mu)$.

This completes the proof of Theorem 2.12. □

2.7. Quasi extensions of fragments. We have defined the notion of a public record associated to an execution as the vector $\vec{R}$ indexed by p, where the entry $\vec{R}_p$ is the list of all writes done by p during the execution. It is convenient to call any such vector $\vec{R}$ indexed by P, where the entry $\vec{R}_p$ is an arbitrary list of elements, a *public record*. A public record is *finite* if each list is finite and infinite otherwise. If $\vec{R}$ and $\vec{R}'$ are public records, then we say that $\vec{R}'$ is an extension of $\vec{R}$ if each of the lists $\vec{R}_p$ is a prefix of the corresponding list $\vec{R}'_p$.

Now if τ is a fragment which is a prefix of the schedule or fragment σ, then clearly the following condition holds:

(QE) For any protocol Π and input $\vec{x}$, the public record of the execution $(\Pi, \vec{x}, \sigma)$ is an extension of the public record of the execution $(\Pi, \vec{x}, \tau)$.

We say that a schedule or fragment σ is a *quasi extension* of the fragment τ if condition (QE) holds. For a fragment τ, Q_τ denotes the set of schedules that are quasi extensions of τ.

Condition (QE) does not imply that τ is a prefix of σ. For instance, we have the following.

LEMMA 2.15. *Let ρ be a fragment, ϕ a schedule or fragment, and $Y \subseteq Active(\phi)$. If $\sigma = \rho\phi$ and $U \subseteq Active(\phi)$, then σ quasi-extends ρU.*

Proof. For any protocol, the public records of $\rho\phi$ and ρU trivially agree up through the end of ρ. Now in ρU, each of the processors in U write once more, and they write the view observed during their last step in ρ. But for each $p \in U$, the same write will occur when executing $\rho\phi$ since $p \in Active(\phi)$. $\square$

We have the following combinatorial characterization of quasi extension, which is analogous to Theorem 2.12.

THEOREM 2.16. *Let μ be a fragment and let σ be a schedule or a fragment. Then the following are equivalent:*

(1) *σ is a quasi extension of μ.*

(2) *$Count(\sigma)$ extends $Count(\mu)$.*

(3) *There exists a prefix ρ of σ, a schedule or fragment ϕ, and a subset U of $Active(\phi)$ such that $\sigma = \rho\phi$ and $\hat{\mu} = \widehat{\rho U}$.*

Proof. (1) $\Rightarrow$ (2). This follows from the definition of quasi extension and the fact that $Count(\sigma)$ and $Count(\mu)$ are the respective public records arising from a protocol.

(2) $\Rightarrow$ (3). Write $\hat{\mu}$ as τU, where U is the last block of $\hat{\mu}$. By hypothesis, and the fact that μ is publicly indistinguishable from $\hat{\mu}$, we have that $Count(\sigma)$ extends $Count(\tau U)$. Since τU is compressed, there is a processor $p \in U$ that also appears in the last block of τ. Let i be the number of steps that p makes in τ. By definition of the counting protocol, $Count_{p,i+1}(\tau U) = t(\tau)$, where $t(\tau)$ is the tally vector associated with fragment τ. By hypothesis, $Count_{p,i+1}(\sigma) = Count_{p,i+1}(\tau U)$. Let ρ be the minimal prefix of σ containing the first i steps of p. Again, by the definition of the counting protocol, $Count_{p,i+1}(\sigma) = t(\rho)$. Hence $t(\tau) = t(\rho)$. Write $\sigma = \rho\phi$. Now, since $Count(\rho\phi)$ extends $Count(\tau U)$ it is clear that each processor in U must take a step in ϕ, so $U \subseteq Active(\phi)$. Finally, we claim that $\widehat{\rho U} = \tau U$, and for this it is enough, by Theorem 2.12, to show that $Count(\rho U) = Count(\tau U)$. By Lemma 2.15, $Count(\rho\phi)$ extends $Count(\rho U)$ and since $Count(\rho\phi)$ also extends $Count(\tau U)$ (by hypothesis) and every processor takes the same number of steps in ρU as in τU, we must have $Count(\rho U) = Count(\tau U)$, as required.

(3) $\Rightarrow$ (1). Assume that (3) holds. By Lemma 2.15, σ is a quasi extension of ρU. But since $\widehat{\rho U} = \hat{\mu}$, ρU and μ are publicly indistinguishable and so σ is also a quasi extension of μ. $\square$

2.8. Knowable sets. For a set of schedules S, let $\hat{S} = \{\hat{\sigma} | \sigma \in S\}$. In particular, $\hat{\Sigma}$ is the set of all compressed schedules. By Theorem 2.12, to check that a protocol for an input-output problem is correct, it suffices to check it for schedules in $\hat{\Sigma}$.

In this subsection, we are interested in the dependence of the public record of a run $(\Pi', \vec{x}, \sigma)$ on σ while holding Π' and $\vec{x}$ fixed. Therefore it is easier to consider the corresponding input-free protocol Π given by Proposition 2.1.

In an input-free protocol, each processor's decision depends only on the schedule. One can view the computational steps taken by the processors as "collecting information" about the schedule. When a processor "knows enough" about the schedule, it can make a decision. Let $K(\Pi, d)$ be the set of all compressed schedules on which some processor running the input-free protocol Π writes d on its list. The pair (Π, d) is an *acceptor* and d is its *accepting* value. We say that (Π, d) *accepts schedule* σ if $\sigma \in K(\Pi, d)$. A set $K \subseteq \hat{\Sigma}$ is *publicly knowable* or simply *knowable* if it equals $K(\Pi, d)$ for some acceptor (Π, d). Below we give several examples. In each example, $d =$ "@."

EXAMPLE 2.17. *$\hat{\Sigma}$ is knowable. If Π is the protocol where every processor writes* "@" *at every opportunity, then $\hat{\Sigma} = K(\Pi, @)$.*

EXAMPLE 2.18. *$\emptyset$ is knowable. If Π is the protocol where every processor writes* "0" *at every opportunity, then $K(\Pi, @) = \emptyset$.*

EXAMPLE 2.19. *For each integer k and processor p, let $S_{p,k}$ be the set of compressed schedules in which processor p takes at least k steps. It is easy to see that $S_{p,k}$ is knowable; a simple input-free protocol that accepts this set is the one where processor p writes* "0" *for its first $k - 1$ steps and* "@" *thereafter, and all other processors always write* "0."

EXAMPLE 2.20. *Let S_i be the set of schedules where every processor takes at least i steps. S_i is a knowable set. The protocol that accepts the set is as follows: Each processor has two possible states, "continue" and "accept." While in the "continue" state, each processor appends* "0" *to the list in its shared register, reads the shared memory, and enters the "accept" state if all of the processors took i steps (have output list of length at least i). Once the processor is in the "accept" state, it writes* "@."

EXAMPLE 2.21. *Let τ be any fragment. The set $\hat{Q}_\tau$ of compressed schedules that are quasi extensions of τ is knowable. Define the following modification of the counting protocol defined in section* 2.5*: if any processor ever observes that the public record is an extension of $Count(\tau)$, then it writes* "@" *at its next opportunity. When this protocol is run on an (infinite) compressed schedule σ,* "@" *is written if and only if $Count(\sigma)$ is an extension of $Count(\tau)$. By Theorem* 2.16 *this is equivalent to $\sigma \in \hat{Q}_\tau$.*

For contrast, we give some examples of sets that are not knowable.

EXAMPLE 2.22. *Let $T_{p,k}$ be the set of compressed schedules where processor p takes* exactly *k steps for some k. Then $T_{p,k}$ is* not *a knowable set. Suppose to the contrary that (Π, d) is an acceptor for $T_{p,k}$. Let $\sigma \in T_{p,k}$ be arbitrary; then d is written in the public record of the run (Π, σ). Now the first write of (a, "d") occurs at some finite block σ_m of σ, so if we define $\phi = \sigma_1, \sigma_2, \ldots, \sigma_m, \{p\}, \{p\}, \ldots,$ then ϕ is also accepted by (Π, d). Then $\hat{\phi}$ is a compressed schedule accepted by (Π, d) but not in $T_{p,k}$, a contradiction.*

EXAMPLE 2.23. *The complement of a knowable set need not be knowable, e.g., consider the complement of $S_{p,k}$ and apply an argument analogous to the previous one.*

EXAMPLE 2.24. *Let N_p be the set of compressed schedules that do not begin with $\{p\}$. Then N_p is not knowable. Suppose to the contrary that (Π, d) is an acceptor for N_1. Let $\sigma \in N_1$ be the schedule $\{1, 2\}, \{2\}, \{2\}, \ldots.$ Then there is a block σ_k of σ such that d is first written in the public record of the run (Π, σ). If*

$\phi = \{1\}, \{2\}^k, \{1, 2\}\{2\}, \{2\}, \{2\}, \ldots,$ *then the public record corresponding to its first* $k+1$ *blocks will match the public record of the first* k *blocks of* σ*. Thus* ϕ *is compressed, is not in* N_1*, and is accepted by* (Π, d)*, a contradiction.*

The last example illustrates an important point: it is not hard to construct a protocol such that for any schedule σ in N_p, the fact "$\sigma \in N_p$" is recorded in the local state of at least one processor q. (Each processor $q \neq p$ records "yes" in its local state, if its first read does not see a write by p, and p records "yes" in its local state if its first read sees at least one write other than its own.) However, in this case, if p appears in the first block and never writes again, this fact does not become "public." The term *publicly knowable* reflects the fact that the value v is written in the memory and thus every nonfaulty processor "eventually knows" that the schedule belongs to K.

An acceptor (Π, d) is said to accept a *fragment* τ if d appears in the public record of Π on τ. The definition of a run of a protocol implies the following.

Proposition 2.25. *Let* K *be a knowable set. Then a compressed schedule* σ *belongs to* K *if and only if for some fragment* τ *of* σ, $\hat{Q}_\tau \subseteq K$*.*

The concept of knowable set allows us to reformulate the impossibility result for a k-set agreement problem. We do this in the next section.

3. Reformulating Theorem 1.1 and a topological analogy.

We want to prove that, in the weakly synchronous model, there is no wait-free protocol that solves the k-set agreement problem in a system of n processors for any $k < n$. Clearly it suffices to prove the impossibility result for $k = n - 1$. As a first step in the proof of Theorem 1.1, we use the notation of the previous section to reformulate it.

Assume, for contradiction, that there exists a wait-free protocol Π that solves the $(n-1)$-set agreement problem for a processor set $P = \{1, \ldots, n\}$. Consider the behavior of Π on the input $\vec{x} = (1, \ldots, n) \in I^n$. For each $i \in P$, let D_i be the set of compressed schedules σ such that on the run $(\Pi, \vec{x}, \sigma)$ at least one processor reaches decision i. By definition, each D_i is a knowable subset of $\hat{\Sigma}$. Also, if Π is correct, then for any schedule σ, at least one processor decides some value in $\{1, \ldots, n\}$, so the sets $D_1, D_2, \ldots, D_n$ together cover $\hat{\Sigma}$. An arbitrary sequence $A_1, A_2, \ldots, A_n$ of subsets of $\hat{\Sigma}$ *satisfies the activity property* if for each $p \in P$, processor p is active in each $\sigma \in A_p$. In other words A_p is disjoint from $\hat{\Sigma}(P - \{p\})$.

Proposition 3.1. *If* Π *is a fully fault-tolerant protocol for* k*-set agreement, then the sequence of sets* $D_1, D_2, \ldots, D_n$ *satisfies the activity property.*

Proof. Assume for the sake of contradiction that for some schedule $\sigma \in D_p$, processor p is not active. Consider a run of protocol Π on σ with the input vector $\vec{y}$ defined by $y_p = n + 1$ and $y_q = x_q$ for $q \neq p$. Then $Pub(\Pi, \vec{x}, \sigma) = Pub(\Pi, \vec{y}, \sigma)$; therefore, on input $\vec{y}$ some processor decides p although p does not appear in $\vec{y}$, which violates the validity condition. □

Our main theorem will thus follow from the following general result about knowable sets.

Theorem 3.2. *If* $K_1, \ldots, K_n$ *is a collection of knowable subsets of* $\hat{\Sigma}(P)$ *that cover* $\hat{\Sigma}$ *and satisfy the activity property, then*

$$\bigcap_{p=1}^{n} K_p \neq \emptyset.$$

Applying this to the sets $D_1, D_2, \ldots, D_n$, we obtain that there is a single schedule σ in which every possible decision $1, 2, \ldots, n$ is reached, contradicting that Π solves the $(n-1)$-set agreement problem.

TABLE 1
The syntactic correspondence between $\mathbf{R}^n$ *and knowable set topologies.*

$Hull(E^P)$	$\longleftrightarrow$	$\hat{\Sigma}(P)$
point $\vec{z} \in Hull(E^P)$	$\longleftrightarrow$	compressed schedule σ
relatively open subset of $Hull(E^P)$	$\longleftrightarrow$	knowable subset of $\hat{\Sigma}(P)$
$Hull(E^{P-\{p\}})$	$\longleftrightarrow$	$\hat{\Sigma}(P - \{p\})$
boundary property	$\longleftrightarrow$	activity property
vertex $\vec{e}^p$	$\longleftrightarrow$	schedule $[p] = \{p\}\{p\}\dots$

There is a striking analogy between the statement of Theorem 3.2 and a well-known theorem concerning the topology of Euclidean space. Let $Hull(Z)$ denote the convex hull of a set of points Z in the Euclidean space. Let $E^P = \{\vec{e}^p | p \in P\}$ be the set of standard unit basis vectors of $\mathbf{R}^P$, which we identify with $\mathbf{R}^n$. Note that $Hull(E^P)$ is the $(n-1)$-dimensional simplex consisting of the set of nonnegative vectors whose coordinates sum to 1. We say that a sequence of subsets $A_1, A_2, \dots, A_n$ of $Hull(E^P)$ *satisfies the boundary property* if for each $p \in \{1, \dots, n\}$, A_p is disjoint from $Hull(E^{P-\{p\}})$, which is the face of $Hull(E^P)$ opposite the vertex $\vec{e}^p$, i.e, each vector $\vec{z} \in A_p$ has positive pth coordinate. Finally, a subset U is *relatively open* in $Hull(E^P)$ if it is the intersection of $Hull(E^P)$ with an open subset of $\mathbf{R}^n$.

The following theorem is essentially equivalent to the Brouwer fixed point theorem for the $(n-1)$-dimensional closed ball.

THEOREM 3.3 (KKM theorem; see [1]). *If $U_1, \dots, U_n$ is a collection of relatively open subsets of $Hull(E^P)$ that cover $Hull(E^P)$ and satisfy the boundary property, then*

$$\bigcap_{i=1}^{n} U_i \neq \emptyset.$$

There is a tight syntactic correspondence between the two situations described by Theorems 3.2 and 3.3, which is given in Table 1.

The obvious question is, Is there some way to make use of this correspondence to prove the desired result for knowable sets? The natural way to do this would be to find a bijection between $\hat{\Sigma}(P)$ and $Hull(E^P)$ which obeys the syntactic correspondence. Given such a bijection Theorem 3.2 would follow from Theorem 3.3. In fact, we believe that there is such a bijection and that we have an existential argument for this. But the technical details involved in turning this argument into a rigorous proof seem to be considerable and except for the case $n = 2$, we have not completed such a proof.

It turns out we don't really need such a bijection. We prove Theorem 3.2 directly by analyzing and imitating the proof of Theorem 3.3. Nevertheless, the ideas of the proof are based on the intuition we developed in trying to construct an appropriate bijection between $\hat{\Sigma}(P)$ to $Hull(E^P)$. In the next section, we discuss some of these intuitions and give an explicit bijection for the $n = 2$ case (which corresponds to the impossibility of 2-processor consensus), a not-so-explicit bijection for the $n = 3$ case, and a glimpse at the $n > 3$ case. While our proofs do not explicitly depend on this section, it provides the key intuitions which motivate the succeeding sections.

4. Bijections between $\hat{\Sigma}(P)$and $Hull(E^P)$.

As described in the previous section, the most natural way to prove our result would be to provide a bijection f obeying the syntactic correspondence. For $n = 2$ we can explicitly construct such a map. For higher dimensions we have no explicit map, although we are fairly certain that the facts we develop later could, if one wanted, be used to prove existence of such a map. We emphasize that the proofs of our results do not rely on this section, and so we freely make claims here without proof.

The conditions we need on a bijection between $\hat{\Sigma}(P)$ and $Hull(E^P)$ are

(A) for $J \subseteq P$, each compressed schedule $\sigma \in \hat{\Sigma}(P)$ is mapped to the simplex $Hull(\{e^i : i \in J\})$;

(B) the image of any knowable set of $\hat{\Sigma}(P)$ is a relatively open subset of $Hull(E^P)$.

The first condition is fairly explicit; it says, for instance, that schedules of the form $(p)(p)(p)\dots$ must be mapped to a corner vertex $\vec{e}^{\,p}$. The second relies on the notion of knowable sets, whose definition in terms of protocols is hard to deal with directly. We now state a combinatorial characterization of knowability. We don't prove this now, but we note that it is equivalent to the characterization proved below as Theorem A.4. If τ is a schedule fragment, the *cylinder* of τ, B_τ is the collection of all schedules that have τ as a prefix.

THEOREM 4.1. *A set S of compressed schedules is knowable if and only if the set $\{\sigma \in \Sigma(P) | \hat{\sigma} \in S\}$ can be expressed as a (possibly infinite) union of cylinders.*

Now suppose we can find a function f whose domain is the set $\Sigma(P)$ of all schedules (not just compressed ones), such that f satisfies the following four conditions:

(1) f maps $\Sigma(P)$ onto $Hull(E^P)$.

(2) Each schedule σ is mapped to the simplex $Hull(\{e^i : i \in J\})$, where $J = Active(\sigma)$.

(3) $f(\sigma) = f(\tau)$ if and only if σ and τ have the same compression.

(4) f maps schedules with a "large" common prefix to points that are "close." More precisely, there exists a function $\alpha(j)$ on the nonnegative integers that tends to 0 such that for any two schedules σ and ϕ, if σ and ϕ have a common prefix of j blocks, then $\|f(\sigma) - f(\phi)\| \leq \alpha(j)$, where $\|\cdot\|$ denotes the usual Euclidean length.

The second condition is just condition (A) above. It is not hard to show that conditions (1), (3), and (4) together with Theorem 4.1 imply condition (B). Note that given a map f satisfying (1) and (3), its restriction to the set $\hat{\sigma}$ of compressed schedules is a bijection.

The following definitions will be useful. A schedule σ is *degenerate* if it has a unique nonfaulty processor. Such a schedule is uniquely of the form $\tau[p]$ for some segment τ and processor p, where either τ is the null fragment or the last block B of τ is not equal to $\{p\}$. Writing τ as μB, we say that μ is the *fundamental fragment* of σ and B is the fundamental block. We also say that σ is p-degenerate. A degenerate schedule whose fundamental fragment has weight at most w is said to be *w-admissible.*

We now describe such a function f for the $n = 2$ case and give some indication how it can be extended to the $n \geq 3$ case.

4.1. A bijection for the 2-processor case.

For simplicity, in the description of f, we consider the range to be the interval $[0, 1]$ instead of $Hull(E^{\{1,2\}})$.

Let us start by describing a mapping that does not work. Take each schedule σ and interpret it as an infinite ternary string $t = t(\sigma)$ by mapping $\{1\}$ to 0, $\{1, 2\}$ to 1, and $\{2\}$ to 2. Then any schedule can be interpreted as a real number between 0 and 1. Furthermore, this mapping is easily seen to satisfy properties (1), (2), and (4) above: the map is onto, it sends the schedule $[1] = \{1\}\{1\}\dots$ to the endpoint 0 and $[2]$ to endpoint 1, and two schedules with a common prefix of j blocks map to points that differ by at most $(1/3)^j$. The problem is condition (3). The map sends the schedules $\{1\}[2]$ and $\{1, 2\}[1]$ to the point 1/3, yet they do not have the same

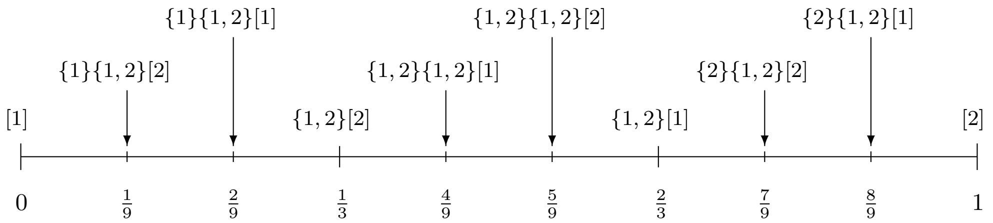


FIG. 1. *The map from* $\hat{\Sigma}(\{1,2\})$ *to the* $[0,1]$ *interval.*

compression, and $\{1\}[2]$ and $\{1,2\}[2]$ which have the same compression are mapped to $1/3$ and $2/3$, respectively. To fix this problem we modify $t = t(\sigma)$. Given $t = t(\sigma)$, define the infinite sequence a by $a_i = t_i$ if $t_1, t_2, \ldots, t_{i-1}$ has an even number of 1's and $a_i = 2 - t_i$ otherwise. Interpret a as a real number between 0 and 1 written in base 3. The map f is now defined to take $f(\sigma) = a$. It still satisfies (1), (2), and (4), but now it can also be shown to satisfy (3).

Figure 1 depicts the restriction of the mapping to compressed schedules. Under this map, degenerate schedules get mapped to rational numbers whose denominators are powers of 3. Observe that for $n = 2$, a compressed schedule σ is equal to the compression of some other schedule if and only if it is degenerate, in which case there is exactly one uncompressed schedule whose compression is σ. The mapping sends these two schedules to the same point. For example, the inverse image of $2/9$ is $\{1\}\{1,2\}[1]$ and $\{1\}\{2\}[1]$.

The bijection has a geometric description. Each prefix τ is associated with an interval R_τ which corresponds to the schedules that start with this prefix. The endpoints of the interval τ are the images of the schedules $\tau[1]$ and $\tau[2]$. The empty prefix corresponds to the entire interval, and if τ is a prefix of μ, then the interval corresponding to μ is a subinterval of that corresponding to τ. In general, the interval corresponding to τ is divided into three subintervals, corresponding to $\tau\{1\}, \tau\{1,2\}$, and $\tau\{2\}$.

It will be useful to describe this process of subdivision in *levels.* The level 0 subdivision is the subdivision into three intervals, corresponding to the prefixes $\{1\}, \{1,2\}$, and $\{2\}$. The level i subdivision is obtained from the level $i-1$ subdivision by taking each interval corresponding to a prefix of weight i (that is, the sum of the block sizes is i) and subdividing it into three subintervals as above. Thus the level 1 subdivision consists of the subdivision of the intervals $[0, 1/3]$ and $[2/3, 1]$ into three parts and the level 2 subdivision subdivides each of the intervals $[0, 1/9]$, $[2/9, 1/3]$, $[1/3, 2/3]$, $[2/3, 7/9]$, and $[8/9, 1]$ into three intervals.

4.2. The case of more than two processors. For $P = \{1,2\}$ we were able to give an explicit map from schedules to $Hull(E^P)$. For $|P| > 2$, we don't know how to do this. However, in the 2-processor case, we saw that the map can be defined by a process of successive subdivision. For each fragment τ we defined an interval R_τ which is the image of all schedules with prefix τ. For each schedule σ, the sequence of regions R_{σ^i}, where σ^i is the unique prefix of σ that appears in the level i subdivision, are nested, and their intersection is the image of σ. It should also be apparent that the lengths of the intervals into which we subdivided each interval are not critical; all we really needed was that the length of R_τ goes to 0 as we take larger and larger prefixes.

For $|P| \geq 3$ we will attempt to construct a map along similar lines. We will

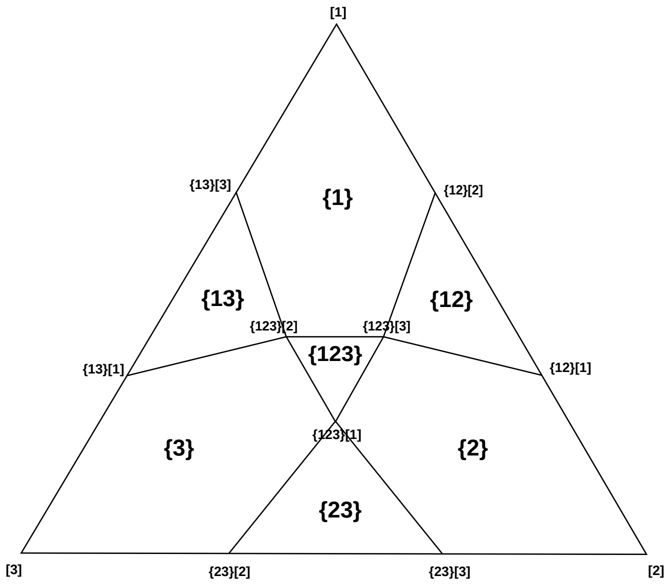


FIG. 2. *The subdivision* $D_0(\{1,2,3\})$.

construct a sequence $D_m(P)$ of decompositions of the simplex $Hull(E^P)$. The decomposition $D_0(P)$ consists of $2^{|P|}-1$ regions R_J, one for each nonempty subset of J of P; all schedules with first block J are mapped to R_J. This tiling is called the *level* 0 *decomposition* of the simplex. More generally, in the level m decomposition $D_m(P)$, $m \geq 0$, the regions correspond to fragments that are minimal subject to their weight being greater than m. The level $m+1$ decomposition is obtained from the level m decomposition by taking each region corresponding to a fragment τ of weight exactly m and tiling it by 2^n-1 regions corresponding to the fragments of the form τJ, where J is a block. The region R_τ is the image of the map applied to schedules with prefix τ. Given the level m decomposition for all m, the image of a schedule σ is determined as follows: for each m, σ is assigned to a unique $R_m(\sigma)$ region in the level m decomposition corresponding to the appropriate prefix of σ. The sequence of regions $R_m(\sigma)$ is nested and their diameters tend to 0, so σ is mapped to the unique point in their intersection. A key property will be that a point on the boundary of two or more regions will correspond to a set of schedules all having the same compression, where each schedule corresponds to one of the regions. The "vertices" of the decomposition $D_m(P)$ will correspond to schedules whose compression is an m-admissible degenerate schedule.

We now sketch how this can be done for $|P|=3$; the same approach would also seem to work for higher dimensions. The combinatorial structure underlying the map is regular, but it is sufficiently complicated that writing a complete description and a rigorous proof that it works seems to be a very tedious undertaking, which is why we do not rely explicitly on this construction in the proof of our main result. The development in the later sections is closely related to the present construction; the reader will see that this construction is the basis for the "triangulation graphs" presented in section 6.2. Conversely, the interested reader can use the precise description of the triangulation graphs to get a precise description of the map in higher dimensions.

Let $P=\{1,2,3\}$. Figure 2 shows the level 0 decomposition, $D_0(P)$ into seven regions. Each region is labeled by a nonempty subset of P. Each singleton set corresponds to a pentagonal region and each other set corresponds to a triangular region. All 0-admissible compressed degenerate schedules map to vertices in the figure. Each vertex is labeled by the unique compressed schedule that maps to it, and for each such vertex its inverse image is the set of all schedules whose compression is equal to its label. For example, the vertex labeled $\{1,2,3\}[2]$ lies on the boundary of the four regions $\{1\},\{3\},\{1,3\},\{1,2,3\}$ and for each such region there is a corresponding schedule that maps to that vertex: of $\{1\}\{2,3\}[2],\{3\}\{1,2\}[2],\{1,3\}[2]$, and $\{1,2,3\}[2]$. Consider a segment that separates two regions. Each point on the segment corresponds to two schedules, one that begins with the fragment defining the first region and one that begins with the fragment defining the other region. For example, the segment joining $\{1,2,3\}[2]$ and $\{1,3\}[1]$ separates the regions $\{3\}$ and $\{1,3\}$ and each point on the segment is the image of exactly two schedules $\{1,3\}\sigma$ and $\{3\}\{1\}\sigma$, where $\sigma \in \Sigma(\{1,2\})$.

In the higher level decompositions, we successively subdivide each region into seven subregions. When we focus on a region associated to fragment τ, it is useful to relabel each vertex of the region by the unique schedule beginning with τ that maps to that vertex. Thus, for instance, for the region $\{1,2\}$, we can view its vertices as $\{1,2\}[1]$, $\{1,2\}[2]$, and $\{1,2\}[3]$. When we subdivide this region we do it in a way analogous to the level 0 decomposition of the entire triangle, with $\{1,2\}[i]$ corresponding to the schedule $[i]$. Any triangular region is subdivided in this way.

To decompose a pentagonal region we treat it as a "distorted" triangle. In general, a pentagonal region will correspond to a fragment τ whose last block is a singleton set $\{i\}$. (The reader should follow this for the region labeled $\{1\}$.) The vertices of the region will correspond to the schedules $\tau[i]$, $\tau[j], \tau\{j,k\}[k], \tau\{j,k\}[j], \tau[k]$. When we subdivide, we view $\tau[i], \tau[j], \tau[k]$ as the vertices of the distorted triangle. The three segments joining $\tau[j]$ and $\tau[k]$ are together viewed as one "side" of the triangle. The two intermediate vertices $\tau\{j,k\}[k]$ and $\tau\{j,k\}[j]$ play the role of the vertices that are added when we subdivide that side.

Figure 3 depicts the level 1 decomposition with the vertices labeled and Figure 4 shows the decomposition with faces labeled. Note that the level 1 decomposition is produced from the level 0 decomposition by decomposing each of the three pentagons corresponding to regions whose fragment has weight 1.

The level 2 decomposition is not shown; it is obtained from the level 1 decomposition by subdividing each of the 12 regions that correspond to fragments of weight 2.

This completes our discussion of our attempt to prove Theorem 3.2 by an appropriate bijection between $Hull(E^P)$ and $\hat{\Sigma}(P)$. While we did not complete this approach, the constructions motivate much of what comes in the proof.

5. Reviewing the geometric case.

As we stated, the proof of Theorem 3.2 is obtained by following closely the proof of its geometric analog Theorem 3.3. It is useful to review the outline of that proof.

Recall that we have a cover of the simplex $Hull(E^P)$ by relatively open sets $U_1, \dots, U_n$ that satisfy the boundary property and we want to show that there is a point belonging to all of the sets. We begin with a lemma which applies to an arbitrary open cover of $Hull(E^P)$. For $\vec{x} \in Hull(E^P)$, let $B(x,\epsilon)$, the *closed ϵ-ball around x*, denote the set of points $\vec{y} \in Hull(E^P)$ within Euclidean distance ϵ of $\vec{x}$.

LEMMA 5.1. *For any finite collection $\mathcal{U}$ of open sets that covers $Hull(E^P)$ there is an $\epsilon = \epsilon(\mathcal{U}) > 0$ such that for each point $\vec{x} \in Hull(E^P)$, some member of $\mathcal{U}$ contains $B(\vec{x},\epsilon)$.*

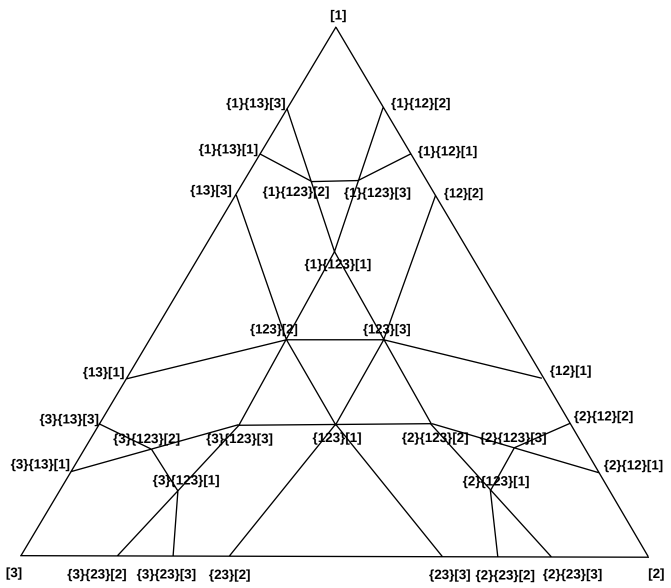


FIG. 3. *The subdivision $D_1(\{1,2,3\})$ with vertex labels.*

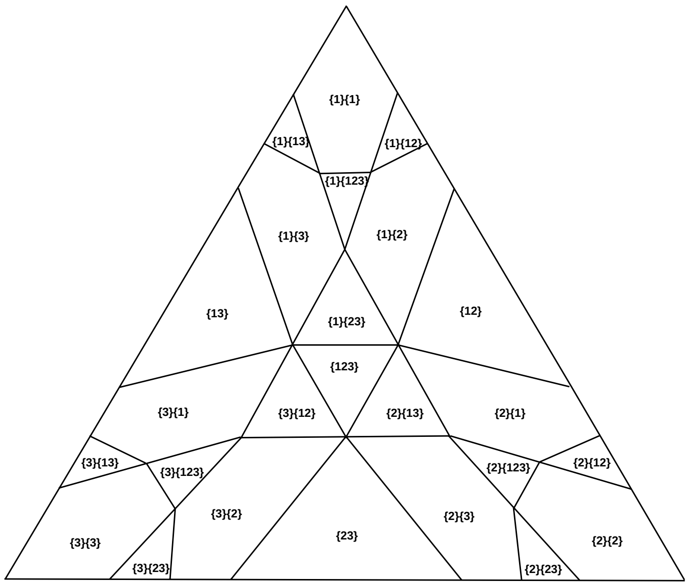


FIG. 4. *The subdivision $D_1(\{1,2,3\})$ with face labels.*

The key point is that the same ϵ works for all $\vec{x}$. Now given our covering $U_1, \ldots, U_n$ satisfying the boundary property we choose an $\epsilon > 0$ for which the conclusion of the above lemma is satisfied. We then define a function (labeling) λ of the points of $Hull(E^P)$ to $[n] = \{1, \ldots, n\}$ as follows: $\lambda(\vec{x})$ is the minimum j such that $B(\vec{x}, \epsilon) \subseteq U_j$. Note that, by the boundary property, this labeling satisfies the following *coherence condition*: for each $J \subset P$, each point of $Hull(E^J)$ is labeled by an element of J. It suffices to show that

(**) for some $\vec{y} \in Hull(E^P)$, $B(\vec{y}, \epsilon)$ contains n points with distinct labels, since then $\vec{y} \in \bigcap_{i=1}^{n} U_i$.

The proof of (**) relies on the notion of a *triangulation* of the simplex. Roughly, a triangulation T of $Hull(E^P)$ is a decomposition into a finite collection of n-vertex simplices with the property that any two simplices in the triangulation are disjoint or their intersection is a face of each (where by a face, we mean a subsimplex spanned by a subset of the vertices). The set of all simplices in T and faces of simplices in T are the *faces of T*.

For the ϵ given by Lemma 5.1, we construct a triangulation T satisfying the following.

LEMMA 5.2. *For each positive ϵ there exists a triangulation T of $Hull(E^P)$ all of whose simplices have diameter at most ϵ.*

Consider the labeling λ restricted to the vertices of T. The following lemma now completes the proof of (**) and Theorem 3.3.

LEMMA 5.3 (Sperner's lemma [1]). *Let $\lambda : Hull(E^P) \longrightarrow [n]$ be a coherent labeling and let T be a triangulation of $Hull(E^P)$. Then some n-vertex simplex of T has all of its vertices distinctly labeled.*

6. Proof of Theorem 3.2.

Recall that we have knowable sets $K_1, K_2, \ldots, K_n$ that cover $\hat{\Sigma}(P)$ and satisfy the activity property. We wish to show that they have a nonempty intersection.

The first step in adapting the proof of Theorem 3.3 is to prove an analog of Lemma 5.1. Recall that for a fragment τ the set of schedules that are quasi extensions of τ is denoted Q_τ, and $\hat{Q}_\tau = \hat{\Sigma}P \cap Q_\tau$. The sets $\hat{Q}_\tau$ play the role of ϵ-balls. For example, for each compressed schedule ϕ, and for each integer w, let $\phi^{(w)}$ denote the maximal prefix of ϕ whose weight (sum of block sizes) is at most w. If we consider the sequence of sets $\hat{Q}_{\phi^{(w)}}$ we see that $\hat{Q}_{\phi^{(1)}} \supseteq \hat{Q}_{\phi^{(2)}} \supseteq \cdots$ and that the intersection of all of them is just ϕ itself. This is analogous to a sequence of balls of decreasing radius around a particular point. The analog of Lemma 5.1 is the following.

LEMMA 6.1. *Let $\mathcal{K}$ be a collection of knowable sets that covers $\hat{\Sigma}(P)$. Then there is an integer $w = w(\mathcal{K})$ with the property that for each schedule ϕ, some member of $\mathcal{K}$ contains $\hat{Q}_{\phi^{(w)}}$.*

Proof. Suppose for contradiction that for each w there is a schedule $\phi(w)$ such that the set $\hat{Q}_{\phi(w)^{(w)}}$ is not contained in any member of $\mathcal{K}$. Let $\Phi = \{\phi(w) : w \geq 1\}$. Construct a schedule σ as follows. Let σ_1 be any set that is the first block of infinitely many members of Φ, and inductively for $i > 1$, having defined $\sigma_1 \ldots \sigma_{i-1}$, let σ_i be a set such that $\sigma_1\sigma_2 \ldots \sigma_i$ is a prefix of infinitely many members of Φ. The compression $\hat{\sigma}$ must belong to some member $K \in \mathcal{K}$. By Proposition 2.25, there is a fragment τ of $\hat{\sigma}$ such that $\hat{Q}_\tau \subseteq K$. By construction of σ, τ is a prefix of infinitely many members of Φ. In particular, it belongs to some $\phi(w)$ with $w > w(\tau)$. Then $\hat{Q}_{\phi(w)^{(w)}} \subseteq \hat{Q}_\tau \subseteq K$ contradicts the choice of $\phi(w)$. ∎

Remark. The above result implies the fact mentioned in section 2.1 that the existence of a wait-free protocol Π for k-set agreement implies the existence of a bounded wait-free protocol. Given Π, define protocol Π', where each processor behaves as in Π except that if it sees that any processor has decided before it has decided, it immediately takes the lowest decision value it sees as its decision value. Clearly Π' is correct if Π is. Letting D_i be the set of schedules where some processor decides i, the above Lemma implies that there is a w such that after at most w total steps (by all of the processors) some processor has reached a decision. Thus, after taking at most $w + 2$ steps a processor will have decided and written its decision value.

Now given our covering $K_1, \ldots, K_n$ satisfying the activity property we choose an m for which the conclusion of the above lemma is satisfied. We then define a function (labeling) λ of the points of $\hat{\Sigma}(\Sigma^P)$ to $[n] = \{1, \ldots, n\}$ as follows: $\lambda(\hat{\sigma})$ is the minimum j such that $\hat{Q}_{\hat{\sigma}^{(m)}} \subseteq U_j$. Note that, by the activity property, this labeling satisfies the following *coherence condition*: for each $J \subset B$, each $\hat{\sigma} \in \hat{\Sigma}(\Sigma_J)$ is labeled by an element of J. It suffices to show the following analog of (**).

LEMMA 6.2. *Let $K_1, \ldots, K_n$ be knowable sets that cover $\hat{\Sigma}(P)$ and let λ be a labeling of the schedules that satisfies the coherence condition. Then for any integer $m \geq 1$, there is a fragment τ of weight at least m that is a prefix of n schedules having different labels.*

The proof of this relies on abstracting the notion of a *triangulation* for the set of compressed schedules and proving an analog of Sperner's lemma.

6.1. Triangulation graphs.

By analyzing the proof of Sperner's lemma, it can be seen that the lemma can be interpreted as a statement about graphs embedded in $Hull(E^P)$ that have certain properties. This motivates the definition of the following class of graphs defined on $\hat{\Sigma}(P)$.

A *triangulation graph* on $\hat{\Sigma}(P)$ is a finite graph $G = (V, E)$ whose vertex set is a subset of $\hat{\Sigma}(P)$ and which satisfies the following properties:

(1) For each $p \in P$, the schedule $[p] = \{p\}\{p\}\{p\} \ldots$ is a vertex.
(2) If C is a clique contained in $\hat{\Sigma}(J)$ of size $|J| - 1$, then
   (a) if C is contained in $\hat{\Sigma}(I)$ for any proper subset I of J, then there is a unique clique of size $|J|$ in $\hat{\Sigma}(J)$ that contains C;
   (b) if C is not contained in $\hat{\Sigma}(I)$ for any proper subset I of J, then there are exactly two cliques of size $|J|$ in $\hat{\Sigma}(J)$ that contain C.

Here we use *clique* to mean a not necessarily maximal complete subgraph.

These conditions correspond to those satisfied by the skeleton of a triangulation in the geometric case. The first condition comes by associating the schedules $[p]$ to the generators $\vec{e}^p$ of the simplex $Hull(E^P)$, which are necessarily vertices of any triangulation. The second condition corresponds to the fact that in any triangulation of the simplex, if a $(|J| - 1)$-vertex face of T lies in $Hull(E^J)$, then (i) if it lies on the boundary of $Hull(E^J)$, then it is a face of a unique $|J|$-vertex face of T, while (ii) if it lies in the interior of $Hull(E^J)$, then it is the common boundary of a pair of $|J|$-vertex faces of T.

These conditions are sufficient to prove an analog of Sperner's lemma.

LEMMA 6.3. *Let $\lambda : \hat{\Sigma}(P) \longrightarrow [n]$ be a coherent labeling and let G be a triangulation graph on $\hat{\Sigma}([n])$. Then some n-vertex clique of G has all of its vertices labeled differently.*

Proof. For $k \in [n]$, let $g(k)$ be the number of k vertex cliques in $\hat{\Sigma}([k])$ whose vertices are labeled differently. Since λ is coherent these labels must be $\{1, 2, \ldots, k\}$. The key step is the following claim.

Claim. For each k between 2 and n, $g(k) \equiv g(k-1) \mod 2$.

It then follows that $g(n) \equiv g(1) \equiv 1 \mod 2$ since the schedule $[1]$ is the unique vertex in $\hat{\Sigma}([1])$, and we conclude that $g(n) \neq 0$.

It suffices to prove the claim. For k between 2 and n, define $p(k)$ to be the number of pairs (C', C), where C is a k-clique in $\hat{\Sigma}([k])$ and C' is a $(k-1)$-clique contained in C that is distinctly labeled $1, 2, \ldots, k-1$. Then the claim follows from

$$\begin{aligned} g(k) &\equiv p(k) \mod 2, \\ p(k) &\equiv g(k-1) \mod 2. \end{aligned}$$

The first relation is obtained by computing $p(k)$ in the following manner. Each k-clique C in $\hat{\Sigma}([k])$ which is not distinctly labeled contains either 0 or 2 cliques of size $k-1$ that are labeled $1, 2, \ldots, k-1$ and so contribute $0 \mod (2)$ to $p(k)$. On the other hand, each distinctly labeled k-clique C has a unique subclique of size $k-1$ that is labeled $1, 2, \ldots, k-1$ and so contributes 1 to $p(k)$.

The second relation follows by considering, for each $(k-1)$-clique C' with labels $1, 2, \ldots, k-1$, the number of k-cliques of $\hat{\Sigma}([k])$ to which it belongs. By the definition of triangulation graph, if C' does not lie in $\hat{\Sigma}(I)$ for some proper subset I of $[k]$, then it belongs to exactly two cliques C of size k in $\hat{\Sigma}([k])$ and so contributes 0 mod (2) to $p(k)$. On the other hand, if $C' \subset \hat{\Sigma}(I)$ for some proper subset I of $[k]$, then the definition of triangulation graph implies that C' is in a unique k-vertex clique $C \subset \hat{\Sigma}([k])$ and so contributes exactly 1 to $p(k)$. Thus $p(k) \mod 2$ counts the parity of the number of $(k-1)$-cliques C' labeled by $1, 2, \ldots, k-1$ that are in $\hat{\Sigma}(I)$ for some I properly contained in $[k]$. But the fact that C' contains all labels $1, 2, \ldots, k-1$ implies, by the coherence of λ, that I contains $[k-1]$ and so C' must be in $\hat{\Sigma}([k-1])$. Therefore, $p(k) \equiv g(k-1)$ mod 2. □

Next we will prove the following lemma.

LEMMA 6.4. *For any integer w, there exists a triangulation graph G on $\hat{\Sigma}(P)$ with the property that for any clique of G there is a fragment τ of weight at least w such that every vertex of the clique is contained in $\hat{Q}_\tau$.*

Together with Lemma 6.3, this lemma completes the proof of Lemma 6.2 and hence of Theorems 3.2 and 1.1.

Finally, it remains to show the existence of the desired triangulation graphs. This turns out to be the most technically arduous part of the proof.

6.2. Constructing triangulation graphs. We will define a sequence of triangulation graphs $\{G_m(P) | m \geq 0\}$ on $\hat{\Sigma}(P)$ with the property that any clique of $G_m(P)$ is a subset of $\hat{Q}_\tau$ for some fragment τ of weight at least $w = m - n$ (where $n = |P|$).

The precise description of $G_m(P)$ is technical, but the "picture" of the construction is nice. The graph $G_m(P)$ is closely related to the decomposition $D_m(P)$, described in section 4, but since we have only given a hint as to the general construction of $D_m(P)$, we cannot use the $D_m(P)$ explicitly in our formal description of the $G_m(P)$. Nevertheless to help understand the technical description, the reader should compare the graphs $G_0(P)$ and $G_1(P)$ depicted in Figures 5 and 6 with the pictures of $D_0(P)$ and $D_1(P)$ in section 4. The following properties are evident from the examples and will also hold in general:

(1) The vertex set of $G_m(P)$ corresponds to the vertex set of $D_m(P)$, i.e., it is the set $V_m(P)$ of m-admissible compressed degenerate schedules in $\Sigma(P)$.
(2) The graph $G_m(P)$ is obtained from the decomposition $D_m(P)$ by adding edges in order to triangulate the pentagonal regions in $D_m(P)$. Recall that each pentagonal face is labeled by a fragment that ends with a singleton block, i.e., one of the form $\tau = \mu\{i\}$. The pentagon is triangulated by connecting the vertex corresponding to (the compression of) $\tau[i]$ to the vertices corresponding to (the compression of) $\tau\{j,k\}[j]$ and $\tau\{j,k\}[k]$.

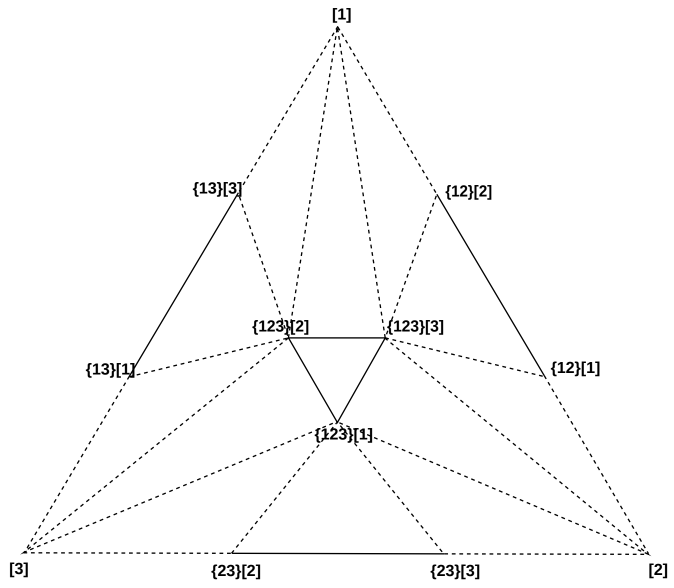


Fig. 5. *The graph* $G_0(\{1,2,3\})$.

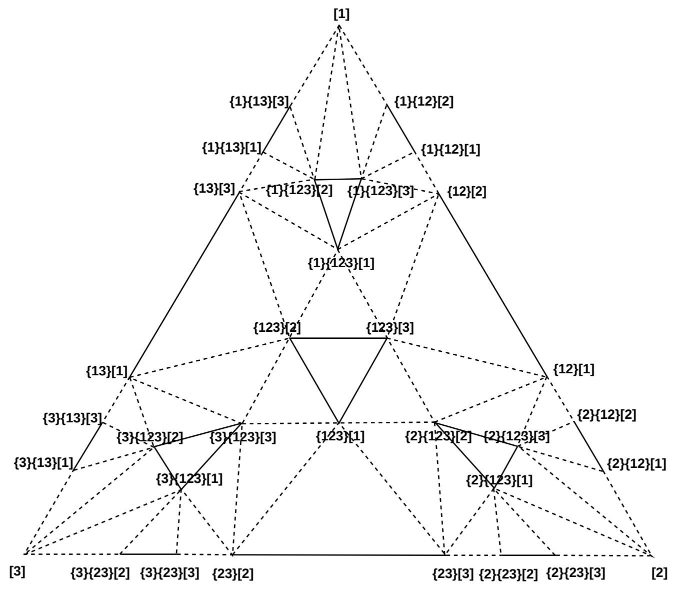


Fig. 6. *The graph* $G_1(\{1,2,3\})$.

(3) The edges in $G_m(P)$ are depicted as either solid or dashed lines. The reason for this distinction will be clear only after we give a precise definition of the graphs.

Let us now give the formal definition of the graphs $G_m(P)$. A schedule σ is *degenerate* if it has a unique nonfaulty processor. Let $V(J)$ denote the set of compressed degenerate schedules σ with $\mathit{Active}(\sigma) \subseteq J$. We say that $\sigma \in V(J)$ is p-degenerate if p is the unique nonfaulty processor, and we denote by $V^p(J)$ the set of p-degenerate schedules. Note that a p-degenerate schedule can be written in the form $\tau[p]$, where τ is a fragment and $[p]$ denotes the schedule all of whose blocks are singleton p blocks.

Recall that the weight of a fragment is the sum of its block sizes. For an arbitrary schedule σ and nonnegative integer m, let $\sigma^{(m)}$ be the maximal prefix of σ having weight less than or equal to m and let $B_m(\sigma)$ be the block of σ following $\sigma^{(m)}$. We say that σ is *m-admissible* if (i) it belongs to $V(P)$ and (ii) all blocks after $B_m(\sigma)$ are singleton p blocks. (This definition is equivalent to the definition of m-admissible given in section 4.) Thus an m-admissible schedule is of the form $\sigma^{(m)}B_m(\sigma)[p]$. Note that if σ is m-admissible, then it is j-admissible for all $j \geq m$. Also, observe that since σ is compressed, $p \in B_m(\sigma)$. We denote by $V^p_m(J)$ the set of m-admissible p-degenerate schedules in $V(J)$ and $V_m(J)$ is the union over p of $V^p_m(J)$. $V_m(P)$ is the vertex set of $G_m(P)$. Note that for $j \geq m$, this implies that the vertex set of $G_m(P)$ is a subset of the vertex set of $G_j(P)$.

We now define the edge set of $G_m(P)$. The edge set is the union of two relations: m-similarity, which is an equivalence relation, and m-shadowing, which is acyclic.

Two m-admissible schedules σ and ϕ in $V_m(P)$ are *m-similar* if $\sigma^{(m)} = \phi^{(m)}$ and $B_m(\phi) = B_m(\sigma)$. This is clearly an equivalence relation.

If $\sigma \in V^p_m(P)$ and $\phi \in V^q_m(P)$, we say that *σ m-shadows ϕ* if there is a set T satisfying $q \in T \subseteq B_m(\sigma) - \{p\}$ such that the compression of the fragment $\sigma^{(m)}T$ is equal to $\phi^{(m)}B_m(\phi)$. In particular this implies that ϕ is equal to the compression of $\sigma^{(m)}T[q]$. The m-shadow relation is acyclic since σ m-shadows ϕ implies that $w(\sigma^{(m)}B_m(\sigma)) > w(\phi^{(m)}B_m(\phi))$.

EXAMPLE 6.1. *Consider the compressed schedules:*

$$\begin{aligned}
\sigma &= \{1,2\}\{2,3\}\{1,2\}[2],\\
\rho &= \{1,2\}\{1,2,3\}[1],\\
\phi &= \{1,2\}\{1,2,3\}[2],\\
\nu &= \{1,2\}\{1,3\}[3],\\
\mu &= \{1,2\}[1].
\end{aligned}$$

Each of the schedules is in $V_m(P)$ for $m \geq 4$, all but σ also belong to $V_2(P)$ and $V_3(P)$, and μ belongs to $V_0(P)$ and $V_1(P)$. σ 4-shadows ρ, and is otherwise unrelated to all other schedules. ρ is m-similar to ϕ for $2 \leq m \leq 4$ and does not m-shadow any of the other schedules for any m. ϕ 2-shadows both μ and ν and 3-shadows ν. ν 2-shadows μ.

We can now define the edge set $E_m(P)$ of the graph $G_m(P)$. The pair $(\sigma, \phi) \in E_m(P)$ if either (i) σ and ϕ are m-similar, (ii) σ m-shadows ϕ, or (iii) ϕ m-shadows σ. In Figures 5 and 6, the solid edges correspond to those that come from m-similarity and the dashed edges arise from m-shadowing. It should be emphasized that while $V_i(P) \subset V_j(P)$ for $i < j$, the edge sets are not nested.

To complete the proof of Lemma 6.4 and hence the proof of the impossibility of wait-free m-set agreement it is now enough to prove two facts, as follows.

LEMMA 6.5.

(1) *The graph G_m is a triangulation graph for $\hat{\Sigma}(P)$.*

(2) *For any clique C in $G_m(P)$, C is contained in a set of the form $\hat{Q}_\tau$ for some fragment τ with $m - n < w(\tau)$.*

$G_m(P)$ trivially satisfies the first condition of triangulation graphs since $[p]$ is m-admissible for all $m \geq 0$.

Table 2
Summary of terminology for cliques and flags.

$\sigma^{(m)}$	the largest prefix of σ with weight $\leq m$
$B_m(\sigma)$	the first block of σ following $\sigma^{(m)}$
σ is m-admissible	σ is compressed and of the form $\sigma^{(m)} B_m(\sigma)[p]$
flag $\mathcal{F}$	a finite family of nested sets including $\emptyset$
$\mathcal{F}$ is a J-flag	the largest set in $\mathcal{F}$ is J
$\mathcal{F}^+(p)$	the unique smallest set in $\mathcal{F}$ containing element p
I-clique	consists of a p-degenerate vertex for each $p \in I$
(I, J)-clique	I-clique contained in $V_m(J)$
$(\tau, \mathcal{F})$ is (m, J)-admissible	τ a fragment, $\mathcal{F}$ a J-flag, $Active(\tau) \subseteq J$, $w(\tau) \leq m$, and $w(\tau F) > m$ for $F \in \mathcal{F} - \{\emptyset\}$
$C_I(\tau, \mathcal{F})$	$\{\sigma^p \vert p \in I\}$ where σ^p is the compression of $\tau \mathcal{F}^+(p)[p]$
$(\tau, \mathcal{F})$ is a (m, J)-flag representation of I-clique C	$C = C_I(\tau, \mathcal{F})$ and $(\tau, \mathcal{F})$ is (m, J)-admissible

The hard part is proving that $G_m(P)$ satisfies the second condition of triangulation graphs. The key is to obtain a complete characterization of the cliques of $G_m(J)$. Along the way we will also prove the second part of Lemma 6.5 (which will follow from Lemma 6.7). We advise the reader to refer frequently to Figure 6 to help in understanding what follows.

Alas, we need some more definitions. (Table 2 provides a summary of some of the key definitions.) Let $J \subseteq P$. A *flag* is a finite family $\mathcal{F}$ of sets that are totally ordered by inclusion and includes the emptyset. The unique maximal set of $\mathcal{F}$ is denoted $\mathcal{F}^M$. If $J = \mathcal{F}^M$ we say that $\mathcal{F}$ is a J-flag. For $p \in J$, $\mathcal{F}^+(p)$ denotes the unique smallest set containing p, and $\mathcal{F}^-(p)$ denotes the unique largest set not containing p.

By the definition of the edge set, any clique C consists of schedules that are p-degenerate for distinct p. If $I \subseteq P$ and C is a clique that consists of one p-degenerate schedule for each $p \in I$, then C is a called an *I-clique*. We typically denote such a clique by $\{\sigma(p) \vert p \in I\}$, where $\sigma(p)$ denotes a p-degenerate schedule.[1] As noted when defining m-admissibility, each $\sigma(p)$ is equal to $\sigma(p)^{(m)} B_m(\sigma(p))[p]$. If C is an I-clique all of whose vertices belong to $V_m(J)$, we say that C is an (I, J)-clique.

Let $C = \{\sigma(p) \vert p \in I\}$ be an (I, J)-clique. A processor $q \in I$ is said to be *dominant* in C if it maximizes $w(\sigma(p)^{(m)} B_m(\sigma(p)))$ among all $p \in I$. The following lemma provides a simple combinatorial representation for (I, J)-cliques.

Lemma 6.6. *Let $C = \{\sigma(p) \vert p \in I\}$ be an I-clique in $G_m(J)$. Let q be a dominant processor and $\tau = \sigma(q)^{(m)}$. Then*

(1) *for each $p \in I$, there is a subset F_p of J containing p such that $\sigma(p)$ is equal to the compression of $\tau F_p[p]$;*

[1] A remark on notation: The (p) in $\sigma(p)$ is simply an index, and so $\sigma(p)$ in this case stands for a p-degenerate schedule. This should not be confused with the notation $\sigma[p]$, which denotes a schedule consisting of a *fragment* σ followed by an infinite sequence of $\{p\}$ blocks.

(2) *the family of sets* $\mathcal{F} = \{F_p | p \in I\} \cup \{\emptyset, J\}$ *is a* J*-flag;*

(3) *for each* $p \in I$, $F_p = \mathcal{F}^+(p)$, *and* $w(\tau F_p) > m$.

Proof. Since q is dominant in C, then for each $p \in I - \{q\}$, $\sigma(q)$ either is m-similar to or m-shadows $\sigma(p)$, which implies that for $p \neq q$, $\sigma(p)$ is the compression of a schedule of the form $\tau F_p[p]$, where $p \in F_p \subset J$. This proves the first part. For the second part, let $p, r \in I$. If $\sigma(p)$ is m-similar to $\sigma(r)$, then the condition that the compression of τF_p is equal to the compression of τF_r implies $F_p = F_r$. Otherwise $\sigma(p)$ m-shadows $\sigma(r)$, which means that $\sigma(r)^{(m)} B_m(\sigma(r))$ can be written as the compression of $\sigma(p)^{(m)}B$ for some $B \subseteq B_m(\sigma(p)) - \{p\}$. Since τF_r and $\sigma(p)^{(m)}B$ have the same compression and τF_p and $\sigma(p)^{(m)} B_m(\sigma(p))$ have the same compression (by comparing the total number of steps taken by each processor in each of these fragments), we conclude that $F_r \subseteq F_p - \{p\}$. From this it follows that $\mathcal{F} = \{F_p | p \in I\} \cup \{\emptyset, J\}$ is a J-flag and for each $p \in I$, $F_p = \mathcal{F}^+(p)$, completing the second part. Finally, the m-admissibility of the vertices in C implies that $w(\tau F) \geq m$ for all nonempty $F \in \mathcal{F}^+(p)$. □

This motivates the following definitions. An (m, J)*-admissible pair* is a pair $(\tau, \mathcal{F})$, where $\tau \in \Phi(J)$ (recall that $\Phi(J)$ is the set of fragments whose blocks are subsets of J) and $\mathcal{F}$ is a J-flag such that $w(\tau) \leq m$ and $w(\tau F) > m$ for all nonempty $F \in \mathcal{F}$. For such a pair, we define $C_I(\tau, \mathcal{F})$ to be the set $\{\sigma(p) | p \in I\}$ of schedules where $\sigma(p)$ is the compression of the schedule $\tau \mathcal{F}^+(p)[p]$. Note that since $p \in \mathcal{F}^+(p)$ this is equal to the schedule obtained by compressing the fragment $\tau \mathcal{F}^+(p)$ and appending $[p]$.

From Lemma 6.6 we have that every (I, J)-clique is of the form $C_I(\tau, \mathcal{F})$ for some (m, J)-admissible pair. In fact, the converse of this statement also holds and we state them together in the following lemma.

LEMMA 6.7.

(1) *For* $I \subseteq J \subseteq P$, *each* (I, J) *clique of* $G_m(P)$ *has the form* $C_I(\tau, \mathcal{F})$ *for some* (m, J) *admissible pair* $(\tau, \mathcal{F})$.

(2) *If* $(\tau, \mathcal{F})$ *is an* (m, J)*-admissible pair and* $I \subseteq J$, *then* $C_I(\tau, \mathcal{F})$ *is an* (I, J)*-clique.*

Proof. The first part follows from Lemma 6.6. To prove the second part, suppose that $(\tau, \mathcal{F})$ is m-admissible. We first must show that each schedule in $C_I(\tau, \mathcal{F}) = \{\sigma(p) | p \in I\}$ is a vertex of $V_m(J)$. Proposition 6.8 follows easily from the definition of compression.

PROPOSITION 6.8. *If* $(\tau, \mathcal{F})$ *is* (m, J) *admissible,* $F = \mathcal{F}^+(p)$, *and* $p \in J$, *then the compression of* $\tau F[p]$ *can be written in the form* $\mu B[p]$, *where* $w(\mu) \leq w(\tau)$ *and* $w(\mu B) = w(\tau F)$. *Thus* $\sigma(p)$ *is* m*-admissible and so belongs to* $V_m(J)$.

Next we must show that for $p, q \in I$, $\sigma(p)$ and $\sigma(q)$ are adjacent in $G_m(P)$, i.e., either they are m-similar or one m-shadows the other. Let $A = \mathcal{F}^+(p)$ and $B = \mathcal{F}^+(q)$. Thus $\sigma(p)$ is the compression of $\tau A[p]$ and $\sigma(q)$ is the compression of $\tau B[q]$, and also $w(\tau A)$ and $w(\tau B)$ are both greater than m. If $A = B$, then $\tau A[p]$ and $\tau A[q]$ have exactly the same hidden blocks, and so $\sigma(p)$ and $\sigma(q)$ are clearly m-similar. If $A \neq B$, then without loss of generality $A \subset B$. Furthermore, $p \in A$ and $q \in B - A$, since $A = \mathcal{F}^+(p)$ and $B = \mathcal{F}^+(q)$. Let $\sigma(q) = \mu B'[q]$ be the compression of $\tau B[q]$. Then $B' = B \cup C$, where C is the union of some number of blocks (possibly 0) at the end of τ. Also, $w(\mu B') = w(\tau B)$. Every hidden block of $\tau B[q]$ is also hidden in $\tau A[p]$, and so $\tau A[q]$ can be partially compressed to $\mu A'[p]$, where $A' = A \cup C$. This implies that $p \in A' \subseteq B' - \{q\}$, and so $\sigma(q)$ m-shadows $\sigma(p)$. □

This lemma provides a nice combinatorial characterization of cliques. If C is an I-clique and $C = C_I(\tau, \mathcal{F})$, where $(\tau, \mathcal{F})$ is an (m, J)-admissible pair, then $(\tau, \mathcal{F})$

is called an (m, J)-*flag representation* of C. This representation is in general not unique. Lemma 6.6 gave such a representation for each I-clique C in $V_m(J)$. This construction has the properties that τ is equal to the prefix $\sigma(q)^{(m)}$ for some (in fact, any) dominant processor q and that $\mathcal{F}$ consists exactly of those sets F_p for $p \in I$, where F_p is the unique set such that $\sigma(p)$ is the compression of $\tau F_p[p]$. In particular, $\mathcal{F}$ is the unique minimal J-flag for which $(\tau, \mathcal{F})$ is a (k, J)-flag representation of C. We call this representation the (m, J)-*canonical* representation of C. It is clear that the (m, J)-canonical representation of the I-clique C is unique.

EXAMPLE 6.2. *Let* $m = 2$ *and* $P = \{1, 2, 3, 4, 5\}$. *The set consisting of* $\{1, 2, 3\}[3]$, $\{1, 2\}\{1, 3\}[1]$, *and* $\{1, 2\}\{1, 5, 3, 4\}[4]$ *has four* $(2, P)$-*flag representations:* $(\tau, \mathcal{F})$, $(\mu, \mathcal{F})$, $(\tau, \mathcal{G})$, *and* $(\mu, \mathcal{G})$, *where* $\mathcal{F} = \{\emptyset, \{3\}, \{1, 3\}, \{1, 3, 4, 5\}, \{1, 2, 3, 4, 5\}\}$, $\mathcal{G} = \{\emptyset, \{3\}, \{1, 3\}, \{1, 3, 5\}, \{1, 3, 4, 5\}, \{1, 2, 3, 4, 5\}\}$, $\tau = \{1, 2\}$, *and* $\mu = \{2\}\{1\}$. *The canonical representation is* $(\tau, \mathcal{F})$.

We can now prove the second part of Lemma 6.5. If C is any I-clique, let $(\tau, \mathcal{F})$ be an (m, P)-flag representation of C. The (m, P)-admissibility of $(\tau, \mathcal{F})$ implies that $w(\tau) > m - n$. Every schedule in C is the compression of a schedule of the form $\tau F[p]$ for some $F \in \mathcal{F}$ and $p \in I$ and is thus a quasi extension of τ. Thus $\sigma \in \hat{Q}_\tau$ as required.

It remains only to check that $G_m(P)$ satisfies the third condition of the definition of triangulation graphs. We will prove the following.

LEMMA 6.9. *Let* $I \subseteq J \subseteq P$ *and let* C *be an* I-*clique in* $G_m(J)$. *Then the number of* (J, J)-*cliques that contain* C *is equal to the number of distinct* J-*flag representations of* C.

In light of this lemma, the two parts of the second property of triangulation graphs follow, respectively, from the two parts of the following lemma.

LEMMA 6.10. *Let* $J \subseteq P$, $p \in J$, *and* $I = J - \{p\}$. *Let* C *be an* I-*clique that is contained in* $V_m(J)$.

(1) *If* C *is contained in* $V_m(I)$, *then* C *has a unique* J-*flag representation.*

(2) *If* C *is not contained in* $V_m(I)$, *then* C *has exactly two* J-*flag representations.*

Thus, all that remains is to prove Lemmas 6.9 and 6.10, which we now do. We first make some preliminary observations about flags and flag representations.

PROPOSITION 6.11. *Let* $\mathcal{F}$ *and* $\mathcal{H}$ *be* J-*flags. If* $\mathcal{H}^+(r) = \mathcal{F}^+(r)$ *for every* $r \in J$, *then* $\mathcal{H} = \mathcal{F}$.

Proof. Suppose that $\mathcal{H} \neq \mathcal{F}$ are J-flags and let A be a set that is in one but not the other, say, it is in $\mathcal{H}$ but not in $\mathcal{F}$. Then there is an element r such that $\mathcal{H}^+(r) = A$, and so $\mathcal{H}^+(r) \neq \mathcal{F}^+(r)$. $\square$

LEMMA 6.12. *Let* $C = \{\sigma(p) | p \in I\}$ *be an* (I, J)-*clique and let* q *be a dominant processor. Let* $(\tau, \mathcal{F})$ *be the canonical* (m, J)-*flag representation of* C. *Let* $(\mu, \mathcal{H})$ *be an arbitrary* (m, J)-*flag representation of* C. *Then*

(1) $\sigma(q) = \tau\mathcal{F}^+(q)[q]$.

(2) μ *can be written in the form* $\nu\lambda$, *where* ν *is a fragment such that the compression of* $\nu\mathcal{F}^+(q)$ *is* $\tau\mathcal{F}^+(q)$ *and* λ *is a possibly empty fragment consisting of pairwise disjoint blocks.*

(3) *Let* B *denote the union of the blocks of* λ. *Then* $B \subseteq J - I$ *and* $\mathcal{H}^+(p) = \mathcal{F}^+(p) - B$ *for each* $p \in I$.

(4) $I \subseteq \mathcal{H}^+(q) \subseteq \mathcal{F}^+(q) \subseteq J$.

Proof. The first part follows immediately from the definition of the canonical (m, J)-flag representation. For the second part, note that $\mu\mathcal{H}^+(q)[q]$ must compress to $\sigma(q) = \tau\mathcal{F}^+(q)[q]$, which means that $\mu\mathcal{H}^+(q)$ must compress to $\tau\mathcal{F}^+(q)$. Let ν be the portion of μ that compresses to τ and let λ be the portion of μ that is merged with $\mathcal{H}^+(q)$ to form $\mathcal{F}^+(q)$. Then $\mu = \nu\lambda$, and λ must consist of disjoint blocks. For the third part, if $p \in I$, then $\tau\mathcal{F}^+(p)$ must have the same compression as $\nu\lambda\mathcal{H}^+(p)$, which means that $\lambda\mathcal{H}^+(p)$ must compress to $\mathcal{F}^+(p)$, so $\mathcal{H}^+(p) = \mathcal{F}^+(p) - B$. Since $p \in \mathcal{H}^+(p)$, we must have $p \notin B$, so $B \subset J - I$. For the fourth part, $\mathcal{H}^+(q)$ contains $\mathcal{H}^+(p)$ for all $p \in I$, so $I \subseteq \mathcal{H}^+(q)$. ☐

LEMMA 6.13. *Each (J, J)-clique has a unique (m, J)-flag representation.*

Proof. Let $C = \{\sigma(p) : p \in J\}$ be a J-clique contained in $V_m(J)$ and let $(\tau, \mathcal{F})$ be its J-canonical representation. Suppose that $(\mu, \mathcal{H})$ is any other J-flag representation. Since $\mathcal{H}$ is a J-flag, there must be $q \in J$ such that $\mathcal{H}^+(q) = J$. Then $\mu J[q]$ is compressed and must be equal to $\sigma(q)$. Furthermore q is dominant in C, so by the definition of the canonical representation $\tau = \mu$.

By Proposition 6.11, if $\mathcal{G} \neq \mathcal{F}$, then there exists r such that $\mathcal{F}^+(r) \neq \mathcal{G}^+(r)$. But then the compression of $\tau\mathcal{F}^+(r)[r]$ cannot be equal to the compression of $\tau\mathcal{G}^+(r)[r]$. Therefore $\mathcal{F} = \mathcal{G}$ and the (m, J)-flag representation is unique. ☐

Now we are ready to prove Lemmas 6.9 and 6.10, to finish the proof of the main theorem.

Proof of Lemma 6.9. Let C be an (I, J)-clique, and let $C^1, C^2, \ldots, C^r$ be the distinct (J, J)-cliques that contain C. By Lemma 6.13, each of the C^i has a unique (m, J)-flag representation, $(\tau_i, \mathcal{F}_i)$, and by the definition of the representation, $(\tau_i, \mathcal{F}_i)$ is also an (m, J)-flag representation of C, i.e., $C = C_I(\tau_i, \mathcal{F}_i)$. Also, these are the only (m, J)-flag representations of C, since any such representation for C is also an (m, J)-flag representation of some J-clique containing C. ☐

Proof of Lemma 6.10. Let $C = \{\sigma(r) | r \in I\}$ be an $I = J - \{p\}$-clique and let $(\mu, \mathcal{H})$ denote an arbitrary (m, J)-flag representation of C. Let q denote a dominant processor of C. From Lemma 6.12, $I \subseteq \mathcal{H}^+(q) \subseteq J$.

To prove the first part of the lemma, suppose that C is contained in $V_m(I)$. Then $\mathcal{H}^+(q) \neq J$ so $\mathcal{H}^+(q) = I$. Let $\mathcal{G} = \mathcal{H} - \{J\}$. Then $(\mu, \mathcal{G})$ is a (m, I)-flag representation of C. But, by Lemma 6.13, there is only one such representation, so this implies that $(\mu, \mathcal{H})$ must be the unique (m, J)-flag representation.

We proceed to the second part of the lemma. Let $(\tau, \mathcal{F})$ be the canonical (m, J)-flag representation of C and let q be a dominant processor of C. We want to show that there is exactly one other representation. Now, by Lemma 6.12, either $\mathcal{F}^+(q) = J$ or $\mathcal{F}^+(q) = I$. We proceed by analyzing these two cases separately.

Case I. $\mathcal{F}^+(q) = I$. First we construct another (m, J)-flag representation. Let S be the last block of τ that contains p. There is such a block since, by hypothesis, $C \not\subset V_m(J - \{p\})$. If $S = \{p\}$, then $\tau\mathcal{F}^+(q)$ would not be compressed, contradicting the definition of the canonical representation. So $S \neq \{p\}$. Let ψ be the sequence obtained by replacing the block S by $\{p\}$ followed by $S - \{p\}$. Then $(\psi, \mathcal{F})$ is also an (m, J)-representation of C.

Now we show that this is the only other (m, J)-flag representation of C. Let $(\mu, \mathcal{H})$ be an arbitrary (m, J)-flag representation of C. Then, by Lemma 6.12, $I \subseteq \mathcal{H}^+(q) \subseteq \mathcal{F}^+(q)$ implies that $\mathcal{H}^+(q) = \mathcal{F}^+(q) = I$. Defining ν, λ, B as in Lemma 6.12 we must have $B = \emptyset$ and λ is the empty string. Then $\mathcal{H}^+(r) = \mathcal{F}^+(r)$ for all $r \in I$, and also $\mathcal{H}^+(p) = \mathcal{F}^+(p) = J$, so Proposition 6.11 implies $\mathcal{H} = \mathcal{F}$. Finally, $\mu I[q]$ must compress to $\tau I[q]$. Then either $\mu I[q]$ is already compressed (and $\mu = \tau$) or μ contains a hidden block. But a block in $\mu I[q]$ can be hidden only if it is disjoint from I, i.e., it is a singleton p block, and it must be the last appearance of p. This means that μ is equal to ψ above and so $(\mu, \mathcal{H}) = (\psi, \mathcal{F})$.

Case II. $\mathcal{F}^+(q) = J$. Then $J - \{p\}$ is not in $\mathcal{F}$. Again, we must construct another (m, J)-flag representation of C and show that this is the only other one. First note the following.

PROPOSITION 6.14. *If* $(\mu, \mathcal{H})$ *is any* (m, J)*-representation of* C*, then either* $\mu = \tau$ *or* $\mu = \tau\{p\}$*.*

To see this, note that by Lemma 6.12, either $\mathcal{H}^+(q) = J$ or $\mathcal{H}^+(q) = I$. Also $\sigma(q) = \tau J[q]$ is the compression of $\mu\mathcal{H}^+(q)[q]$. Thus if $\mathcal{H}^+(q) = J$, then $\mu = \tau$, and if $\mathcal{H}^+(q) = I$, then μ must be $\tau\{p\}$.

Now we proceed with constructing an alternative representation of C. Let $A = \mathcal{F}^-(p) \cup \{p\}$. Note that $A \notin \mathcal{F}$ since in the canonical representation, every set in $\mathcal{F}$ is of the form $\mathcal{F}^+(r)$ for some $r \neq p$. Let $\mathcal{G} = \mathcal{F} \cup \{A\}$; then $\mathcal{G}^+(r) = \mathcal{F}^+(r)$ for each $r \neq p$. Thus $(\tau, \mathcal{G})$ would seem to be another (m, J)-representation of C. This is indeed true, except in one case. The problem is that the definition of (m, J) representation requires that $(\tau, \mathcal{G}^+(s))$ be m-admissible for all $s \in J$, which means that $w(\tau) \leq m < w(\tau\mathcal{G}^+(s))$. Now, this is true for all $s \neq p$, since it was true for $(\tau, \mathcal{F})$. However, it is possible that $w(\tau\mathcal{G}^+(p)) \leq m$. This happens if and only if $w(\tau) < m$ and $\mathcal{F}^-(p) = \emptyset$ so that $A = \{p\}$. So we consider two subcases, depending on whether this happens.

Subcase II*a*: $(\tau, \mathcal{G})$ *is* (m, J)*-admissible.* As we have just discussed, this means that either $\mathcal{F}^-(p) \neq \emptyset$ or $\mathcal{F}^-(p) = \emptyset$ and $w(\tau) = m$. Then $(\tau, \mathcal{G})$ is a second (m, J)-representation of $(\tau, \mathcal{F})$; we need to prove that there are no others.

If $(\mu, \mathcal{H})$ is an (m, J)-flag representation of C, then let ν, λ, B be as in Lemma 6.12. Then $B = \{p\}$ or $B = \emptyset$. We claim $B = \emptyset$. If $B = \{p\}$, then $w(\mu) = w(\tau) + 1$ and so the (m, J)-admissibility of $(\mu, \mathcal{H})$ requires $w(\tau) < m$ and so $\mathcal{F}^-(p) \neq \emptyset$. Let $s \in \mathcal{F}^-(p)$; then $\mathcal{F}^+(s) \subseteq \mathcal{F}^-(p)$. But $\tau\{p\}\mathcal{H}^+(s)$ must compress to $\tau\mathcal{F}^+(s)$, which is impossible since $p \notin \mathcal{F}^+(s)$.

Thus $B = \emptyset$, and so $\mathcal{H}^+(q) = \mathcal{F}^+(q) = J$, which means by Proposition 6.14 that $\mu = \tau$. We now need to show that $\mathcal{H} = \mathcal{F}$ or $\mathcal{H} = \mathcal{G}$, i.e., $\mathcal{F} \subseteq \mathcal{H} \subseteq \mathcal{G}$. $\mathcal{H}$ must contain $\mathcal{F}$, since $\mathcal{H}^+(r) = \mathcal{F}^+(r)$ for all $r \in I$ and $\mathcal{F}$ was defined only to contain $\emptyset$, J, and the sets $\mathcal{F}^+(r)$ for $r \in I$. If $\mathcal{H}$ is not a subset of $\mathcal{G}$ let D be the minimal member of $\mathcal{H} - \mathcal{G}$ and let D_0 be the largest subset of D in $\mathcal{G}$. Choose $x \in D - D_0$ and note that $x \neq p$, since $\mathcal{F}^-(p)$ and A are both in $\mathcal{G}$. Then $\mathcal{H}^+(x) = D \neq \mathcal{G}^+(x)$, which contradicts that $(\tau, \mathcal{H})$ and $(\tau, \mathcal{G})$ both are (m, J) representations of C.

Subcase II*b*: $(\tau, \mathcal{G})$ *is not* (m, J)*-admissible.* This means that $\mathcal{F}^-(p) = \emptyset$ and $w(\tau) < m$. Thus $(\tau\{p\}, \mathcal{E})$ is another (m, J)-flag representation, where $\mathcal{E}$ is obtained from $\mathcal{F}$ by deleting p from each set in $\mathcal{F}$ and adding the set J.

Suppose that $(\mu, \mathcal{H})$ is any (m, J)-flag representation of C; we want to show that $(\mu, \mathcal{H}) = (\tau, \mathcal{F})$ or $(\mu, \mathcal{H}) = (\tau\{p\}, \mathcal{E})$. Let ν, λ, B be as in Lemma 6.12. Once again we have either $\mu = \tau$ and $B = \emptyset$ or $\mu = \tau\{p\}$ and $B = \{p\}$.

In the case $\mu = \tau$, we also have that $\mathcal{F}^+(r) = \mathcal{H}^+(r)$ for all $r \neq p$. We claim that $\mathcal{F}^+(p) = \mathcal{H}^+(p)$, which would imply that $\mathcal{F} = \mathcal{H}$. To see the claim, observe that $\mathcal{F}^+(p)$ is the smallest nonempty set of $\mathcal{F}$, and it belongs to $\mathcal{H}$ since $\mathcal{F} \subseteq \mathcal{H}$. Thus it suffices to show that $\mathcal{H}$ contains no smaller set. $\mathcal{H}$ cannot contain $\{p\}$ since $w(\tau p) \leq m$ would violate m-admissibility of $(\tau, \mathcal{H})$. $\mathcal{H}$ cannot contain any other subset of $\mathcal{H}$ because $\mathcal{H}^+(r) = \mathcal{F}^+(r)$ for all $r \neq p$. Thus $\mathcal{F} = \mathcal{H}$, and $(\mu, \mathcal{H}) = (\tau, \mathcal{F})$.

In the case that $\mu = \tau\{p\}$, for any $r \neq q$ we must have that $\tau\mathcal{F}^+(r)$ and $\tau\{p\}\mathcal{H}^+(r)$ compress to the same vertex of C. Then for every $r \neq p$, $\mathcal{H}^+(r) = \mathcal{F}^+(r) - \{p\} = \mathcal{E}^+(r)$. We also have $\mathcal{H}^+(p) = \mathcal{E}^+(p) = J$ since $J - \{p\}$ is a member of both of them. From Proposition 6.11, $\mathcal{H} = \mathcal{E}$, and thus $(\mu, \mathcal{H}) = (\tau\{p\}, \mathcal{E})$. $\square$

This completes the proof of Lemma 6.10 which, as explained, completes the proof that G_m is a triangulation graph and thus completes the proof of the main theorem.

Appendix. The topology of knowable sets. In this appendix, we look more closely at the structure of the collection of knowable sets, $\mathcal{K} = \{K | K \subseteq \hat{\Sigma} \text{ is a knowable set}\}$. In particular, we prove that $\mathcal{K}$ defines a compact Hausdorff topological space on $\hat{\Sigma}(P)$. Along the way we give two characterizations of this space: (i) we give a nice basis for $\mathcal{K}$, and (ii) we show that $\mathcal{K}$ is the quotient of the Cantor topology on $\Sigma(P)$ with respect to the compression map.

Notions from point set topology are briefly reviewed as needed. For more details, see [24].

A.1. $\mathcal{K}$ is a Hausdorff topology. Recall that formally, a topological space on a set X is a collection $\mathcal{U}$ of subsets of X that includes $\emptyset$ and X and is closed under arbitrary union and finite intersection. The members of $\mathcal{U}$ are the *open sets* of the topology, and complements of members of $\mathcal{U}$ are the *closed sets* of the topology.

THEOREM A.1. *$\mathcal{K}$ is a topology on the set $\hat{\Sigma}$.*

Proof. We observed in section 2.8 that $\emptyset$ and $\hat{\Sigma}$ are both in $\mathcal{K}$. We need to show that the union of an arbitrary collection of knowable sets is knowable and the intersection of a finite collection of knowable sets is knowable.

Let $\{K^i\}_{i\in\Lambda}$ be an arbitrary collection of knowable sets. We will show that $K^{\cup} = \cup_{i\in\Lambda}K^i$ is a knowable set. Let (Π^i, d^i) be an acceptor for K_i and V^i be the set of values that can be written to the registers by this protocol. We exhibit an acceptor $(\Pi^{\cup}, d)$ for $K^{\cup}$. Informally the protocol simulates all the protocols in the above collection in parallel and a processor writes the accept value d when it sees that at least one of accept values d^i has been written.

More formally protocol $\Pi^{\cup}$ is defined as follows. The set of processor states S consists of the (possibly infinite) product set $\prod_{i\in\Lambda}(S^i)$ together with a special state $s^\star$. Thus a state value s is either $s^\star$ or a tuple $(s^i | i \in \Lambda)$, where $s^i \in S^i$. (A state value of $s^\star$ will mean that a processor is ready to write the accept value d.) The initial state e_p is the tuple $(e^i_p | i \in \Lambda)$. The set of write values V is the product set $\prod_{i\in\Lambda} V^i$ together with the accept value d. If no processor has ever written d, then the contents of shared memory $\vec{l}$ can be viewed as a tuple $(\vec{l}^{\,i} : i \in \Lambda)$, where $\vec{l}^{\,i}$ corresponds to the run of Π^i. The write map w is defined as $w(s) = d$ if $s = s^\star$ and otherwise $w(s)$ is the tuple $(w^i(s^i) : i \in \Lambda)$. The state update map u is defined as $u(s, \vec{l}) = s^\star$ if d appears in $\vec{l}$ or there is at least one $i \in \Lambda$ such that d^i appears in $\vec{l}^{\,i}$; otherwise $u(s, \vec{l}) = (u^i(s^i, \vec{l}^{\,i}) : i \in \Lambda)$. It is easy to see that the accept value d is written by $\Pi^{\cup}$ on schedule σ if and only if there is an $i \in \Lambda$ such that d^i is written by Π^i on σ. Thus the set $K^{\cup}$ is knowable.

Next we want to show that the intersection of a finite collection of knowable sets is knowable. As above, let $\{K^i\}_{i\in\Lambda}$ be a collection of knowable sets and (Π, d^i) be acceptors. We define a protocol $\Pi^{\cap}$ by a minor modification of $\Pi^{\cup}$. The only difference is in the state update map u. The condition for a processor to enter state $s^\star$ is either that some processor has written d or *for every* $i \in \Lambda$, d_i appears in $\vec{l}^{\,i}$.

It is easy to see that if the accept value d is written by $\Pi^{\cap}$ on schedule σ, then for all $i \in \Lambda$, d^i is written by Π^i on σ and thus $K(\Pi^{\cap}, d)$ is a subset of $K^{\cap}$. The reverse containment holds if Λ is finite (although it need not hold if Λ is infinite; see below). For $\sigma \in K^{\cap}$ let j^i be the index of the block of σ in which d^i is first written and let j be the maximum of the j^i. Then any processor taking a step subsequent to block j will see all of the decision values d^i and thus move to state $s^\star$. At least one such processor will take another step and will thus write d. □

Observe that this final argument fails when the collection $\{K^i\}$ is infinite because the index j may not be defined. As an example, let K^i be the set of compressed schedules where processor p takes at least i steps, and let i range over the positive integers.

Next, recall that a topological space $(X, \mathcal{U})$ is a *Hausdorff space* if for any two distinct points $x, y \in X$ there are disjoint open sets U_x and U_y with $x \in U_x$ and $y \in U_y$.

LEMMA A.2. *$(\Sigma, \mathcal{K})$ satisfies the Hausdorff condition.*

Proof. In this case the Hausdorff condition means that if σ and ϕ are distinct compressed schedules, then there is a pair of acceptors (Π, d) and (Φ, d') such that Π accepts σ and Φ accepts ϕ and no schedule is accepted by both. We will take Π and Φ to be a minor modification of the counting protocol: when processor p writes for the ith time, it writes (T, p), where T is its current tally vector (instead of just T).

Now we need to choose the accepting values for Φ and Π, which will be of the form (T, p). Since σ and ϕ are distinct compressed schedules, Theorem 2.12 implies that their tally records must be different. Apply Lemma 2.5. Under the first conclusion of this lemma, there is a processor p and a positive integer i such that p takes at least i steps in both schedules and tally vectors $Count_{p,i}(\phi)$ and $Count_{p,i}(\phi)$ are different. Thus take $d = (Count_{p,i}(\sigma), p)$ and $d' = (Count_{p,i}(\phi), p)$. Note that for any schedule ρ, at most one of d and d' can appear in its public tally since both can appear only in the list of processor p, and that list can contain only one vector that has an $i - 1$ in position p, while both of these vectors have an $i - 1$ in that position.

Under the second conclusion of Lemma 2.5 there is a pair of crossing vectors v and w such that v appears in the public tally corresponding to σ and w appears in the public tally corresponding to ϕ. Let q be a processor that writes v during σ and r be a processor that writes w during ϕ. Let $d = (v, q)$ and $d' = (w, r)$; the fact that v and w are crossing ensures that no schedule can be accepted by both protocols. □

A.2. A basis for $\mathcal{K}$. A basis for a topology $(X, \mathcal{U})$ is a collection $\mathcal{B}$ of open sets with the property that every open set is a union of members of $\mathcal{B}$. Equivalently, $\mathcal{B}$ is a basis if for any point x and open set U containing x, there is a $B \in \mathcal{B}$ such that $x \in B \subseteq U$.

Recall that for a fragment τ, the set Q_τ is the set of schedules that are quasi extensions of τ, and $\hat{Q}_\tau = Q_\tau \cap \hat{\Sigma}(P)$. Then Example 2.21 and Proposition 2.25 imply the following.

THEOREM A.3. *The set $\{\hat{Q}_\tau : \tau$ a fragment$\}$ is a basis for the knowable set topology.*

A.3. Representing $\mathcal{K}$ as a quotient topology. Let $(X, \mathcal{U})$ be a topological space and $f : X \longrightarrow Y$ be any surjective map. It is easily checked that the collection $\mathcal{U}/f = \{W \subseteq Y : f^{-1}(W) \in \mathcal{U}\}$ defines a topology on Y, called the *quotient* of $(X, \mathcal{U})$ by f. Here we represent the knowable set topology on $\hat{\Sigma}(P)$ as a quotient of a simple topology on $\Sigma(P)$.

For a fragment τ, let B_τ be the set of all schedules that have τ as a prefix. Let $(\Sigma(P), \mathcal{S})$ be the topology whose open sets are unions of sets B_τ. (This is called the *Cantor topology* on $\Sigma(P)$.)

Consider the map $f : \Sigma \longrightarrow \hat{\Sigma}$, where $f(\sigma) = \hat{\sigma}$. In this case, the quotient topology $\mathcal{R} = \mathcal{S}/f$ is defined on $\hat{\Sigma}$ and is given by $\mathcal{R} = \{U \subset \hat{\Sigma} | f^{-1}(U) \in \mathcal{S}\}$.

THEOREM A.4. *$\mathcal{K}$ and $\mathcal{R}$ define the same topology on $\hat{\Sigma}$.*

Proof. (1) $\mathcal{K} \subseteq \mathcal{R}$. Since $\{\hat{Q}_\tau | \tau \in \Phi(P)\}$ is a basis for $\mathcal{K}$, it suffices to show that for each fragment τ, $\hat{Q}_\tau$ is in $\mathcal{R}$, i.e., that $f^{-1}(\hat{Q}_\tau)$ is open in the Cantor topology. For this, it suffices to show that if $\sigma \in f^{-1}(\hat{Q}_\tau)$, then there is a fragment μ so that $\sigma \in B_\mu \subseteq f^{-1}(\hat{Q}_\tau)$. Since σ is in $f^{-1}(\hat{Q}_\tau)$ it is a quasi extension of τ, which by Theorem 2.16 means that its public tally is an extension of the public tally of τ. Since τ is finite, there is a prefix μ of σ such that the public tally of μ extends the public tally of τ. This implies that any schedule in B_μ is a quasi extension of τ, i.e., $B_\mu \subseteq f^{-1}(\hat{Q}_\tau)$.

(2) $\mathcal{R} \subseteq \mathcal{K}$. Let $S \subset \hat{\Sigma}$ be a set such that $f^{-1}(S)$ is an open set in the Cantor topology. We will prove that S is a knowable set. Let $\sigma \in S$ be arbitrary. It suffices to show that there is a knowable set K containing σ such that $K \subset S$.

Let $f^{-1}(\sigma) = \{\sigma^1, \ldots, \sigma^k\}$, which is a finite set by Proposition 2.10. For each $\sigma^i \in f^{-1}(\sigma)$ let B_{ρ_i} be a basis set in the Cantor topology such that $\sigma^i \in B_{\rho_i} \subset f^{-1}(S)$.

Choose a prefix ρ of σ that has greater weight (total number of steps) than each of the ρ_i and also contains all of the steps of the faulty processors of σ. We write $\sigma = \rho\chi$, where Active(χ) is equal to N, the set of nonfaulty processors of σ. Hence, ρN is compressed since $\rho\chi$ is. By Lemma 2.15, $\sigma \in \hat{Q}_{\rho N}$. We claim that $\hat{Q}_{\rho N} \subseteq S$, which will complete the proof.

Let γ be an arbitrary schedule in $\hat{Q}_{\rho N}$; we show that $\gamma \in S$. For this it suffices to find a j such that $\gamma \in B_{\rho_j}$, i.e., ρ_j is a prefix of γ. By Theorem 2.16 there is a prefix τ of γ, a schedule ϕ, and a subset U of Active(ϕ) such that $\gamma = \tau\phi$ and $\widehat{\tau U} = \rho N$. Since $\widehat{\tau U} = \rho N$ we must have $U \subseteq N$ and $\tau = \beta\zeta$, where ζ consists of some sequence of disjoint blocks whose union is $U - N$ and the last block of β has nonempty intersection with N. Thus $\widehat{\beta N} = \widehat{\tau U} = \rho N$.

We now claim that $\widehat{\beta\chi} = \sigma$. Now, since σ is compressed, so is χ. It is easy to see that a block is hidden in $\beta\chi$ if and only if it is hidden inside β within the fragment βN, and since the compression of βN is ρN, we have $\widehat{\beta\chi} = \rho\chi = \sigma$. Hence $\beta\chi \in f^{-1}(\sigma)$, and $\beta\chi \in B_{\rho_j}$ for some j. Since β and ρ have the same weight, which is at least the weight of ρ_j, we have that ρ_j is a prefix of β. Therefore $\gamma = \beta\zeta\phi \in B_{\rho_j}$. ☐

A topological space $(X, \mathcal{U})$ is *compact* if for any collection of open sets whose union is X there is a finite subcollection whose union is X. We remark that since the Cantor topology is known to be compact, and a quotient of a compact topology is compact, we have the following.

COROLLARY A.5. *The topological space $(\hat{\Sigma}, \mathcal{K})$ is compact.*

Acknowledgments. We would like to express our appreciation to Soma Chaudhuri for introducing us to this problem and for several enlightening discussions. We thank Shlomo Moran for helpful comments on an earlier draft. We thank two anonymous referees for numerous helpful suggestions. Thanks to Shiyu Zhou and Sri Divakaran for help in making several of the figures.

REFERENCES

[1] P. S. ALEKSANDROV, *Combinatorial Topology*, Graylock Press, Rochester, NY, 1956.

[2] Y. AFEK, H. ATTIYA, D. DOLEV, E. GAFNI, M. MERRITT, AND N. SHAVIT, *Atomic snapshots*, J. ACM, 40 (1993), pp. 873–890. A preliminary version appeared as *Atomic snapshots of shared memory* in Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, 1990, pp. 1–13.

[3] J. ANDERSON, *Composite registers*, Distrib. Comput., 6 (1993), pp. 141–154. A preliminary version appeared in Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, 1990, pp. 15–29.

[4] J. Aspnes and M. Herlihy, *Wait-free data structures in the asynchronous PRAM model*, in Proceedings of the Second Annual Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 340–349.

[5] H. Attiya, N. Lynch, and N. Shavit, *Are wait-free algorithms fast?*, J. ACM, 41 (1994), pp. 725–763. A preliminary version appeared in Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 55–64.

[6] O. Biran, S. Moran, and S. Zaks, *A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor*, J. Algorithms, 11 (1990), pp. 420–440. A preliminary version appeared in Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, 1988, pp. 263–275.

[7] E. Borowsky and E. Gafni, *Generalized FLP impossibility result for t-resilient asynchronous computations*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, CA, 1993, pp. 91–100.

[8] E. Borowsky and E. Gafni, *Immediate atomic snapshots and fast renaming*, in Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing, Ithaca, NY, 1993, pp. 41–51.

[9] H. Brit and S. Moran, *Wait-freedom vs. bounded wait-freedom in public data structures*, J. UCS, 2 (1996), pp. 2–19.

[10] S. Chaudhuri, *More choices allow more faults: Set consensus problems in totally asynchronous systems,* Inform. and Comput., 105 (1993), pp. 132–158. A preliminary version appeared as *Agreement is harder than consensus: Set consensus problems in totally asynchronous systems* in Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, 1990, pp. 311–324.

[11] S. Chaudhuri, M. Herlihy, N. Lynch, and M. Tuttle, *A tight lower bound for k-set agreement*, in Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, 1994, pp. 206–215.

[12] R. Cole and O. Zajicek, *The expected advantage of asynchrony*, J. Comput. System Sci., 51 (1995), pp. 286–300. A preliminary version appeared in Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, 1990, pp. 85–94.

[13] D. Dolev and N. Shavit, *Bounded concurrent time-stamps*, SIAM J. Comput., 6 (1997), pp. 418–455. A preliminary version appeared as *Bounded concurrent time-stamp systems are constructible!* in Proceedings of the 21th Annual ACM Symposium on Theory of Computing, Seattle, WA, 1989, pp. 454–465.

[14] R. Fagin, Y. J. Halpern, Y. Moses, and M. Vardi, *Reasoning about Knowledge*, MIT Press, Cambridge, UK, 1995.

[15] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distributed consensus with one faulty process*, J. ACM, 32 (1985), pp. 374–382.

[16] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Languages Systems, 13 (1991), pp. 124–149.

[17] M. Herlihy and N. Shavit, *The topological structure of asynchronous computability*, J. ACM, to appear. A preliminary version appeared as *The asynchronous computability theorem for t-resilient tasks* in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, CA, 1993, pp. 111–120.

[18] M. Herlihy and N. Shavit, *A simple constructive computability theorem for wait-free computation*, in Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 243–252.

[19] A. Israeli and Ming Li, *Bounded time-stamps*, Distrib. Comput., 6 (1993), pp. 205–209. A preliminary version appeared in Proceedings of the 28th Annual Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 371–382.

[20] L. Lamport, *On Interprocess Communication. Part* I: *Basic Formalism, Part* II: *Algorithms*, Distrib. Comput., 1 (1986), pp. 77–101.

[21] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, in Adv. Comput. Res. 4, JAI Press, Greenwich, CT, 1987, pp. 163–183.

[22] C. Martel, A. Park, and R. Subramonian, *Work-optimal asynchronous algorithms for shared memory parallel computers*, SIAM J. Comput., 21 (1992), pp. 1070–1099. A preliminary version appeared as *Asynchronous PRAMs are (almost) as good as synchronous PRAMs* in Proceeding of the 31st Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 590–599.

[23] S. Moran and Y. Wolfstahl, *Extended impossibility results for asynchronous complete networks*, Inform. Process. Lett., 26 (1987), pp. 145–151.

[24] J. R. Munkres, *Topology: A First Course*, Prentice–Hall, Englewood Cliffs, NJ, 1975.

[25] G. L. Peterson and J. E. Burns, *Concurrent reading while writing* II: *The multiwriter case*, in Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987.

[26] G. M. Reed, A. W. Roscoe, and R. F. Wachter, eds., *Topology and Category Theory in Computer Science*, Clarendon Press, Oxford, UK, 1991.

[27] S. Vickers, *Topology Via Logic*, Cambridge Tracts in Theoret. Comput. Sci. 6, Cambridge University Press, Cambridge, UK, 1997.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1484–1496

AN OPTIMAL ALGORITHM FOR MONTE CARLO ESTIMATION*

PAUL DAGUM†, RICHARD KARP‡, MICHAEL LUBY§, AND SHELDON ROSS¶

Abstract. A typical approach to estimate an unknown quantity μ is to design an experiment that produces a random variable Z distributed in $[0,1]$ with $\mathbf{E}[Z]=\mu$, run this experiment independently a number of times, and use the average of the outcomes as the estimate. In this paper, we consider the case when no a priori information about Z is known except that is distributed in $[0,1]$. We describe an approximation algorithm $\mathcal{AA}$ which, given ϵ and δ, when running independent experiments with respect to any Z, produces an estimate that is within a factor $1+\epsilon$ of μ with probability at least $1-\delta$. We prove that the expected number of experiments run by $\mathcal{AA}$ (which depends on Z) is optimal to within a constant factor for every Z.

Key words. stopping rule, approximation algorithm, Monte Carlo estimation, sequential estimation, stochastic approximation

AMS subject classifications. 60G40, 60G44

PII. S0097539797315306

1. Introduction.

The choice of experiment, or experimental design, forms an important aspect of statistics. One of the simplest design problems is that of deciding when to stop sampling. For example, suppose $Z_1, Z_2, \ldots$ are independently and identically distributed according to Z in the interval $[0,1]$ with mean μ_Z. From Bernstein's inequality, we know that if N is fixed proportional to $\ln(1/\delta)/\epsilon^2$ and $S_N = Z_1 + \cdots + Z_N$, then with probability at least $1-\delta$, S/N approximates μ_Z with absolute error ϵ. Often, however, μ_Z is small and a good absolute error estimate of μ_Z is typically a poor relative error approximation of μ_Z.

We say $\tilde{\mu}_Z$ is an (ϵ,δ)-approximation of μ_Z if

$$\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] \ge 1-\delta.$$

In engineering and computer science applications, we often desire an (ϵ,δ)-approximation of μ_Z in problems where exact computation of μ_Z is NP-hard. For example, many researchers have devoted substantial effort to the important and difficult problem of approximating the permanent of $0-1$ valued matrices [1, 4, 5, 9, 10, 13, 14]. Researchers have also used (ϵ,δ)-approximations to tackle many other difficult problems,

*Received by the editors January 20, 1997; accepted for publication (in revised form) December 9, 1998; published electronically March 15, 2000. An announcement of these results appears in P. Dagum, D. Karp, M. Luby, and S. Ross, An optimal algorithm for Monte Carlo Estimation (extended abstract), *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, 1995, pp. 142–149.

http://www.siam.org/journals/sicomp/29-5/31530.html

†Section on Medical Informatics, Stanford University School of Medicine, Stanford, CA 94305-5479 (dagum@smi.stanford.edu). The research of this author was supported in part by National Science Foundation operating grant IRI-93-11950.

‡International Computer Science Institute, Berkeley, CA 94704 (karp@icsi.berkeley.edu).

§International Computer Science Institute, Berkeley, CA, 94704, and Computer Science Division, University of California at Berkeley, Berkeley, CA 94704 (luby@icsi.berkeley.edu). The research of this author was supported in part by National Science Foundation operating grants CCR-9304722 and NCR-9416101, United States–Israel Binational Science Foundation grant 92-00226, and ESPRIT BR grant EC-US 030.

¶Department of Industrial Engineering and Operations Research, University of California at Berkeley, Berkeley, CA 94720 (smross@euler.berkeley.edu). The research of this author was supported in part by National Science Foundation operating grant DMI-9610046.

such as approximating probabilistic inference in Bayesian networks [6], approximating the volume of convex bodies [7], solving the Ising model of statistical mechanics [11], solving for network reliability in planar multiterminal networks [15, 16], approximating the number of solutions to a DNF formula [17] or, more generally, to a GF[2] formula [18], and approximating the number of Eulerian orientations of a graph [19].

Define

$$\lambda = (\mathrm{e} - 2) \approx .72,$$
$$\Upsilon = 4\lambda \ln(2/\delta)/\epsilon^2$$

and let σ_Z^2 denote the variance of Z. Define

$$\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}.$$

We first prove a slight generalization of the zero-one estimator theorem [12, 15, 16, 17]. The new theorem, the generalized zero-one estimator theorem, proves that if

$$N = \Upsilon \cdot \rho_Z/\mu_Z^2, \tag{1}$$

then S_N/N is an (ϵ, δ)-approximation of μ_Z.

To apply the generalized zero-one estimator theorem we require the values of the unknown quantities μ_Z and σ_Z^2. Researchers circumvent this problem by computing an upper bound κ on ρ_Z/μ_Z^2 and using κ in place of ρ_Z/μ_Z^2 to determine a value for N in (1). An a priori upper bound κ on ρ_Z/μ_Z^2 that is close to ρ_Z/μ_Z^2 is often very difficult to obtain, and a poor bound leads to a prohibitive large bound on N.

To avoid the problem encountered with the generalized zero-one estimator theorem, we use the outcomes of previous experiments to decide when to stop iterating. This approach is known as *sequential analysis* and originated with the work of Wald on statistical decision theory [22]. Related research has applied sequential analysis to specific Monte Carlo approximation problems such as estimating the number of points in a union of sets [17] and estimating the number of self-avoiding walks [20]. In other related work, Dyer et al. describe a stopping rule–based algorithm that provides an upper bound estimate on μ_Z [8]. With probability $1 - \delta$, the estimate is at most $(1 + \epsilon)\mu_Z$, but the estimate can be arbitrarily smaller than μ in the challenging case when μ is small.

We first describe an approximation algorithm based on a simple stopping rule. Using the stopping rule, the approximation algorithm outputs an (ϵ, δ)-approximation of μ_Z after an expected number of experiments proportional to Υ/μ_Z. The variance of the random variable Z is maximized subject to a fixed mean μ_Z if Z takes on value 1 with probability μ_Z and 0 with probability $1-\mu_Z$. In this case, $\sigma_Z^2 = \mu_Z(1-\mu_Z) \approx \mu_Z$, and the expected number of experiments run by the stopping rule–based algorithm is within a constant factor of optimal. In general, however, σ_Z^2 is significantly smaller than μ_Z, and for small values of σ_Z^2 the stopping rule–based algorithm performs $1/\epsilon$ times as many experiments as the optimal number.

We describe a more powerful algorithm, the $\mathcal{AA}$ algorithm that, on inputs ϵ, δ, and independently and identically distributed outcomes $Z_1, Z_2, \ldots$ generated from any random variable Z distributed in $[0, 1]$, outputs an (ϵ, δ)-approximation of μ_Z after an expected number of experiments proportional to $\Upsilon \cdot \rho_Z/\mu_Z^2$. Unlike the simple, stopping rule–based algorithm, we prove that for all Z, $\mathcal{AA}$ runs the optimal number of experiments to within a constant factor. Specifically, we prove that if $\mathcal{BB}$ is any algorithm that produces an (ϵ, δ)-approximation of μ_Z using the inputs ϵ, δ, and

$Z_1, Z_2, \dots$, then $\mathcal{BB}$ runs an expected number of experiments proportional to at least $\Upsilon \cdot \rho_Z/\mu_Z^2$. (Canetti, Even, and Goldreich prove the related lower bound $\Omega(\ln(1/\delta)/\epsilon^2)$ on the number of experiments required to approximate μ_Z with absolute error ϵ with probability at least $1-\delta$ [2].) Thus we show that for any random variable Z, $\mathcal{AA}$ runs an expected number of experiments that is within a constant factor of the minimum expected number.

The $\mathcal{AA}$ algorithm is a general method for optimally using the outcomes of Monte Carlo experiments for approximation—that is, to within a constant factor, the algorithm uses the minimum possible number of experiments to output an (ϵ, δ)-approximation on each problem instance. Thus, $\mathcal{AA}$ provides substantial computational savings in applications that employ a poor upper bound κ on ρ_Z/μ_Z^2. For example, the best known a priori bound on κ for the problem of approximating the permanent of size n is superpolynomial in n [13]. Yet for many problem instances of size n, the number of experiments run by $\mathcal{AA}$ is significantly smaller than this bound. Other examples exist where the bounds are also extremely loose for many typical problem instances [7, 10, 11]. In all those applications, we expect $\mathcal{AA}$ to provide substantial computational savings, and possibly render problems that were intractable because of the poor upper bounds on ρ_Z/μ_Z^2, amenable to efficient approximation.

2. Approximation algorithm.

In subsection 2.1, we describe a stopping rule algorithm for estimating μ_Z. This algorithm is used in the first step of the approximation algorithm $\mathcal{AA}$ that we describe in subsection 2.2.

2.1. Stopping Rule Algorithm.

Let Z be a random variable distributed in the interval $[0,1]$ with mean μ_Z. Let $Z_1, Z_2, \dots$ be independently and identically distributed according to Z.

Stopping Rule Algorithm.

```
Input parameters: $(\epsilon, \delta)$ with $0 < \epsilon < 1$, $\delta > 0$.
Let $\Upsilon_1 = 1 + (1+\epsilon)\Upsilon$.

Initialize $N \leftarrow 0$, $S \leftarrow 0$
While $S < \Upsilon_1$ do: $N \leftarrow N + 1$; $S \leftarrow S + Z_N$

Output: $\tilde{\mu}_Z \leftarrow \Upsilon_1/N$
```

Stopping Rule Theorem. *Let Z be a random variable distributed in $[0,1]$ with $\mu_Z = \mathbf{E}[Z] > 0$. Let $\tilde{\mu}_Z$ be the estimate produced and let N_Z be the number of experiments that the Stopping Rule Algorithm runs with respect to Z on input ϵ and δ. Then,*

(1) $\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] > 1-\delta$,

(2) $\mathbf{E}[N_Z] \le \Upsilon_1/\mu_Z$.

The proof of this theorem can be found in section 5.

2.2. Approximation Algorithm $\mathcal{AA}$.

The (ϵ, δ)-approximation algorithm $\mathcal{AA}$ consists of three main steps. The first step uses the stopping rule–based algorithm to produce an estimate $\hat{\mu}_Z$ that is within a constant factor of μ_Z with probability at least $1-\delta$. The second step uses the value of $\hat{\mu}_Z$ to set the number of experiments to run in order to produce an estimate $\hat{\rho}_Z$ that is within a constant factor of ρ with probability at least $1-\delta$. The third step uses the values of $\hat{\mu}_Z$ and $\hat{\rho}_Z$ produced in the first two steps to set the number of experiments and runs this number of experiments to produce an (ϵ, δ)-estimate of $\tilde{\mu}_Z$ of μ_Z.

Let Z be a random variable distributed in the interval $[0,1]$ with mean μ_Z and variance σ_Z^2. Let $Z_1, Z_2, \ldots$ and $Z'_1, Z'_2, \ldots$ denote two sets of random variables independently and identically distributed according to Z.

APPROXIMATION ALGORITHM $\mathcal{AA}$.

```
**Input Parameters:** $(\epsilon,\delta)$, with $0<\epsilon\le 1$ and $0<\delta\le 1$.
Let $\Upsilon_2 = 2(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})(1+\ln(\frac{3}{2})/\ln(\frac{2}{\delta}))\Upsilon \approx 2\Upsilon$ for small $\epsilon$ and $\delta$.

**Step 1:** Run the Stopping Rule Algorithm using $Z_1, Z_2 \ldots$ with input parameters
$\min\{1/2,\sqrt{\epsilon}\}$ and $\delta/3$. This produces an estimate $\hat{\mu}_Z$ of $\mu_Z$.

**Step 2:** Set $N = \Upsilon_2 \cdot \epsilon/\hat{\mu}_Z$ and initialize $S \leftarrow 0$.
For $i = 1,\ldots,N$ do: $S \leftarrow S + (Z'_{2i-1} - Z'_{2i})^2/2$.
$\hat{\rho}_Z \leftarrow \max\{S/N, \epsilon\hat{\mu}_Z\}$.

**Step 3:** Set $N = \Upsilon_2 \cdot \hat{\rho}_Z/\hat{\mu}_Z^2$ and initialize $S \leftarrow 0$.
For $i = 1,\ldots,N$ do: $S \leftarrow S + Z_i$.
$\tilde{\mu}_Z \leftarrow S/N$.

**Output:** $\tilde{\mu}_Z$
```

$\mathcal{AA}$ THEOREM. *Let Z be any random variable distributed in $[0,1]$, $\mu_Z = \mathbf{E}[Z] > 0$ be the mean of Z, σ_Z^2 be the variance of Z, and $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$. Let $\tilde{\mu}_Z$ be the approximation produced by $\mathcal{AA}$ and let N_Z be the number of experiments run by $\mathcal{AA}$ with respect to Z on input parameters ϵ and δ. Then,*

(1) $\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] \ge 1-\delta$.

(2) *There is a universal constant c' such that* $\mathbf{Pr}[N_Z \ge c'\Upsilon \cdot \rho_Z/\mu_Z^2] \le \delta$.

(3) *There is a universal constant c' such that* $\mathbf{E}[N_Z] \le c'\Upsilon\rho_Z/\mu_Z^2$.

We prove the $\mathcal{AA}$ theorem in section 6.

3. Lower bound. Algorithm $\mathcal{AA}$ is able to produce a good estimate of μ_Z using no a priori information about Z. An interesting question is, What is the inherent number of experiments needed to produce an (ϵ,δ)-approximation of μ_Z? In this section, we state a lower bound on the number of experiments needed by any (ϵ,δ)-approximation algorithm to estimate μ_Z when there is no a priori information about Z. This lower bound shows that, to within a constant factor, $\mathcal{AA}$ runs the minimum number of experiments for every random variable Z.

To formalize the lower bound, we introduce the following natural model. Let $\mathcal{BB}$ be any algorithm that on input (ϵ,δ) works as follows with respect to Z. Let $Z_1, Z_2, \ldots$ denote independently and identically distributed according to Z with values in the interval $[0,1]$. $\mathcal{BB}$ runs an experiment, and on the Nth run $\mathcal{BB}$ receives the value Z_N. The measure of the running time of $\mathcal{BB}$ is the number of experiments it runs, i.e., the time for all other computations performed by $\mathcal{BB}$ is not counted in its running time. $\mathcal{BB}$ is allowed to use any criteria it wants to decide when to stop running experiments and produce an estimate, and in particular $\mathcal{BB}$ can use the outcome of all previous experiments. The estimate that $\mathcal{BB}$ produces when it stops can be any function of the outcomes of the experiments it has run up to that point. The requirement of $\mathcal{BB}$ is that it produces an (ϵ,δ)-approximation of μ_Z for any Z.

This model captures the situation where the algorithm can only gather information about μ_Z through running random experiments and where the algorithm has no a priori knowledge about the value of μ_Z before starting. This is a reasonable pair of assumptions for practical situations. It turns out that the assumption about a priori knowledge can be substantially relaxed: the algorithm may know a priori that the outcomes are being generated according to some known random variable Z or to some closely related random variable Z', and still the lower bound on the number of experiments applies.

Note that Algorithm $\mathcal{AA}$ fits into this model, and thus the average number of experiments it runs with respect to Z is minimal for all Z to within a constant factor among all such approximation algorithms.

Lower Bound Theorem. *Let $\mathcal{BB}$ be any algorithm that works as described above on input (ϵ, δ). Let Z be a random variable distributed in $[0,1]$, μ_Z be the mean of Z, σ_Z^2 be the variance of Z, and $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$. Let $\tilde{\mu}_Z$ be the approximation produced by $\mathcal{BB}$ and let N_Z be the number of experiments run by $\mathcal{BB}$ with respect to Z. Suppose that $\mathcal{BB}$ has the the following properties:*

(1) *For all Z with $\mu_Z > 0$, $\mathbf{E}[N_Z] < \infty$.*

(2) *For all Z with $\mu_Z > 0$,*

$$\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] > 1 - \delta.$$

Then, there is a universal constant $c > 0$ such that for all Z, $\mathbf{E}[N_Z] \ge c\Upsilon \cdot \rho_Z/\mu_Z^2$.

We prove this theorem in section 7.

4. Preliminaries for the proofs.

We begin with some notation that is used hereafter. Let $\xi_0 = 0$ and for $k > 0$ let

$$\xi_k = \sum_{i=1}^{k}(Z_i - \mu_Z). \tag{2}$$

For fixed $\alpha, \beta \ge 0$, we define the random variables

$$\zeta_k^+ = \xi_k - \alpha - \beta k, \tag{3}$$

$$\zeta_k^- = -\xi_k - \alpha - \beta k. \tag{4}$$

The main lemma we use to prove the first part of the Stopping Rule Theorem provides bounds on the probabilities that the random variables ζ_k^+ and ζ_k^- are greater than zero. We first form the sequences of random variables $\mathrm{e}^{d\zeta_0^+}, \mathrm{e}^{d\zeta_1^+}, \ldots$ and $\mathrm{e}^{d\zeta_0^-}, \mathrm{e}^{d\zeta_1^-}, \ldots$ for any real-valued d. We prove that these sequences are *supermartingales* when $0 \le d \le 1$ and $\beta \ge d\sigma_Z^2$, i.e., for all $k > 0$

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^+} | \mathrm{e}^{d\zeta_{k-1}^+}, \ldots, \mathrm{e}^{d\zeta_0^+}] \le \mathrm{e}^{d\zeta_{k-1}^+},$$

and similarly,

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^-} | \mathrm{e}^{d\zeta_{k-1}^-}, \ldots, \mathrm{e}^{d\zeta_0^-}] \le \mathrm{e}^{d\zeta_{k-1}^-}.$$

We then use properties of supermartingales to bound the probabilities that the random variables ζ_k^+ and ζ_k^- are greater than zero. For these and subsequent proofs, we use the following two inequalities.

Inequality 4.1. *For all α, $\mathrm{e}^{\alpha} \ge 1 + \alpha$.*

Inequality 4.2. *Let $\lambda = (\mathrm{e} - 2) \approx .72$. For all α with $|\alpha| \le 1$,*

$$1 + \alpha + \alpha^2/(2+\lambda) \le \mathrm{e}^{\alpha} \le 1 + \alpha + \lambda \cdot \alpha^2.$$

LEMMA 4.3. *For $|d| \leq 1$, $\mathbf{E}[\mathrm{e}^{dZ}] \leq \mathrm{e}^{d\mu_Z + \lambda d^2 \sigma_Z^2}$.*

Proof. Observe that $E[e^{dZ}] = \mathrm{e}^{d\mu_Z} E[\mathrm{e}^{d(Z-\mu_Z)}]$. But from Inequality 4.2,

$$\mathrm{e}^{d(Z-\mu_Z)} \leq 1 + d(Z-\mu_Z) + \lambda d^2 (Z-\mu_Z)^2.$$

Taking expectations and applying Inequality 4.1 completes the proof. □

LEMMA 4.4. *For $0 \leq d \leq 1$, and for $\beta \geq \lambda d \sigma_Z^2$, the sequences of random variables $\mathrm{e}^{d\zeta_0^+}, \mathrm{e}^{d\zeta_1^+}, \ldots$ and $\mathrm{e}^{d\zeta_0^-}, \mathrm{e}^{d\zeta_1^-}, \ldots$ form supermartingales.*

Proof. For $k \geq 1$

$$\mathrm{e}^{d\zeta_k^+} = \mathrm{e}^{d\zeta_{k-1}^+} \cdot \mathrm{e}^{-d\beta} \cdot \mathrm{e}^{d(\xi_k - \xi_{k-1})}$$

and thus

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^+} | \mathrm{e}^{d\zeta_{k-1}^+}, \ldots, \mathrm{e}^{d\zeta_0^+}] = \mathrm{e}^{d\zeta_{k-1}^+} \cdot \mathrm{e}^{-d\beta} \mathbf{E}[\mathrm{e}^{d(\xi_k - \xi_{k-1})}].$$

Similarly, for $k \geq 1$

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^-} | \mathrm{e}^{d\zeta_{k-1}^-}, \ldots, \mathrm{e}^{d\zeta_0^-}] = \mathrm{e}^{d\zeta_{k-1}^-} \cdot \mathrm{e}^{-d\beta} \mathbf{E}[\mathrm{e}^{-d(\xi_k - \xi_{k-1})}].$$

But $\xi_k - \xi_{k-1} = Z_k - \mu_Z$ and thus from Lemma 4.3,

$$\mathbf{E}[\mathrm{e}^{d(\xi_k - \xi_{k-1})}] \leq \mathrm{e}^{\lambda d^2 \sigma_Z^2}$$

and

$$\mathbf{E}[\mathrm{e}^{-d(\xi_k - \xi_{k-1})}] \leq \mathrm{e}^{\lambda d^2 \sigma_Z^2}.$$

Thus, for $\beta \geq \lambda d \sigma_Z^2$,

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^+} | \mathrm{e}^{d\zeta_{k-1}^+}, \ldots, \mathrm{e}^{d\zeta_0^+}] \leq \mathrm{e}^{d\zeta_{k-1}^+} \cdot \mathrm{e}^{d(\lambda d \sigma_Z^2 - \beta)} \leq \mathrm{e}^{d\zeta_{k-1}^+}$$

and

$$\mathbf{E}[\mathrm{e}^{d\zeta_k^-} | \mathrm{e}^{d\zeta_{k-1}^-}, \ldots, \mathrm{e}^{d\zeta_0^-}] \leq \mathrm{e}^{d\zeta_{k-1}^-} \cdot \mathrm{e}^{d(\lambda d \sigma_Z^2 - \beta)} \leq \mathrm{e}^{d\zeta_{k-1}^-}. \qquad \square$$

Lemma 4.5, needed for the proof of Lemma 4.6, follows directly from the properties of conditional expectations and of martingales.

LEMMA 4.5. *If $\eta_0, \ldots, \eta_k$ is a supermartingale, then for all $0 \leq i \leq k$*

$$\mathbf{E}[\eta_i | \eta_0] \leq \eta_0. \qquad \square$$

We next prove Lemma 4.6. This lemma is the key to the proof of the first part of the Stopping Rule Theorem. In addition, from this lemma we easily prove a slightly more general version of the zero-one estimator theorem.

LEMMA 4.6. *For any fixed $N > 0$, for any $\beta \leq 2\lambda\rho_Z$,*

$$\mathbf{Pr}[\xi_N / N \geq \beta] \leq \mathrm{e}^{\frac{-N\beta^2}{4\lambda\rho_Z}} \tag{5}$$

and

$$\mathbf{Pr}[\xi_N / N \leq -\beta] \leq \mathrm{e}^{\frac{-N\beta^2}{4\lambda\rho_Z}}. \tag{6}$$

Proof. Recall the definitions of ζ_N^+ and ζ_N^- from (3) and (4). Let $\alpha = 0$. Then, the left-hand side of (5) is equivalent to $\mathbf{Pr}[\zeta_N^+ \geq 0]$ and the left-hand side of (6) is equivalent to $\mathbf{Pr}[\zeta_N^- \geq 0]$. Let $\alpha' = \beta N/2$ and $\beta' = \beta/2$. For $0 \leq i \leq N$, let $\zeta_i'^+ = \xi_i - \alpha' - \beta' i$ and $\zeta_i'^- = -\xi_i - \alpha' - \beta' i$. Thus, $\zeta_N'^+ = \zeta_N^+$ and $\zeta_N'^- = \zeta_N^-$.

We now give the remainder of the proof of (5), using $\zeta_N'^+$, and omit the remainder of the analogous proof of (6) which uses $\zeta_N'^-$ in place of $\zeta_N'^+$. For any $N > 0$,

$$\mathbf{Pr}[\zeta_N'^+ \geq 0] = \mathbf{Pr}[\mathrm{e}^{d\zeta_N'^+} \geq 1] \leq \mathbf{E}[\mathrm{e}^{d\zeta_N'^+}].$$

Set $d = \beta'/(\lambda\rho_Z) = \beta/(2\lambda\rho_Z)$. Note that $\beta \leq 2\lambda\rho_Z$ implies that $d \leq 1$. Note also that since $\rho_Z \geq \sigma_Z^2$, $\beta' \geq \lambda d \sigma_Z^2$. Thus, by Lemma 4.4, $\mathrm{e}^{d\zeta_0'^+}, \ldots, \mathrm{e}^{d\zeta_N'^+}$ is a supermartingale. Thus, by Lemma 4.5

$$\mathbf{E}[\mathrm{e}^{d\zeta_N'^+} | \mathrm{e}^{d\zeta_0'^+}] \leq \mathrm{e}^{d\zeta_0'^+} = \mathrm{e}^{\frac{-\alpha'\beta'}{\lambda\rho_Z}} = \mathrm{e}^{\frac{-N\beta^2}{4\lambda\rho_Z}}.$$

Since $\mathrm{e}^{d\zeta_0'^+}$ is a constant,

$$\mathbf{E}[\mathrm{e}^{d\zeta_N'^+} | \mathrm{e}^{d\zeta_0'^+}] = \mathbf{E}[\mathrm{e}^{d\zeta_N'^+}],$$

completing the proof of (5). □

We use Lemma 4.6 to generalize the zero-one estimator theorem [17] from $\{0,1\}$-valued random variables to random variables in the interval $[0,1]$.

Generalized zero-one estimator theorem. *Let $Z_1, Z_2, \ldots, Z_N$ denote random variables that are independent and identically distributed according to Z. If $\epsilon < 1$ and*

$$N = 4\lambda \ln(2/\delta) \cdot \rho_Z/(\epsilon\mu_Z)^2,$$

then

$$\mathbf{Pr}\left[(1-\epsilon)\mu_Z \leq \sum_{i=1}^{N} Z_i/N \leq (1+\epsilon)\mu_Z\right] > 1 - \delta.$$

Proof. The proof follows directly from Lemma (4.6), using $\beta = \epsilon\mu_Z$, noting that $\epsilon\mu_Z \leq 2\lambda\rho_Z$ and that $N \cdot (\epsilon\mu_Z)^2/(4\lambda\rho_Z) = \ln(2/\delta)$. □

5. Proof of the Stopping Rule Theorem.

We next prove the Stopping Rule Theorem. Recall that $\lambda = (\mathrm{e}-2) \approx .72$ and $\Upsilon_1 = 1 + (1+\epsilon)\Upsilon = 1 + 4\lambda\ln(2/\delta)(1+\epsilon)/\epsilon^2$.

Proof of part (1). Recall that $\tilde{\mu}_Z = \Upsilon_1/N_Z$. It suffices to show that

$$\mathbf{Pr}[N_Z < \Upsilon_1/(\mu_Z(1+\epsilon))] + \mathbf{Pr}[N_Z > \Upsilon_1/(\mu_Z(1-\epsilon))] \leq \delta.$$

We first show that $\mathbf{Pr}[N_Z < \Upsilon_1/(\mu_Z(1+\epsilon))] \leq \delta/2$. Let $L = \lfloor \Upsilon_1/(\mu_Z(1+\epsilon)) \rfloor$. Assuming that $\mu_Z(1+\epsilon) \leq 1$, the definition of Υ_1 and L implies that

$$L \geq \frac{4\lambda\ln(2/\delta)}{\epsilon^2\mu_Z}. \tag{7}$$

Since N_Z is an integer, $N_Z < \Upsilon_1/(\mu_Z(1+\epsilon))$ if and only if $N_Z \leq L$. But $N_Z \leq L$ if and only if $S_L \geq \Upsilon_1$. Thus,

$$\mathbf{Pr}[N_Z < \Upsilon_1/(\mu_Z(1+\epsilon))] = \mathbf{Pr}[N_Z \leq L] = \mathbf{Pr}[S_L \geq \Upsilon_1].$$

Let $\beta = \Upsilon_1/L - \mu_Z$. Then,

$$\mathbf{Pr}[S_L \geq \Upsilon_1] = \mathbf{Pr}[S_L - \mu_Z L - \beta L \geq 0] = \mathbf{Pr}[\xi_L/L \geq \beta].$$

Noting that $\epsilon\mu_Z \leq \beta \leq 2\lambda\rho_Z$, Lemma (4.6) implies that

$$\mathbf{Pr}[\xi_L/L \geq \beta] \leq \mathrm{e}^{\frac{-L\beta^2}{4\lambda\rho_Z}} \leq \mathrm{e}^{\frac{-L(\epsilon\mu_Z)^2}{4\lambda\rho_Z}}.$$

Using inequality (7) and noting that $\rho_Z \leq \mu_Z$, it follows that this is at most $\delta/2$.

The proof that $\mathbf{Pr}[N_Z > \Upsilon_1/(\mu_Z(1-\epsilon))] \leq \delta/2$ is similar. □

Proof of part (2). The random variable N_Z is the stopping time such that

$$\Upsilon_1 \leq S_{N_Z} < \Upsilon_1 + 1.$$

Using Wald's equation [22] and $\mathbf{E}[N_Z] < \infty$, it follows that

$$\mathbf{E}[S_{N_Z}] = \mathbf{E}[N_Z]\mu_Z$$

and thus,

$$\Upsilon_1/\mu_Z \leq \mathbf{E}[N_Z] < (\Upsilon_1 + 1)/\mu_Z. \qquad \square$$

Similar to the proof of the first part of the Stopping Rule Theorem, we can show that

$$\mathbf{Pr}[N_Z > (1+\epsilon)\Upsilon_1/\mu_Z] \leq \delta/2, \tag{8}$$

and therefore with probability at least $1 - \delta/2$ we require at most $(1+\epsilon)\Upsilon_1/\mu_Z$ experiments to generate an approximation. The following lemma is used in the proof of the $\mathcal{AA}$ Theorem in section 6.

Stopping Rule Lemma.

(1) $\mathbf{E}[1/\tilde{\mu}_Z] = \mathcal{O}(1/\mu_Z)$.

(2) $\mathbf{E}[1/\tilde{\mu}_Z^2] = \mathcal{O}(1/\mu_Z^2)$.

Proof of the Stopping Rule Lemma. $\mathbf{E}[1/\tilde{\mu}_Z] = \mathcal{O}(1/\mu_Z)$ follows directly from part (2) of the Stopping Rule Theorem and the definition of N_Z. $\mathbf{E}[1/\tilde{\mu}_Z^2] = \mathcal{O}(1/\mu_Z^2)$ can be easily proved based on the ideas used in the proof of part (2) of the Stopping Rule Theorem. □

6. Proof of the $\mathcal{AA}$ Theorem.

$\mathcal{AA}$ theorem. *Let Z be any random variable distributed in $[0,1]$, μ_Z be the mean of Z, σ_Z^2 be the variance of Z, and $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$. Let $\tilde{\mu}_Z$ be the approximation produced by $\mathcal{AA}$ and let N_Z be the number of experiments run by $\mathcal{AA}$ with respect to Z on input parameters ϵ and δ. Then*

(1) $\mathbf{Pr}[\mu_Z(1-\epsilon) \leq \tilde{\mu}_Z \leq \mu_Z(1+\epsilon)] \geq 1-\delta$,

(2) *there is a universal constant c' such that* $\mathbf{Pr}[N_Z \geq c'\Upsilon \cdot \rho_Z/\mu_Z^2] \leq \delta$,

(3) *there is a universal constant c' such that* $\mathbf{E}[N_Z] \leq c'\Upsilon\rho_Z/\mu_Z^2$.

Proof of part (1). From the Stopping Rule Theorem, after step (1) of $\mathcal{AA}$, $\mu_Z(1-\sqrt{\epsilon}) \leq \hat{\mu}_Z \leq \mu_Z(1+\sqrt{\epsilon})$ holds with probability at least $1-\delta/3$. Let $\Phi = 2(1+\sqrt{\epsilon})^2$. We show next that if $\mu_Z(1-\sqrt{\epsilon}) \leq \hat{\mu}_Z \leq \mu_Z(1+\sqrt{\epsilon})$, then in step (2) the choice of Υ_2 guarantees that $\hat{\rho}_Z \geq \rho_Z/2$. Thus, after steps (1) and (2), $\Phi\hat{\rho}_Z/\hat{\mu}_Z^2 \geq \rho_Z/\mu_Z^2$ with probability at least $1-\delta/3$. But by the generalized zero-one estimator theorem, for $N = (1+\ln(\frac{3}{2})/\ln(\frac{2}{\delta}))\Upsilon \cdot \rho_Z/\mu_Z^2 \leq \Phi(1+\ln(\frac{3}{2})/\ln(\frac{2}{\delta}))\Upsilon \cdot \hat{\rho}_Z/\hat{\mu}_Z^2 \leq \Upsilon_2 \cdot \hat{\rho}_Z/\hat{\mu}_Z^2$, step

(3) guarantees that the output $\tilde{\mu}_Z$ of $\mathcal{AA}$ satisfies $\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] \ge 1 - 2\delta/3$.

For all i, let $\xi_i = (Z_{2i-1} - Z_{2i})^2/2$, and observe that, $\mathbf{E}[\xi] = \sigma_Z^2$. First assume that $\sigma_Z^2 \ge \epsilon\mu_Z$, and hence $\rho_Z = \sigma_Z^2$. If $\sigma_Z^2 \ge 2(1-\sqrt{\epsilon})\epsilon\mu_Z$, then from the generalized zero-one estimator theorem, after at most $(2/(1-\sqrt{\epsilon}))(1+\ln(\frac{3}{2})/\ln(\frac{2}{\delta}))\cdot\Upsilon\cdot\epsilon/\mu_Z \le \Upsilon_2\cdot\epsilon/\hat{\mu}_Z$ experiments, $\rho_Z/2 \le S/N \le 3\rho_Z/2$ with probability at least $1-2\delta/3$. Thus $\hat{\rho}_Z \ge \rho_Z/2$. If $\epsilon\mu_Z \le \sigma_Z^2 \le 2(1-\sqrt{\epsilon})\epsilon\mu_Z$, then $\epsilon\mu_Z \ge \sigma_Z^2/(2(1-\sqrt{\epsilon}))$, and therefore, $\hat{\rho}_Z \ge \epsilon\hat{\mu}_Z \ge \rho_Z/2$.

Next, assume that $\sigma_Z^2 \le \epsilon\mu_Z$ and thus $\rho_Z = \epsilon\mu_Z$. But steps (1) and (2) guarantee that $\hat{\rho}_Z \ge \epsilon\hat{\mu}_Z \ge \rho_Z(1-\sqrt{\epsilon})$, with probability at least $1-\delta/3$.

Proof of part (2). $\mathcal{AA}$ may fail to terminate after $\mathcal{O}(\Upsilon\cdot\rho_Z/\mu_Z^2)$ experiments either because step (1) failed with probability at least $\delta/2$ to produce an estimate $\hat{\mu}_Z$ such that $\mu_Z(1-\sqrt{\epsilon}) \le \hat{\mu}_Z \le \mu_Z(1+\sqrt{\epsilon})$, or, because in step (2), for $\sigma_Z^2 \le 2(1-\sqrt{\epsilon})\epsilon\mu_Z$, $\hat{\rho}_Z = S/N$ and S/N is not $\mathcal{O}(\epsilon\mu_Z)$ with probability at least $1-\delta/2$.

But (8) guarantees that step (1) of $\mathcal{AA}$ terminates after $\mathcal{O}(\Upsilon\cdot\rho_Z/\mu_Z^2)$ experiments with probability at least $1-\delta/2$. In addition, we can show similarly to Lemma 4.6, that if $\sigma_Z^2 \le 2\epsilon\mu_Z$, then

$$\mathbf{Pr}[S/N \ge 4\epsilon\mu_Z] \le \mathrm{e}^{-N\epsilon\mu_Z/2}.$$

Thus, for $N \ge 2\Upsilon\cdot\epsilon/\mu_Z$, we have that $\mathbf{Pr}[S/N \ge 4\epsilon\mu_Z] \le \delta/2$.

Proof of part (3). Observe that from the Stopping Rule Theorem, the expected number of experiments in step (1) is $\mathcal{O}(\ln(1/\delta)/(\epsilon\mu_Z))$. From the Stopping Rule Lemma, the expected number of experiments in step (2) is $\mathcal{O}(\ln(1/\delta)/(\epsilon\mu_Z))$. Finally, in step (3) we observe that $\mathbf{E}[\hat{\rho}_Z/\hat{\mu}_Z^2] = \mathbf{E}[\hat{\rho}_Z]\mathbf{E}[1/\hat{\mu}_Z^2]$ since $\hat{\rho}_Z$ and $\hat{\mu}_Z$ are computed from disjoint sets of independently and identically distributed random variables. From the Stopping Rule Lemma $\mathbf{E}[1/\hat{\mu}_Z^2]$ is $\mathcal{O}(\ln(1/\delta)/\mu_Z^2)$. Furthermore, observe that $\mathbf{E}[\hat{\rho}_Z] \le \mathbf{E}[S/N] + \mathbf{E}[\epsilon\hat{\mu}_Z]$. But $\mathbf{E}[S/N] = \sigma_Z^2$ and $\mathbf{E}[\epsilon\hat{\mu}_Z] = \mathcal{O}(\epsilon\mu_Z)$. Thus, if $\sigma_Z^2 \ge \epsilon\mu_Z$, then $\rho_Z = \sigma_Z^2$ and $\mathbf{E}[\hat{\rho}_Z] = \mathcal{O}(\sigma_Z^2) = \mathcal{O}(\rho_Z)$. If $\sigma_Z^2 \le \epsilon\mu_Z$, then $\rho_Z = \epsilon\mu_Z$ and $\mathbf{E}[\hat{\rho}_Z] = \mathcal{O}(\epsilon\mu_Z) = \mathcal{O}(\rho_Z)$. Thus, the expected number of experiments in step (3) is $\mathcal{O}(\ln(1/\delta)\cdot\rho_Z/(\epsilon\mu_Z)^2)$ □

7. Proof of the Lower Bound Theorem.

LOWER BOUND THEOREM. *Let $\mathcal{BB}$ be any algorithm that works as described above on input (ϵ,δ). Let Z be a random variable distributed in $[0,1]$, μ_Z be the mean of Z, σ_Z^2 be the variance of Z, and $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$. Let $\tilde{\mu}_Z$ be the approximation produced by $\mathcal{BB}$ and let N_Z be the number of experiments run by $\mathcal{BB}$ with respect to Z. Suppose that $\mathcal{BB}$ has the the following properties:*

(1) *For all Z with $\mu_Z > 0$, $\mathbf{E}[N_Z] < \infty$.*

(2) *For all Z with $\mu_Z > 0$, $\mathbf{Pr}[\mu_Z(1-\epsilon) \le \tilde{\mu}_Z \le \mu_Z(1+\epsilon)] > 1-\delta$.*

Then there is a universal constant $c > 0$ such that for all Z, $\mathbf{E}[N_Z] \ge c\Upsilon\cdot\rho_Z/\mu_Z^2$.

Let $f_Z(x)$ and $f_{Z'}(x)$ denote two given distinct probability mass (or in the continuous case, density) functions. Let $Z_1, Z_2, \ldots$ denote independent and identically distributed random variables with probability density $f(x)$. Let H_Z denote the hypothesis $f = f_Z$ and let $H_{Z'}$ denote the hypothesis $f = f_{Z'}$. Let α denote the probability that we reject H_Z under f_Z and let β denote the probability that we accept $H_{Z'}$ under $f_{Z'}$.

The sequential probability ratio test minimizes the number of expected sample size under both H_Z and $H_{Z'}$ among all tests with the same error probabilities α and β. Theorem 7.1 states the result of the sequential probability ratio test. We prove the result for completeness, although similar proofs exist [21].

THEOREM 7.1. *If T is the stopping time of any test of H_Z against $H_{Z'}$ with error probabilities α and β, and $\mathbf{E}_Z[T], \mathbf{E}_{Z'}[T] < \infty$, then*

$$\mathbf{E}_Z[T] \geq \frac{1}{\omega_Z}\left(\alpha \ln \frac{1-\beta}{\alpha} + (1-\alpha)\ln\frac{\beta}{1-\alpha}\right)$$

and

$$\mathbf{E}_{Z'}[T] \geq \frac{1}{\omega_{Z'}}\left((1-\beta) \ln \frac{1-\beta}{\alpha} + \beta\ln\frac{\beta}{1-\alpha}\right),$$

where $\omega_Z = \mathbf{E}_Z[\ln(f_{Z'}(x)/f_Z(x))]$ and $\omega_{Z'} = \mathbf{E}_{Z'}[\ln(f_{Z'}(x)/f_Z(x))]$.

Proof. For the independent and identically distributed random variables $Z_1, Z_2, \ldots$, let $\lambda_k(Z_k) = \ln(f_{Z'}(Z_k)/f_Z(Z_k))$. Define $\zeta_k^+ = \lambda_k + \cdots + \lambda_1$ and $\zeta_k^- = \zeta_k^+$.

For stopping time T, we get from Wald's first identity

$$\mathbf{E}_Z[\zeta_T^+] = \mathbf{E}_Z[T]\mathbf{E}_Z[\lambda_1]$$

and

$$\mathbf{E}_{Z'}[\zeta_T^-] = \mathbf{E}_{Z'}[T]\mathbf{E}_{Z'}[\lambda_1].$$

Next, let Ω denote the space of all inputs on which the test rejects H_Z, and let Ω^c denote its complement. Thus by definition, we require that $\mathbf{Pr}_Z[\Omega] = \alpha$ and $\mathbf{Pr}_Z[\Omega^c] = 1-\alpha$. Similarly, we require that $\mathbf{Pr}_{Z'}[\Omega] = 1-\beta$ and $\mathbf{Pr}_{Z'}[\Omega^c] = \beta$. From the properties of expectations we can show that

$$\mathbf{E}_Z[\zeta_T^+] = \mathbf{E}_Z[\zeta_T^+|\Omega]\mathbf{Pr}_Z[\Omega] + \mathbf{E}_Z[\zeta_T^+|\Omega^c]\mathbf{Pr}_Z[\Omega^c].$$

We can decompose $\mathbf{E}_{Z'}[\zeta_T^-]$ similarly. Let $\mu = \mathbf{E}_Z[\zeta_T^+|\Omega]$ and observe that from Inequality 4.1, $\mathbf{E}_Z[\mathrm{e}^{\zeta_T^+ - \mu}|\Omega] \geq 1$. Thus

$$\mu \leq \ln \mathbf{E}_Z[\mathrm{e}^{\zeta_T^+}|\Omega].$$

But

$$\mathbf{E}_Z[\mathrm{e}^{\zeta_T^+}|\Omega] = \mathbf{E}_Z[\mathrm{e}^{\zeta_T^+}I_\Omega]/\mathbf{Pr}_Z[\Omega],$$

where I_Ω denotes the characteristic function for the set Ω. Thus, since

$$\mathrm{e}^{\zeta_T^+} = \prod_{i=1}^{T}\frac{f_{Z'}(Z_i)}{f_Z(Z_i)},$$

we can show that

$$\mathbf{E}_Z[\mathrm{e}^{\zeta_T^+}I_\Omega] = \mathbf{Pr}_{Z'}[\Omega],$$

and finally,

$$\mathbf{E}_Z[\zeta_T^+|\Omega] \leq \ln \mathbf{E}_Z[\mathrm{e}^{\zeta_T^+}|\Omega] = \ln\frac{1-\beta}{\alpha}.$$

Similarly, we can show that

$$\mathbf{E}_Z[\zeta_T^+|\Omega^c] \leq \ln \frac{\beta}{1-\alpha},$$
$$\mathbf{E}_{Z'}[\zeta_T^-|\Omega] \leq \ln \frac{\alpha}{1-\beta},$$

and

$$\mathbf{E}_{Z'}[\zeta_T^-|\Omega^c] \leq \ln \frac{1-\alpha}{\beta}.$$

Thus,

$$-\mathbf{E}_Z[T]\mathbf{E}_Z[\lambda_0] \leq \alpha \ln \frac{1-\beta}{\alpha} + (1-\alpha) \ln \frac{\beta}{1-\alpha},$$

which proves the first part of the lemma. Similarly,

$$-\mathbf{E}_{Z'}[T]\mathbf{E}_{Z'}[\lambda_0] \leq (1-\beta) \ln \frac{\alpha}{1-\beta} + \beta \ln \frac{1-\alpha}{\beta},$$

proves the second part of the lemma. □

COROLLARY 7.2. *If T is the stopping time of any test of H_Z against $H_{Z'}$ with error probabilities α and β such that $\alpha + \beta = \delta$, then*

$$\mathbf{E}_Z[T] \geq -\frac{1-\delta}{\omega_Z} \ln \frac{2-\delta}{\delta}$$

and

$$\mathbf{E}_{Z'}[T] \geq \frac{1-\delta}{\omega_{Z'}} \ln \frac{2-\delta}{\delta}.$$

Proof. If $\alpha + \beta = \delta$, then

$$(1-\beta) \ln \frac{1-\beta}{\alpha} + \beta \ln \frac{\beta}{1-\alpha} = -\alpha \ln \frac{1-\beta}{\alpha} - (1-\alpha) \ln \frac{\beta}{1-\alpha}$$

achieves a minimum at $\alpha = \beta = \delta/2$. Substitution of $\alpha = \beta = \delta/2$ completes the proof. □

Before we present Lemma 7.5 that proves the Lower Bound Theorem for $\sigma_Z^2 \geq \epsilon\mu_Z$, we begin with some definitions. Let $\xi = Z - \mu_Z$ and for any $0 \leq d \leq 1$ let

$$\psi = \mathbf{E}_0[\mathrm{e}^{d\xi}].$$

Define $\zeta = d\xi - \ln \psi$ and

$$f_{Z'}(x) = f_Z(x) \cdot e^{\zeta}.$$

LEMMA 7.3. $1 + d^2\sigma_Z^2/(2+\lambda) \leq \psi \leq 1 + \lambda d^2 \sigma_Z^2$.

Proof. Since $\psi = \mathbf{E}_Z[\mathrm{e}^{d\xi}]$ we use Inequality 4.2 to show that

$$1 + d\xi + d^2\xi^2/(2+\lambda) \leq \mathrm{e}^{d\xi} \leq 1 + d\xi + \lambda d^2 \xi^2.$$

Taking expectations completes the proof. ∎

LEMMA 7.4. $2d\sigma_Z^2/((2+\lambda)\psi) \le \mathbf{E}_{Z'}[\xi] \le 2\lambda d\sigma_Z^2/\psi$.

Proof. Since $\mathbf{E}_{Z'}[\xi] = \psi^{-1}\psi'$, where ψ' denotes the derivative of ψ with respect to d, the proof follows directly from Lemma 7.3. ∎

LEMMA 7.5. *If* $\sigma_Z^2 \ge \epsilon\mu_Z$, *then for* $\epsilon < 1$, $\mathbf{E}[N_Z] \ge (1-\delta)(1-\epsilon)^2 \ln(\frac{2-\delta}{\delta}) \cdot \sigma_Z^2/((2+\lambda)2\epsilon\mu_Z)^2$,

Proof. Let T denote the stopping time of any test of H_Z against $H_{Z'}$. Note that $\mathbf{E}_{Z'}[Z] - \mathbf{E}_Z[Z] = \mathbf{E}_{Z'}[Z] - \mu_Z = \mathbf{E}_{Z'}[\xi]$. If we set $d = \epsilon\mu_Z/\sigma_Z^2$, then, by Lemmas 7.3 and 7.4, $\mathbf{E}_{Z'}[Z] - \mu_Z > \epsilon\mu_Z/(2+\lambda)$. Thus to test H_Z against H_Z', we can use the $\mathcal{BB}$ with input ϵ^* such that $\mu_Z(1+\epsilon^*) \le \mu_Z(1+\epsilon/(2+\lambda))(1-\epsilon^*) < \mu_{Z'}(1-\epsilon^*)$. Solving for ϵ^* we obtain $\epsilon^* \le \epsilon/(2(2+\lambda)+\epsilon)$. But Corollary 7.2 gives a lower bound on the expected number of experiments $\mathbf{E}[N_Z^*]$ run by $\mathcal{BB}$ with respect the Z. We observe that

$$-\omega_Z = \mathbf{E}_Z[\zeta] = \ln\psi \le d^2\sigma_Z^2,$$

where the inequality follows from Lemma 7.3. We let $d = \epsilon\mu_Z/\sigma_Z^2$ and substitute $\epsilon = 2(2+\lambda)\epsilon^*/(1-\epsilon^*)$ to complete the proof. ∎

We now prove the Lower Bound Theorem that holds also when $\sigma_Z^2 < \epsilon\mu_Z$. We define the density

$$f_{Z'}(x) = f_Z(x)\cdot(1-\epsilon\mu_Z) + \epsilon\mu_Z.$$

LEMMA 7.6. *If* $\mu_Z \le 1/4$, *then* $\mathbf{E}_{Z'}[Z] - \mu_Z \ge \epsilon\mu_Z/4$.

Proof. Observe that $\mathbf{E}_{Z'}[Z] = (1-\epsilon\mu_Z)\mathbf{E}_Z[Z] + \epsilon\mu_Z/2$ and therefore $\mathbf{E}_{Z'}[Z] - \mu_Z = \epsilon\mu_Z(1/2-\mu_Z)$. ∎

LEMMA 7.7. $-\mathbf{E}_Z[\ln\frac{f_{Z'}(x)}{f_Z(x)}] \le \frac{\epsilon\mu_Z}{1-\epsilon\mu_Z}$.

Proof. Observe that

$$-\ln\frac{f_{Z'}(x)}{f_Z(x)} \le -\ln(1-\epsilon\mu_Z) \le \frac{\epsilon\mu_Z}{1-\epsilon\mu_Z}.$$

Taking expectations completes the proof. ∎

LEMMA 7.8. *If* $\mu_Z \le 1/4$, *then* $\mathbf{E}[N_Z] \ge (1-\delta)(1-\epsilon)\ln(\frac{2-\delta}{\delta})/(16\epsilon\mu_Z)$.

Proof. Let T denote the stopping time of any test of H_Z against $H_{Z'}$. From Lemma 7.6, and since $\mu_Z \le \frac{1}{4}$, $\mathbf{E}_{Z'}[Z] - E_Z[Z] > \epsilon\mu_Z/4$. Thus to test H_Z against H_Z', we can use the $\mathcal{BB}$ with input ϵ^* such that $\mu_Z(1+\epsilon^*) \le \mu_Z(1+\epsilon/4)(1-\epsilon^*) < \mu_{Z'}(1-\epsilon^*)$. Solving for ϵ^* we obtain $\epsilon^* \le \epsilon/(8+\epsilon)$. But Corollary 7.2 gives a lower bound on the expected number of experiments $\mathbf{E}[N_Z^*]$ run by $\mathcal{BB}$ with respect the Z. Next observe that, by Lemma 7.7, $-\omega_Z$ in Corollary 7.2 is at most $2\epsilon\mu_Z$. Substitution of $\epsilon = 8\epsilon^*/(1-\epsilon^*)$ completes the proof. ∎

Proof of the Lower Bound Theorem. The proof follows from Lemma 7.5 and Lemma 7.8. ∎

REFERENCES

[1] A. Broder, *How hard is it to marry at random? (On the approximation of the permanent)*, in Proceedings of the 18th IEEE Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 50–58; Erratum in Proceedings of the 20th IEEE Symposium on the Theory of Computing, 1988, p. 551.

[2] R. Canetti, G. Even, and O. Goldreich, *Lower Bounds for Sampling Algorithms for Estimating the Average*, Technical Report 789, Department of Computer Science, Technion, Tel Aviv, Israel, 1993.

[3] P. Dagum, D. Karp, M. Luby, and S. Ross, *An optimal algorithm for Monte-Carlo estimation (extended abstract)*, in Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, Milwaukee, WI, 1995, pp. 142–149.

[4] P. Dagum, M. Luby, M. Mihail, and U. Vazirani, *Polytopes, permanents and graphs with large factors*, in Proceedings of the 29th IEEE Symposium on Foundations of Computer Science, White Plains, NY, 1988, pp. 412–421.

[5] P. Dagum and M. Luby, *Approximating the permanent of graphs with large factors*, Theoret. Comput. Sci., Part A, 102 (1992), pp. 283–305.

[6] P. Dagum and M. Luby, *An optimal approximation algorithm for Bayesian inference*, Artificial Intelligence, 78, (1997), pp. 1–27.

[7] M. Dyer, A. Frieze, and R. Kannan, *A random polynomial time algorithm for approximating the volume of convex bodies*, in Proceedings of the 21st IEEE Symposium on the Theory of Computing, Seattle, WA, 1989, pp. 375–381.

[8] M. Dyer, A. Frieze, R. Kannan, A. Kapoor, L. Kerkovic, and U. Vazirani, *A mildly exponential time algorithm for approximating the number of solutions to a multidimensional knapsack problem*, Comb. Probab. Comput., 2 (1993), pp. 271–284.

[9] A. Frieze and M. Jerrum, *An analysis of a Monte Carlo algorithm for estimating the permanent*, Combinatorica, 15 (1995), pp. 67–83.

[10] M. Jerrum and A. Sinclair, *Conductance and the rapid mixing property for Markov chains: The approximation of the permanent resolved*, in Proceedings of the 20th IEEE Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 235–243.

[11] M. Jerrum and A. Sinclair, *Polynomial-time approximation algorithms for the Ising model*, SIAM J. Comput., 22 (1993), pp. 1087–1116.

[12] M. Jerrum, L. Valiant, and V. Vazirani, *Random generation of combinatorial structures from a uniform distribution*, Theoret. Comput. Sci., 43 (1986), pp. 169–188.

[13] M. Jerrum and U. Vazirani, *A mildly exponential approximation algorithm for the permanent*, in Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 320–326.

[14] N. Karmarkar, R. Karp, R. Lipton, L. Lovász, and M. Luby, *A Monte Carlo algorithm for estimating the permanent*, SIAM J. Comput., 22 (1993), pp. 284–293.

[15] R. Karp and M. Luby, *Monte Carlo algorithms for planar multiterminal network reliability problems*, J. Complexity, 2 (1985), pp. 45–64.

[16] R. Karp and M. Luby, *Monte Carlo algorithms for enumeration and reliability problems*, in Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 56–64.

[17] R. Karp, M. Luby, and N. Madras, *Monte Carlo approximation algorithms for enumeration problems*, J. Algorithms, 10 (1989), pp. 429–448.

[18] M. Karpinski and M. Luby, *Approximating the number of solutions to a* GF[2] *formula*, J. Algorithms, 14 (1993), pp. 280–287.

[19] M. Mihail and P. Winkler, *On the number of Eulerian orientations of a graph*, in Proceedings of the Third ACM/SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 138–145.

[20] D. Randall and A. Sinclair, *Testable algorithms for selfavoiding walks*, in Proceedings of the 5th ACM/SIAM Symposium on Discrete Algorithms, 1994, pp. 593–602.

[21] D. Siegmund, *Sequential Analysis*, Springer-Verlag, New York, 1985.

[22] A. Wald, *Sequential Analysis*, John Wiley, New York, 1947.

SIAM J. Comput.
Vol. 29, No. 5, pp. 1497–1520

SEPARATING COMPLEXITY CLASSES USING AUTOREDUCIBILITY[*]

HARRY BUHRMAN[†], LANCE FORTNOW[‡], DIETER VAN MELKEBEEK[§], AND LEEN TORENVLIET[¶]

Abstract. A set is autoreducible if it can be reduced to itself by a Turing machine that does not ask its own input to the oracle. We use autoreducibility to separate the polynomial-time hierarchy from exponential space by showing that all Turing complete sets for certain levels of the exponential-time hierarchy are autoreducible but there exists some Turing complete set for doubly exponential space that is not.

Although we already knew how to separate these classes using diagonalization, our proofs separate classes solely by showing they have different structural properties, thus applying Post's program to complexity theory. We feel such techniques may prove unknown separations in the future. In particular, if we could settle the question as to whether all Turing complete sets for doubly exponential time are autoreducible, we would separate either polynomial time from polynomial space, and nondeterministic logarithmic space from nondeterministic polynomial time, or else the polynomial-time hierarchy from exponential time.

We also look at the autoreducibility of complete sets under nonadaptive, bounded query, probabilistic, and nonuniform reductions. We show how settling some of these autoreducibility questions will also lead to new complexity class separations.

Key words. complexity classes, completeness, autoreducibility, coherence

AMS subject classifications. 68Q15, 68Q05, 03D15

PII. S0097539798334736

1. Introduction.

While complexity theorists have made great strides in understanding the structure of complexity classes, they have not yet found the proper tools to do nontrivial separation of complexity classes such as $\mathcal{P}$ and $\mathcal{NP}$. They have developed sophisticated diagonalization, combinatorial and algebraic techniques, but none of these ideas have yet proven very useful in the separation task.

Back in the early days of computability theory, Post [13] wanted to show that the set of noncomputable computably enumerable sets strictly contains the Turing complete computably enumerable sets. In what we now call "Post's program" (see

[*] Received by the editors February 27, 1998; accepted for publication (in revised form) April 20, 1999; published electronically March 15, 2000. A preliminary version of this paper was presented at the 35th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1995.

http://www.siam.org/journals/sicomp/29-5/33473.html

[†]CWI, Kruislaan 413, P. O. Box 94079, 1090 GB Amsterdam, The Netherlands (buhrman@cwi.nl). Part of this research was done while the author was visiting the Univ. Politècnica de Catalunya in Barcelona. The research of this author was partially supported by the Dutch foundation for scientific research (NWO) through NFI Project ALADDIN, under contract NF 62-376 and a TALENT stipend.

[‡]Department of Computer Science, University of Chicago, 1100 East 58th Street, Chicago, IL 60637 (fortnow@cs.uchicago.edu). The research of this author was supported in part by NSF grant CCR 92-53582. The research was partly conducted while the author was visiting CWI.

[§]DIMACS, Rutgers University, 96 Frelinghuysen Rd., Piscataway, NJ 08854-8018 (dieter@dimacs.rutgers.edu). The research of this author was supported in part by the NSF through grant CCR 92-53582, by the European Union through TMR grant ERB-4001-GT-96-0783 while the author was visiting CWI and the University of Amsterdam, and by the Fields Institute while the author was visiting the Fields Institute at the University of Toronto.

[¶]Faculteit WINS, Universiteit van Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands (leen@wins.uva.nl). The research of this author was partially supported by HC&M grant ERB4050PL93-0516.

[11, 15]), Post tried to show these classes differ by finding a property that holds for all sets in the first class but not for some set in the second.

We would like to resurrect Post's program for separating classes in complexity theory. In particular we will show how some classes differ by showing that their complete sets have different structures. While we do not separate any classes not already separable by known diagonalization techniques, we feel that refinements to our techniques may yield some new separation results.

In this paper we will concentrate on the property known as "autoreducibility." A set A is autoreducible if we can decide whether an input x belongs to A in polynomial-time by making queries about membership of strings different from x to A.

Trakhtenbrot [16] first looked at autoreducibility in both the unbounded and space-bounded models. Ladner [10] showed that there exist Turing complete computably enumerable sets that are not autoreducible. Ambos-Spies [1] first transferred the notion of autoreducibility to the polynomial-time setting. More recently, Yao [18] and Beigel and Feigenbaum [5] have studied a probabilistic variant of autoreducibility known as "coherence."

In this paper, we ask for what complexity classes do all the complete sets have the autoreducibility property. In particular we show the following.

(i) All Turing complete sets for $\Delta_k^{\mathcal{EXP}}$ are autoreducible for any constant k, where $\Delta_{k+1}^{\mathcal{EXP}}$ denotes the sets that are exponential-time Turing reducible to $\Sigma_k^{\mathcal{P}}$.

(ii) There exists a Turing complete set for doubly exponential space that is not autoreducible.

Since the union of all sets $\Delta_k^{\mathcal{EXP}}$ coincides with the exponential-time hierarchy, we obtain a separation of the exponential-time hierarchy from doubly exponential space and thus of the polynomial-time hierarchy from exponential space. Although these results also follow from the space hierarchy theorems [9] which we have known for a long time, our proof does not directly use diagonalization, but rather separates the classes by showing that they have different structural properties.

Issues of relativization do not apply to this work because of oracle access (see [8]): a polynomial-time autoreduction cannot view as much of the oracle as an exponential or doubly exponential computation. To illustrate this point we show that there exists an oracle relative to which some complete set for exponential time is not autoreducible.

Note that if we can settle whether the Turing complete sets for doubly exponential time are all autoreducible one way or the other, we will have a major separation result. If there exists a Turing complete set for doubly exponential time that is not autoreducible, then we get that the exponential-time hierarchy is strictly contained in doubly exponential time and hence that the polynomial-time hierarchy is strictly contained in exponential time. If all of the Turing complete sets for doubly exponential time are autoreducible, we get that doubly exponential time is strictly contained in doubly exponential space, and thus polynomial time strictly in polynomial space. We will also show that this assumption implies a separation of nondeterministic logarithmic space from nondeterministic polynomial time. Similar implications hold for space-bounded classes (see section 5). Autoreducibility questions about doubly exponential time and exponential space thus remain an exciting line of research.

We also study the nonadaptive variant of the problem. Our main results scale down one exponential as follows:

(i) All truth-table complete sets $\Delta_k^{\mathcal{P}}$ are truth-table autoreducible for any constant k, where $\Delta_{k+1}^{\mathcal{P}}$ denotes the sets polynomial-time Turing reducible to $\Sigma_k^{\mathcal{P}}$.

(ii) There exists a truth-table complete set for exponential space that is not truth-table autoreducible.

Again, finding out whether all truth-table complete sets for intermediate classes, namely polynomial space and exponential time, are truth-table autoreducible, would have major implications.

In contrast to the above results we exhibit the limitations of our approach: For the restricted reducibility where we are only allowed to ask two nonadaptive queries, all complete sets for $\mathcal{EXP}$, $\mathcal{EXPSPACE}$, $\mathcal{EEXP}$, $\mathcal{EEXPSPACE}$, etc., are autoreducible.

We also argue that uniformity is crucial for our technique of separating complexity classes, because our nonautoreducibility results fail in the nonuniform setting. Razborov and Rudich [14] show that if strong pseudorandom generators exist, "natural proofs" cannot separate certain nonuniform complexity classes. Since this paper relies on uniformity in an essential way, their result does not apply.

Regarding the probabilistic variant of autoreducibility mentioned above, we can strengthen our results and construct a Turing complete set for doubly exponential space that is not even probabilistically autoreducible. We leave the analog of this theorem in the nonadaptive setting open: Does there exist a truth-table complete set for exponential space that is not probabilistically truth-table autoreducible? We do show that every truth-table complete set for exponential time is probabilistically truth-table autoreducible. Thus, a positive answer to the open question would establish that exponential time is strictly contained in exponential space. A negative answer, on the other hand, would imply a separation of nondeterministic logarithmic space from nondeterministic polynomial time.

Here is the outline of the paper: First, we introduce our notation and state some preliminaries in section 2. Next, in section 3 we establish our negative autoreducibility results, for the adaptive as well as the nonadaptive case. Then we prove the positive results in section 4, where we also briefly look at the randomized and nonuniform settings. Section 5 discusses the separations that follow from our results and would follow from improvements on them. Finally, we conclude in section 6 and mention some possible directions for further research.

1.1. Errata to conference version.

A previous version of this paper [6] erroneously claimed proofs showing all Turing complete sets for $\mathcal{EXPSPACE}$ are autoreducible and all truth-table complete sets for $\mathcal{PSPACE}$ are nonadaptively autoreducible. Combined with the additional results in this version, we would have a separation of $\mathcal{NL}$ and $\mathcal{NP}$ (see section 5).

However the proofs in the earlier version failed to account for the growth of the running time when recursively computing previous players' moves. We use the proof technique in section 3 though unfortunately we get weaker theorems. The original results claimed in the previous version remain important open questions as resolving them either way will yield new separation results.

2. Notation and preliminaries.

Most of our complexity theoretic notation is standard. We refer the reader to the textbooks by Balcázar, Díaz, and Gabarró [4, 3], and by Papadimitriou [12].

We use the binary alphabet $\Sigma = \{0, 1\}$. We denote the difference of a set A with a set B, i.e., the subset of elements of A that do not belong to B, by $A \setminus B$.

For any integer $k \geq 0$, a Σ_k-formula is a Boolean expression of the form

$$\exists\, y_1 \in \Sigma^{n_1}, \forall\, y_2 \in \Sigma^{n_2}, \ldots, Q_k\, y_k \in \Sigma^{n_k} : \phi(y_1, y_2, \ldots, y_k, z), \tag{2.1}$$

where ϕ is a Boolean formula, Q_i denotes $\exists$ if i is odd, and $\forall$ otherwise, and the n_i's are positive integers. We say that (2.1) has $k-1$ alternations. A Π_k-formula is just like (2.1) except that it starts with a $\forall$-quantifier. It also has $k-1$ alternations. A QBF$_k$-formula is a Σ_k-formula (2.1) or a Π_k-formula without free variables z.

For any integer $k \geq 0$, $\Sigma_k^{\mathcal{P}}$ denotes the kth Σ-level of the polynomial-time hierarchy. We define these levels recursively by $\Sigma_0^{\mathcal{P}} = \mathcal{P}$ and $\Sigma_{k+1}^{\mathcal{P}} = \mathcal{NP}^{\Sigma_k^{\mathcal{P}}}$. The Δ-levels of the polynomial-time and exponential-time hierarchy are defined as $\Delta_{k+1}^{\mathcal{P}} = \mathcal{P}^{\Sigma_k^{\mathcal{P}}}$ and $\Delta_{k+1}^{\mathcal{EXP}} = \mathcal{EXP}^{\Sigma_k^{\mathcal{P}}}$, respectively. The polynomial-time hierarchy $\mathcal{PH}$ equals the union of all sets $\Delta_k^{\mathcal{P}}$, and the exponential-time hierarchy $\mathcal{EXPH}$ similarly equals the union of all sets $\Delta_k^{\mathcal{EXP}}$.

A *reduction* of a set A to a set B is a polynomial-time oracle Turing machine M such that $M^B = A$. We say that A reduces to B and write $A \leq_{\mathrm{T}}^{\mathcal{P}} B$ ("T" for Turing). The reduction M is *nonadaptive* if the oracle queries M makes on any input are independent of the oracle, i.e., the queries do not depend upon the answers to previous queries. In that case we write $A \leq_{\mathrm{tt}}^{\mathcal{P}} B$ ("tt" for truth-table). Reductions of functions to sets are defined similarly. If the number of queries on an input of length n is bounded by $q(n)$, we write $A \leq_{q(n)-\mathrm{T}}^{\mathcal{P}} B$ and $A \leq_{q(n)-\mathrm{tt}}^{\mathcal{P}} B$, respectively; if it is bounded by some constant, we write $A \leq_{\mathrm{btt}}^{\mathcal{P}} B$ ("b" for bounded). We denote the set of queries of M on input x with oracle B by $Q_{M^B}(x)$; in case of nonadaptive reductions, we omit the oracle B in the notation. If the reduction asks only one query and answers the answer to that query, we write $A \leq_{\mathrm{m}}^{\mathcal{P}} B$ ("m" for many-one).

For any reducibility $\leq_r^{\mathcal{P}}$ and any complexity class $\mathcal{C}$, a set C is $\leq_r^{\mathcal{P}}$*-hard* for $\mathcal{C}$ if we can $\leq_r^{\mathcal{P}}$-reduce every set $A \in \mathcal{C}$ to C. If in addition $C \in \mathcal{C}$, we call C $\leq_r^{\mathcal{P}}$*-complete* for $\mathcal{C}$. For any integer $k \geq 0$, the set TQBF$_k$ of all true QBF$_k$-formulae is $\leq_{\mathrm{m}}^{\mathcal{P}}$-complete for $\Sigma_k^{\mathcal{P}}$. For $k = 1$, this reduces to the fact that the set SAT of satisfiable Boolean formulae is $\leq_{\mathrm{m}}^{\mathcal{P}}$-complete for $\mathcal{NP}$.

Now we get to the key concept of this paper in the following definition.

DEFINITION 2.1. *A set A is* autoreducible *if there is a reduction M of A to itself that never queries its own input, i.e., for any input x and any oracle B, $x \notin Q_{M^B}(x)$. We call such M an* autoreduction *of A.*

We will also discuss randomized and nonuniform variants. A set is *probabilistically autoreducible* if it has a probabilistic autoreduction with bounded two-sided error. Yao [18] first studied this concept under the name "coherence." A set is *nonuniformly autoreducible* if it has an autoreduction that uses polynomial advice. For all these notions, we can consider both the adaptive and the nonadaptive case. For randomized autoreducibility, nonadaptiveness means that the queries only depend on the input and the random seed.

3. Nonautoreducibility results.

In this section, we show that large complexity classes have complete sets that are not autoreducible.

THEOREM 3.1. *There is a $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-complete set for $\mathcal{EEXPSPACE}$ that is not autoreducible.*

Most natural classes containing $\mathcal{EEXPSPACE}$, e.g., triply exponential time and triply exponential space, also have this property.

We can even construct the complete set in Theorem 3.1 to defeat every probabilistic autoreduction.

THEOREM 3.2. *There is a $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-complete set for $\mathcal{EEXPSPACE}$ that is not probabilistically autoreducible.*

In the nonadaptive setting, we obtain the following theorem.

THEOREM 3.3. *There is a $\leq_{3-\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{EXPSPACE}$ that is not nonadaptively autoreducible.*

Unlike the case of Theorem 3.1, our construction does not seem to yield a truth-table complete set that is not probabilistically nonadaptively autoreducible. In fact, as we shall show in section 4.3, such a result would separate $\mathcal{EXP}$ from $\mathcal{EXPSPACE}$; see also section 5.

We will detail in section 4.3 that our nonautoreducibility results do not hold in the nonuniform setting.

3.1. Adaptive autoreductions. Suppose we want to construct a nonautoreducible Turing complete set for a complexity class $\mathcal{C}$, i.e., a set A such that

1. A is not autoreducible,
2. A is Turing hard for $\mathcal{C}$,
3. A belongs to $\mathcal{C}$.

If $\mathcal{C}$ has a $\leq_{\mathrm{m}}^{\mathcal{P}}$-complete set K, realizing goals 1 and 2 is not too hard: We can encode K in A, and at the same time diagonalize against all autoreductions. A straightforward implementation would be to encode $K(y)$ as $A(\langle 0, y\rangle)$, and stagewise diagonalize against all $\leq_{\mathrm{T}}^{\mathcal{P}}$-reductions M by picking for each M an input x not of the form $\langle 0, y\rangle$ that is not queried during previous stages, and setting $A(x) = 1 - M^A(x)$. However, this construction does not seem to achieve goal 3. In particular, deciding the membership of a diagonalization string x to A might require computing $A(\langle 0, y\rangle) = K(y)$ on inputs y of length $|x|^c$, assuming M runs in time n^c. Since we have to do this for all potential autoreductions M, we can only bound the resources (time, space) needed to decide A by a function in $t(n^{\omega(1)})$, where $t(n)$ denotes the amount of resources some deterministic Turing machine accepting K uses. That does not suffice to keep A inside $\mathcal{C}$.

To remedy this problem, we will avoid the need to compute $K(y)$ on large inputs y, say of length at least $|x|$. Instead, we will make sure we can encode the membership of such strings to *any* set, not just K, and at the same time diagonalize against M on input x. We will argue that we can do this by considering two possible coding regions at every stage as opposed to a fixed one: the left region L, containing strings of the form $\langle 0, y\rangle$, and the right region R, similarly containing strings of the form $\langle 1, y\rangle$. The following states that we can use one of the regions to encode an arbitrary sequence, and set the other region such that the output of M on input x is fixed and indicates the region used for encoding.

STATEMENT 3.4. *Either it is the case that for any setting of L there is a setting of R such that $M^A(x)$ accepts, or for any setting of R there is a setting of L such that $M^A(x)$ rejects.*

This allows us to achieve goals 1 and 2 from above as follows. In the former case, we will set $A(x) = 0$ and encode K in L (at that stage); otherwise we will set $A(x) = 1$ and encode K in R. Since the value of $A(x)$ does not affect the behavior of M^A on input x, we diagonalize against M in both cases. Also, in any case,

$$K(y) = A(\langle A(x), y\rangle),$$

so deciding K is still easy when given A. Moreover—and crucially—in order to compute $A(x)$, we no longer have to decide $K(y)$ on large inputs y, of length $|x|$ or more. Instead, we have to check whether the former case in Statement 3.4 holds or not. Although quite complex a task, it only depends on M and on the part of A constructed so far, not on the value of $K(y)$ for any input of length $|x|$ or more: We verify whether we can encode *any* sequence, not just the characteristic sequence of K for lengths at least $|x|$, and at the same time diagonalize against M on input x.

Provided the complexity class $\mathcal{C}$ is sufficiently powerful, we can perform this task in $\mathcal{C}$.

There is still a catch, though. Suppose we have found out that the former case in Statement 3.4 holds. Then we will use the left region L to encode K (at that stage), and we know we can diagonalize against M on input x by setting the bits of the right region R appropriately. However, deciding exactly how to set these bits of the noncoding region requires, in addition to determining which region we should use for coding, the knowledge of $K(y)$ for all y such that $|x| \leq |y| \leq |x|^c$. In order to also circumvent the need to decide K for too large inputs here, we will use a slightly stronger version of Statement 3.4 obtained by grouping quantifiers into blocks and rearranging them. We will partition the coding and noncoding regions into intervals. We will make sure that for any given interval, the length of a string in that interval (or any of the previous intervals) is no more than the square of the length of any string in that interval. Then we will blockwise alternately set the bits in the coding region according to K, and the corresponding ones in the noncoding region so as to maintain the diagonalization against M on input x as in Statement 3.4. This way, in order to compute the bit $A(\langle 1, z\rangle)$ of the noncoding region, we will only have to query K on inputs y with $|y| \leq |z|^2$, as opposed to $|y| \leq |z|^c$ for an arbitrarily large c depending on M as was the case before.

This is what happens in the next lemma, which we prove in a more general form, because we will need the generalization later on in section 5.

LEMMA 3.5. *Fix a set K, and suppose we can decide it simultaneously in time $t(n)$ and space $s(n)$. Let $\alpha : \mathbb{N} \to (0, \infty)$ be a constructible monotone unbounded function, and suppose there is a deterministic Turing machine accepting TQBF that takes time $t'(n)$ and space $s'(n)$ on QBF-formulae of size $2^{n^{\alpha(n)}}$ with at most* $\log \alpha(n)$ *alternations. Then there is a set A such that*

1. *A is not autoreducible;*
2. *$K \leq^{\mathcal{P}}_{2-\mathrm{T}} A$;*
3. *We can decide A simultaneously in time $O(2^{n^2} \cdot t(n^2) + 2^n \cdot t'(n))$ and space $O(2^{n^2} + s(n^2) + s'(n))$.*

Proof. Fix a function α satisfying the hypotheses of the lemma, and let $\beta = \sqrt{\alpha}$. Let $M_1, M_2, \ldots$ be a standard enumeration of autoreductions clocked such that M_i runs in time $n^{\beta(i)}$ on inputs of length n. Our construction starts out with A being the empty set, and then adds strings to A in subsequent stages $i = 1, 2, 3, \ldots$ defined by the following sequence:

$$\begin{cases} n_0 &= 0, \\ n_{i+1} &= n_i^{\beta(n_i)} + 1. \end{cases}$$

Note that since M_i runs in time $n^{\beta(i)}$, M_i cannot query strings of length n_{i+1} or more on input 0^{n_i}.

Fix an integer $i \geq 1$ and let $m = n_i$. For any integer j such that $0 \leq j \leq \log \beta(m)$, let I_j denote the set of all strings with lengths in the interval $[m^{2^j}, \min(m^{2^{j+1}}, m^{\beta(m)} + 1))$. Note that $\{I_j\}_{j=0}^{\log \beta(m)}$ forms a partition of the set I of strings with lengths in $[m, m^{\beta(m)} + 1) = [n_i, n_{i+1})$ with the property that for any $0 \leq k \leq \log \beta(m)$, the length of any string in $\cup_{j=0}^{k} I_j$ is no more than the square of the length of any string in I_k.

During the ith stage of the construction, we will encode the restriction $K|_I$ of K to I into $\{\langle b, y\rangle \mid b \in \{0, 1\}$ and $y \in I\}$, and use the string 0^m for diagonalizing against

```
if formula (3.1) holds
  then for $j = 0, \dots, \log\beta(m)$
        $(\ell_y)_{y\in I_j} \leftarrow (K(y))_{y\in I_j}$
        $(r_y)_{y\in I_j} \leftarrow$ the lexicographically first value satisfying
              $(\forall\,\ell_y)_{y\in I_j+1}, (\exists\,r_y)_{y\in I_{j+1}}, (\forall\,\ell_y)_{y\in I_{j+2}}, (\exists\,r_y)_{y\in I_{j+2}},$
                $\dots, (\forall\,\ell_y)_{y\in I_{\log\beta(m)}}, (\exists\,r_y)_{y\in I_{\log\beta(m)}} : M_i^{A'}(0^m)$ accepts,
           where $A' = A \cup \{\langle 0, y\rangle \mid y \in I \text{ and } \ell_y = 1\} \cup \{\langle 1, y\rangle \mid y \in I \text{ and } r_y = 1\}$
        end for
      $A \leftarrow A \cup \{\langle 0, y\rangle \mid y \in I \text{ and } \ell_y = 1\} \cup \{\langle 1, y\rangle \mid y \in I \text{ and } r_y = 1\}$
  else { formula (3.2) holds }
      for $j = 0, \dots, \log\beta(m)$
        $(r_y)_{y\in I_j} \leftarrow (K(y))_{y\in I_j}$
        $(\ell_y)_{y\in I_j} \leftarrow$ the lexicographically first value satisfying
              $(\forall\,r_y)_{y\in I_{j+1}}, (\exists\,\ell_y)_{y\in I_{j+1}}, (\forall\,r_y)_{y\in I_{j+2}}, (\exists\,\ell_y)_{y\in I_{j+2}},$
                $\dots, (\forall\,r_y)_{y\in I_{\log\beta(m)}}, (\exists\,\ell_y)_{y\in I_{\log\beta(m)}} : M_i^{A'}(0^m)$ accepts,
           where $A' = A \cup \{\langle 0, y\rangle \mid y \in I \text{ and } \ell_y = 1\} \cup \{\langle 1, y\rangle \mid y \in I \text{ and } r_y = 1\}$
        end for
      $A \leftarrow A \cup \{0^m\} \cup \{\langle 0, y\rangle \mid y \in I \text{ and } \ell_y = 1\} \cup \{\langle 1, y\rangle \mid y \in I \text{ and } r_y = 1\}$
  end if
```

FIG. 3.1. *Stage i of the construction of the set A in Lemma* 3.5.

M_i, applying the next strengthening of Statement 3.4 to do so.

CLAIM 3.6. *For any set A, at least one of the following holds:*

$$(\forall\,\ell_y)_{y\in I_0}, (\exists\,r_y)_{y\in I_0}, (\forall\,\ell_y)_{y\in I_1}, (\exists\,r_y)_{y\in I_1}, \dots, (\forall\,\ell_y)_{y\in I_{\log\beta(m)}}, (\exists\,r_y)_{y\in I_{\log\beta(m)}} : M_i^{A'}(0^m) \text{ accepts} \tag{3.1}$$

or

$$(\forall\,r_y)_{y\in I_0}, (\exists\,\ell_y)_{y\in I_0}, (\forall\,r_y)_{y\in I_1}, (\exists\,\ell_y)_{y\in I_1}, \dots, (\forall\,r_y)_{y\in I_{\log\beta(m)}}, (\exists\,\ell_y)_{y\in I_{\log\beta(m)}} : M_i^{A'}(0^m) \text{ rejects}, \tag{3.2}$$

where A' denotes $A \cup \{\langle 0, y\rangle \mid y \in I$ and $\ell_y = 1\} \cup \{\langle 1, y\rangle \mid y \in I$ and $r_y = 1\}$.

Here we use $(Q\,z_y)_{y\in Y}$ as a shorthand for $Q\,z_{y_1}, Q\,z_{y_2}, \dots, Q\,z_{y_{|Y|}}$, where $Y = \{y_1, y_2, \dots, y_{|Y|}\}$ and all variables are quantified over $\{0, 1\}$. Without loss of generality we assume that the range of the pairing function $\langle\cdot,\cdot\rangle$ is disjoint from 0^*.

Proof of Claim 3.6. Fix A. If (3.1) does not hold, then its negation holds, i.e,

$$(\exists\,\ell_y)_{y\in I_0}, (\forall\,r_y)_{y\in I_0}, (\exists\,\ell_y)_{y\in I_1}, (\forall\,r_y)_{y\in I_1}, \dots, (\exists\,\ell_y)_{y\in I_{\log\beta(m)}}, (\forall\,r_y)_{y\in I_{\log\beta(m)}} : M_i^{A'}(0^m) \text{ rejects}. \tag{3.3}$$

Switching the quantifiers $(\exists\,\ell_y)_{y\in I_j}$ and $(\forall\,r_y)_{y\in I_j}$ pairwise for every $0 \le j \le \log\beta(m)$ in (3.3) yields the weaker statement (3.2). ☐

Figure 3.1 describes the ith stage in the construction of the set A. Note that the lexicographically first values in this algorithm always exist, so the construction works fine. We now argue that the resulting set A satisfies the properties of Lemma 3.5:

1. The construction guarantees that $A(0^m) = 1 - M_i^{A\setminus\{0^m\}}(0^m)$ holds by the end of stage i. Since M_i on input 0^m cannot query 0^m (because M_i is an autoreduction)

nor any of the strings added during subsequent stages (because M_i does not even have the time to write down any of these strings), $A(0^m) = 1 - M_i^A(0^m)$ holds for the final set A. Thus, M_i is not an autoreduction of A. Since this is true of any autoreduction M_i, the set A is not autoreducible.

2. During stage i, we encode $K|_I$ in the left region iff we do not put 0^m into A; otherwise we encode $K|_I$ in the right region. Thus, for any $y \in I$, $K(y) = A(\langle A(0^m), y\rangle)$. Therefore, $K \leq_{2-\mathrm{T}}^{\mathcal{P}} A$.

3. First note that A only contains strings of the form 0^m with $m = n_i$ for some integer $i \geq 1$, and strings of the form $\langle b, y\rangle$ with $b \in \{0,1\}$ and $y \in \Sigma^*$. Assume we have executed the construction of A up to but not including stage i and stored the result in memory. The additional work to decide the membership to A of a string belonging to the ith stage is as follows.

(i) *Case* 0^m. Since $0^m \in A$ iff formula (3.1) does not hold and (3.1) is a $\mathrm{QBF}_{2\log\beta(m)}$-formula of size $2^{O(m^{\beta(m)})} \leq 2^{m^{\alpha(m)}}$, we can decide whether $0^m \in A$ in time $O(t'(m))$ and space $O(s'(m))$.

(ii) *Case* $\langle b, z\rangle$ *where* $b = A(0^m)$ *and* $z \in I$. Then $\langle b, z\rangle \in A$ iff $z \in K$, which we can decide in time $t(|z|)$ and space $s(|z|)$.

(iii) *Case* $\langle b, z\rangle$ *where* $b = 1 - A(0^m)$ *and* $z \in I$. Say $z \in I_k$, $0 \leq k \leq \log\beta(m)$. In order to compute whether $\langle b, z\rangle \in A$, we run the part of stage i corresponding to the values of j in Figure 3.1 up to and including k, and store the results in memory. This involves computing K on $\cup_{j=0}^{k} I_j$ and deciding $O(2^{|z|})$ $\mathrm{QBF}_{2\log\beta(m)}$-formulae of size $2^{O(m^{\beta(m)})} \leq 2^{m^{\alpha(m)}}$, namely one formula for each $y \in \cup_{j=0}^{k} I_j$ which precedes or equals z in lexicographic order. The latter takes $O(2^{|z|} \cdot t'(m))$ time and $O(2^{|z|} + s'(m))$ space. Since every string in $\cup_{j=0}^{k} I_j$ is of size no more than $|z|^2$, we can do the former in time $O(2^{|z|^2} \cdot t(|z|^2))$ and space $O(2^{|z|^2} + s(|z|^2))$. So, the requirements for this stage are $O(2^{|z|^2} \cdot t(|z|^2) + 2^{|z|} \cdot t'(|z|))$ time and $O(2^{|z|^2} + s(|z|^2) + s'(|z|))$ space.

A similar analysis also shows that we can perform the stages up to but not including i in time $O(2^m \cdot (t(m) + t'(m)))$ and space $O(2^m + s(m) + s'(m))$. All together, this yields the time and space bounds claimed for A. $\square$

Using the upper bound $2^{n^{\alpha(n)}}$ for $s'(n)$, the smallest standard complexity class to which Lemma 3.5 applies, seems to be $\mathcal{EEXPSPACE}$. This results in Theorem 3.1.

Proof of Theorem 3.1. In Lemma 3.5, set K a $\leq_{\mathrm{m}}^{\mathcal{P}}$-complete set for $\mathcal{EEXPSPACE}$, and $\alpha(n) = n$. $\square$

In section 4.2, we will see that $\leq_{2-\mathrm{T}}^{\mathcal{P}}$ in the statement of Theorem 3.1 is optimal: Theorem 4.6 shows that Theorem 3.1 fails for $\leq_{2-\mathrm{tt}}^{\mathcal{P}}$.

We note that the proof of Theorem 3.1 carries through for $\leq_{\mathrm{T}}^{\mathcal{EXPSPACE}}$-reductions with polynomially bounded query lengths. This implies the strengthening given by Theorem 3.2.

3.2. Nonadaptive autoreductions. Diagonalizing against nonadaptive autoreductions M is easier. If M runs in time $\tau(n)$, there can be no more than $\tau(n)$ coding strings that interfere with the diagonalization, as opposed to $2^{\tau(n)}$ in the adaptive case. This allows us to reduce the complexity of the set constructed in Lemma 3.5 as follows.

LEMMA 3.7. *Fix a set K, and suppose we can decide it simultaneously in time $t(n)$ and space $s(n)$. Let $\alpha : \mathbb{N} \to (0, \infty)$ be a constructible monotone unbounded function, and suppose there is a deterministic Turing machine accepting TQBF that takes time $t'(n)$ and space $s'(n)$ on QBF-formulae of size $n^{\alpha(n)}$ with at most* $\log\alpha(n)$ *alternations. Then there is a set A such that*

1. *A is not nonadaptively autoreducible;*
2. *$K \leq^{\mathcal{P}}_{3-\mathrm{tt}} A$;*
3. *We can decide A simultaneously in time $O(2^n + n^{\alpha(n)}) \cdot (t(n^2) + t'(n)))$ and space $O(2^n + n^{\alpha(n)} + s(n^2) + s'(n))$.*

Proof. The construction of the set A is the same as in Lemma 3.5 (see Figure 3.1) apart from the following differences.

(i) $M_1, M_2, \ldots$ now is a standard enumeration of nonadaptive autoreductions clocked such that M_i runs in time $n^{\beta(i)}$ on inputs of length n. Note that the set $Q_M(x)$ of possible queries M makes on input x contains no more than $|x|^{\beta(i)}$ elements.

(ii) During stage $i \geq 1$ of the construction, I denotes the set of all strings y with lengths in $[m, m^{\beta(m)} + 1) = [n_i, n_{i+1})$ such that $\langle 0, y\rangle \in Q_{M_i}(0^m)$ or $\langle 1, y\rangle \in Q_{M_i}(0^m)$, and I_j for $0 \leq j \leq \log \beta(m)$ denotes the set of strings in I with lengths in $[m^{2^j}, \min(m^{2^{j+1}}, m^{\beta(m)} + 1))$. Note that the only ℓ_y's and r_y's that affect the validity of the predicate "$M_i^{A'}(0^m)$ accepts" in formula (3.1) and the corresponding formulae in Figure 3.1, are those for which $y \in I$.

(iii) At the end of stage i in Figure 3.1, we add the following line:

$$A \leftarrow A \cup \{\langle b, y\rangle \mid b \in \{0,1\}, y \in \Sigma^* \text{ with } m \leq |y| < m^{\beta(m)} + 1, y \notin I \text{ and } K(y) = 1\}.$$

This ensures coding $K(y)$ for strings y with lengths in $[n_i, n_{i+1})$ such that neither $\langle 0, y\rangle$ nor $\langle 1, y\rangle$ are queried by M_i on input 0^m. Although not essential, we choose to encode them in both the left and the right region.

The proof that A satisfies the three properties claimed carries over. Only the time and space analysis in the third point needs modification. The crucial simplification over the adaptive case lies in the fact that (3.1) and the similar formulae in Figure 3.1 now become $\mathrm{QBF}_{2\log\beta(n)}$-formulae of size $n^{O(\beta(m))}$ as opposed to of size $2^{O(m^{\beta(m)})}$ in Lemma 3.5. More specifically, referring to the proof of Lemma 3.5, we have the following cases regarding the work at stage i of the construction.

(i) *Case 0^m.* The above mentioned simplification takes care of this case.

(ii) *Case $\langle b, z\rangle$, where $b = A(0^m)$ and $z \in I$.* The argument of Lemma 3.5 carries over as such.

(iii) *Case $\langle b, z\rangle$, where $b = 1 - A(0^m)$ and $z \in I$.* Computing K on $\cup_{j=0}^{k} I_j$ and storing the result can be done in time $O(m^{\beta(m)} \cdot t(|z|^2))$ and space $O(m^{\beta(m)} + s(|z|^2))$. Deciding the $O(m^{\beta(m)})$ $\mathrm{QBF}_{2\log\beta(m)}$-formulae of size $m^{O(\beta(m))} \leq m^{\alpha(m)}$ involved requires no more than $O(m^{\beta(m)} \cdot t'(m))$ time and $O(m^{\beta(m)} + s'(m))$ space.

(iv) *Case $\langle b, z\rangle$ where $b \in \{0,1\}$, $m \leq |z| \leq m^{\beta(m)} + 1$, and $z \notin I$.* This is an additional case. By construction, $\langle b, z\rangle \in A$ iff $z \in K$, which we can decide in time $t(|z|)$ and space $s(|z|)$.

By similar analysis, a rough estimate of the resources required for the previous stages of the construction is $O(2^m \cdot (t(m) + t'(m)))$ time and $O(2^m + s(m) + s'(m))$ space, resulting in a total as stated in the lemma. ∎

As a consequence, we can lower the space complexity in the equivalent of Theorem 3.1 from doubly exponential to singly exponential, yielding Theorem 3.3. In section 4.2 we will show we cannot reduce the number of queries from three to two in Theorem 3.3.

If we restrict the number of queries the nonadaptive autoreduction is allowed to make to some fixed polynomial, the proof technique of Theorem 3.3 also applies to $\mathcal{EXP}$. In particular, we obtain the following theorem.

THEOREM 3.8. *There is a $\leq^{\mathcal{P}}_{3-\mathrm{tt}}$-complete set for $\mathcal{EXP}$ that is not $\leq^{\mathcal{P}}_{\mathrm{btt}}$-autoreducible.*

4. Autoreducibility results.

For small complexity classes, all complete sets turn out to be autoreducible. Beigel and Feigenbaum [5] established this property of all levels of the polynomial-time hierarchy as well as of $\mathcal{PSPACE}$, the largest class for which it was known to hold before our work. In this section, we will prove it for the Δ-levels of the exponential-time hierarchy.

As to nonadaptive reductions, the question was even open for all levels of the polynomial-time hierarchy. We will show here that the $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for the Δ-levels of the polynomial-time hierarchy are nonadaptively autoreducible. For any complexity class containing $\mathcal{EXP}$, we will prove that the $\leq_{2-\mathrm{tt}}^{\mathcal{P}}$-complete sets are $\leq_{2-\mathrm{tt}}^{\mathcal{P}}$-autoreducible.

Finally, we will also consider nonuniform and randomized autoreductions.

Throughout this section, we will assume without loss of generality an encoding γ of a computation of a given oracle Turing machine M on a given input x with the following properties. γ will be a marked concatenation of successive instantaneous descriptions of M, starting with the initial instantaneous description of M on input x, such that:

(i) Given a pointer to a bit in γ, we can find out whether that bit represents the answer to an oracle query by probing a constant number of bits of γ.

(ii) If it is the answer to an oracle query, the corresponding query is a substring of the prefix of γ up to that point, and we can easily compute a pointer to the beginning of that substring without probing γ any further.

(iii) If it is not the answer to an oracle query, we can perform a local consistency check for that bit which only depends on a constant number of previous bit positions of γ and the input x. Formally, there exist a function g_M and a predicate e_M, both polynomial-time computable, and a constant c_M such that the following holds: For any input x, any index i to a bit position in γ, and any j, $1 \leq j \leq c_M$, $g_M(x; i, j)$ is an index no larger than i, and

$$e_M(x; i, \gamma_{g_M(x;i,1)}, \gamma_{g_M(x;i,2)}, \ldots, \gamma_{g_M(x;i;c_M)}) \tag{4.1}$$

indicates whether γ passes the local consistency test for its ith bit γ_i. Provided the prefix of γ up to but not including position i is correct, the local consistency test is passed iff γ_i is correct.

We call such an encoding a *valid computation* of M on input x iff the local consistency tests (4.1) for all the bit positions i that do not correspond to oracle answers, are passed, and the other bits equal the oracle's answer to the corresponding query. Any other string we will call a *computation.*

4.1. Adaptive autoreductions.

We will first show that every $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set for $\mathcal{EXP}$ is autoreducible, and then generalize to all Δ-levels of the exponential-time hierarchy.

THEOREM 4.1. *Every $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set for $\mathcal{EXP}$ is autoreducible.*

Here is the proof idea: For any of the standard deterministic complexity classes $\mathcal{C}$, we can decide each bit of the computation on a given input x within $\mathcal{C}$. Thus, if A is a $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set for $\mathcal{C}$ that can be decided by a machine M within the confines of the class $\mathcal{C}$, then we can $\leq_{\mathrm{T}}^{\mathcal{P}}$-reduce deciding the ith bit of the computation of M on input x to A. Now, consider the two (possibly invalid) computations we obtain by applying the above reduction to every bit position, answering all queries except for x according to A, assuming $x \in A$ for one computation and $x \notin A$ for the other.

Note that the computation corresponding to the right assumption about $A(x)$ is certainly correct. Thus, if both computations yield the same answer (which we

```
if $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p(|x|)}\rangle) = R_\mu^{A\cup\{x\}}(\langle x, 2^{p(|x|)}\rangle)$
  then accept iff $R_\mu^{A\cup\{x\}}(\langle x, 2^{p(|x|)}\rangle) = 1$
  else $i \leftarrow R_\sigma^{A\cup\{x\}}(x)$
      accept iff $e_M(x; i, R_\mu^{A\setminus\{x\}}(\langle x, g_M(x;i;1)\rangle), R_\mu^{A\setminus\{x\}}(\langle x, g_M(x;i,2)\rangle), \ldots,$
                                  $\ldots, R_\mu^{A\setminus\{x\}}(\langle x, g_M(x;i,c_M)\rangle)) = 0$
  end if
```

FIG. 4.1. *Autoreduction for the set A of Theorem* 4.1 *on input x.*

can efficiently check using A without querying x), that answer is correct. If not, the other computation contains a mistake. We cannot check both computations entirely to see which one is right, but given a pointer to the first incorrect bit of the wrong computation, we can efficiently verify that it is mistaken by checking only a constant number of bits of that computation. The pointer is again computable within $\mathcal{C}$.

In case $\mathcal{C} \subseteq \mathcal{EXP}$, using a $\leq_{\mathrm{T}}^{\mathcal{P}}$-reduction to A and assuming $x \in A$ or $x \notin A$ as above, we can determine the pointer with oracle A (but without querying x) in polynomial time, since the pointer's length is polynomially bounded.

We now fill out the details.

Proof of Theorem 4.1. Fix a $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set A for $\mathcal{EXP}$. Say A is accepted by a Turing machine M such that the computation of M on an input of size n has length $2^{p(n)}$ for some fixed polynomial p. Without loss of generality the last bit of the computation gives the final answer. Let g_M, e_M, and c_M be the formalization of the local consistency test for M as described by (4.1).

Let $\mu(\langle x, i\rangle)$ denote the ith bit of the computation of M on input x. We can compute μ in $\mathcal{EXP}$, so there is an oracle Turing machine R_μ $\leq_{\mathrm{T}}^{\mathcal{P}}$-reducing μ to A.

Let $\sigma(x)$ be the first i, $1 \leq i \leq 2^{p(|x|)}$, such that $R_\mu^{A\setminus\{x\}}(\langle x, i\rangle) \neq R_\mu^{A\cup\{x\}}(\langle x, i\rangle)$, provided it exists. Again, we can compute σ in $\mathcal{EXP}$, so there is a $\leq_{\mathrm{T}}^{\mathcal{P}}$-reduction R_σ from σ to A.

Consider the algorithm in Figure 4.1 for deciding A on input x. The algorithm is a polynomial-time oracle Turing machine with oracle A that does not query its own input x. We now argue that it correctly decides A on input x. We distinguish between two cases.

(i) *Case* $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p(|x|)}\rangle) = R_\mu^{A\cup\{x\}}(\langle x, 2^{p(|x|)}\rangle)$. Since at least one of the computations $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ or $R_\mu^{A\cup\{x\}}(\langle x, \cdot\rangle)$ coincides with the actual computation of M on input x, and the last bit of the computation equals the final decision, correctness follows.

(ii) *Case* $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p(|x|)}\rangle) \neq R_\mu^{A\cup\{x\}}(\langle x, 2^{p(|x|)}\rangle)$. If $x \in A$, $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p(|x|)}\rangle) = 0$, so $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ contains a mistake. Variable i gets the correct value of the index of the first incorrect bit in this computation, so the local consistency test for $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ being the computation of M on input x fails on the ith bit, and we accept x. If $x \notin A$, then $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is a valid computation, so no local consistency test fails, and we reject x. ▯

The local checkability property of computations used in the proof of Theorem 4.1 does not relativize, because the oracle computation steps depend on the entire query, i.e., on a number of bits that is only limited by the resource bounds of the base machine, in this case exponentially many. We next show that Theorem 4.1 itself also does not relativize.

THEOREM 4.2. *Relative to some oracle, $\mathcal{EXP}$ has a $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-complete set that is* *not autoreducible.*

Proof. Note that $\mathcal{EXP}$ has the following property.

PROPERTY 4.3. *There is an oracle Turing machine N running in $\mathcal{EXP}$ such that for any oracle B, the set accepted by N^B is $\leq_{\mathrm{m}}^{\mathcal{P}}$-complete for $\mathcal{EXP}^B$.*

Without loss of generality, we assume that N runs in time 2^n. Let K^B denote the set accepted by N^B.

We will construct an oracle B and a set A such that A is $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-complete for $\mathcal{EXP}^B$ and is not $\leq_{\mathrm{T}}^{\mathcal{P}^B}$-autoreducible.

The construction of A is the same as in Lemma 3.5 (see Figure 3.1) with $\beta(n) = \log n$ and $K = K^B$, except that the reductions M_i now also have access to the oracle B.

We will encode in B information about the construction of A that reduces the complexity of A relative to B but do it high enough so as not to destroy the $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-completeness of A for $\mathcal{EXP}^B$ nor the diagonalizations against $\leq_{\mathrm{T}}^{\mathcal{P}^B}$-autoreductions.

We construct B in stages along with A. We start with B empty. Using the notation of Lemma 3.5, at the beginning of stage i, we add 0^{2^m} to B iff property (3.1) does not hold, and at the end of substage j, we join B with

$$\left\{\left\langle 0^{2^{m^{2^{j+1}}}}, y\right\rangle \mid y \in I_j \text{ and } r(y) = 1\right\} \text{ if (3.1) holds at stage } i,$$
$$\left\{\left\langle 0^{2^{m^{2^{j+1}}}}, y\right\rangle \mid y \in I_j \text{ and } \ell(y) = 1\right\} \text{ otherwise.}$$

Note that this does not affect the value of $K^B(y)$ for $|y| < m^{2^{j+1}}$ nor the computations of M_i on inputs of size at most m (for sufficiently large i such that $m^{\log m} < 2^m$). It follows from the analysis in the proof of Lemma 3.5 that the set A is $\leq_{2-\mathrm{T}}^{\mathcal{P}}$-hard for $\mathcal{EXP}^B$ and not $\leq_{\mathrm{T}}^{\mathcal{P}^B}$-autoreducible.

Regarding the complexity of deciding A relative to B, note that the encoding in the oracle B allows us to eliminate the need for evaluating $\mathrm{QBF}_{\log\beta(n)}$-formulae of size $2^{n^{\beta(n)}}$. Instead, we just query B on easily constructed inputs of size $O(2^{n^2})$. Therefore, we can drop the terms corresponding to the $\mathrm{QBF}_{\log\beta(n)}$-formulae of size $2^{n^{\beta(n)}}$ in the complexity of A. Consequently, $A \in \mathcal{EXP}^B$. ☐

Theorem 4.2 applies to any complexity class containing $\mathcal{EXP}$ that has Property 4.3, e.g., $\mathcal{EXPSPACE}$, $\mathcal{EEXP}$, $\mathcal{EEXPSPACE}$, etc.

Sometimes, the structure of the oracle allows us to get around the lack of local checkability of oracle queries. This is the case for oracles from the polynomial-time hierarchy, and leads to the following extension of Theorem 4.1.

THEOREM 4.4. *For any integer $k \geq 0$, every $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set for $\Delta_{k+1}^{\mathcal{EXP}}$ is autoreducible.*

The proof idea is as follows: Let A be a $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set accepted by the deterministic oracle Turing machine M with oracle TQBF_k. First note that there is a polynomial-time Turing machine N such that a query q belongs to the oracle TQBF_k iff

$$\exists\, y_1, \forall\, y_2, \ldots, Q_k\, y_k : N(q, y_1, y_2, \ldots, y_k) \text{ accepts,} \tag{4.2}$$

where the y_ℓ's are of size polynomial in $|q|$.

We consider the two purported computations of M on input x constructed in the proof of Theorem 4.1. One belongs to a party assuming $x \in A$, the other to a party assuming $x \notin A$. The computation corresponding to the right assumption is correct; the other one might not be.

Now suppose the computations differ and we are given a pointer to the first bit position where they disagree, which turns out to be the answer to an oracle query q. Then we can have the two parties play the k-round game underlying (4.2): The party claiming $q \in \mathrm{TQBF}_k$ plays the existentially quantified y_ℓ's, the other one the universally quantified y_ℓ's. The players' strategies will consist of computing the game history so far, determining their optimal next move, $\leq_{\mathrm{T}}^{\mathcal{P}}$-reducing this computation to A, and finally producing the result of this reduction under their respective assumption about $A(x)$. This will guarantee that the party with the correct assumption plays optimally. Since this is also the one claiming the correct answer to the oracle query q, he will win the game, i.e., $N(q, y_1, y_2, \ldots, y_k)$ will equal his answer bit.

The only thing the autoreduction for A has to do is determine the value of $N(q, y_1, y_2, \ldots, y_k)$ in polynomial time using A as an oracle but without querying x. It can do that along the lines of the base case algorithm given in Figure 4.1. If during this process the local consistency test for N's computation requires the knowledge of bits from the y_ℓ's, we compute these via the reduction defining the strategy of the corresponding player. The bits from q we need we can retrieve from the M-computations, since both computations are correct up to the point where they finished generating q. Once we know $N(q, y_1, y_2, \ldots, y_k)$ we can easily decide the correct assumption about $A(x)$.

The construction hinges on the hypothesis that we can $\leq_{\mathrm{T}}^{\mathcal{P}}$-reduce determining the player's moves to A. Computing these moves can become quite complex, though, because we have to recursively reconstruct the game history so far. The number of rounds k being constant seems crucial for keeping the complexity under control. The conference version of this paper [6] erroneously claimed the proof works for $\mathcal{EXPSPACE}$, which can be thought of as alternating exponential time with an exponential number of alternations. Establishing Theorem 4.4 for $\mathcal{EXPSPACE}$ would actually separate $\mathcal{NL}$ from $\mathcal{NP}$, as we will see in section 5.

Proof of Theorem 4.4. Let A be a $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete set for $\Delta_{k+1}^{\mathcal{EXP}} = \mathcal{EXP}^{\Sigma_k^{\mathcal{P}}}$ accepted by the exponential-time oracle Turing machine M with oracle TQBF_k. Let g_M, e_M, and c_M be the formalization of the local consistency test for M as described by (4.1). Without loss of generality there is a polynomial p and a polynomial-time Turing machine N such that on inputs of size n, M makes exactly $2^{p(n)}$ oracle queries, all of the form

$$\exists\, y_1 \in \Sigma^{2^{p(n)}}, \forall\, y_2 \in \Sigma^{2^{p(n)}}, \ldots, Q_k\, y_k \in \Sigma^{2^{p(n)}} : N(q, y_1, y_2, \ldots, y_k) \text{ accepts}, \tag{4.3}$$

where q has length $2^{p^2(n)}$. Moreover, the computations of N in (4.3) each have length $2^{p^3(n)}$, and their last bit represents the answer; the same holds for the computations of M on inputs of length n. Let g_N, e_N, and c_N be the formalization of the local consistency test for N.

We first define a bunch of functions computable in $\Delta_{k+1}^{\mathcal{EXP}}$. For each of them, say ξ, we fix an oracle Turing machine R_ξ that $\leq_{\mathrm{T}}^{\mathcal{P}}$-reduces ξ to A, and which the final autoreduction for A will use. The proofs that we can compute these functions in $\Delta_{k+1}^{\mathcal{EXP}}$ are straightforward.

Let $\mu(\langle x, i\rangle)$ denote the ith bit of the computation of M^{TQBF_k} on input x, and $\sigma(x)$ the first i (if any) such that $R_\mu^{A \setminus \{x\}}(\langle x, i\rangle) \neq R_\mu^{A \cup \{x\}}(\langle x, i\rangle)$. The roles of μ and σ are the same as in the proof of Theorem 4.1: We will use R_μ to figure out whether both possible answers for the oracle query "$x \in A$?" lead to the same final answer, and if not, use R_σ to find a pointer i to the first incorrect bit (in any) of the simulated computation getting the negative oracle answer $x \notin A$. If i turns out not to point to an oracle query, we can proceed as in the proof of Theorem 4.1. Otherwise, we will make use of the following functions and associated reductions to A.

We define the functions η_ℓ and y_ℓ inductively for $\ell = 1, \ldots, k$. At each level ℓ we first define η_ℓ, which induces a reduction R_{η_ℓ}, and then define y_ℓ based on R_{η_ℓ}. All of these functions take an input x such that the ith bit of $R_\mu^{A \setminus \{x\}}(\langle x, \cdot \rangle)$ is the answer to an oracle query (4.3), where $i = R_\sigma^{A \cup \{x\}}(x)$. We define $\eta_\ell(x)$ as the lexicographically least $y_\ell \in \Sigma^{2^{p(|x|)}}$ such that

$$\begin{aligned}&\chi[Q_{\ell+1}\, y_{\ell+1}, Q_{\ell+2}\, y_{\ell+2}, \ldots, Q_k\, y_k :\\ &\quad N(q, y_1(x), y_2(x), \ldots, y_{\ell-1}(x), y_\ell, y_{\ell+1}, \ldots, y_k) \text{ accepts}] \equiv \ell \bmod 2;\end{aligned} \tag{4.4}$$

if this value does not exist, we set $\eta_\ell(x) = 0^{2^{p(|x|)}}$. Note that the right-hand side of (4.4) is 1 iff y_ℓ is existentially quantified in (4.3).

$$y_\ell(x) = \begin{cases} R_{\eta_\ell}^{A \cup \{x\}}(x) & \text{if } \ell \equiv R_\mu^{A \cup \{x\}}(\langle x, i \rangle) \bmod 2, \\ R_{\eta_\ell}^{A \setminus \{x\}}(x) & \text{otherwise.} \end{cases} \tag{4.5}$$

The condition on the right-hand side of (4.5) means that we use the hypothesis $x \in A$ to compute $y_\ell(x)$ from R_{η_ℓ} in case

(i) either y_ℓ is existentially quantified in (4.3) and the player assuming $x \in A$ claims (4.3) holds,

(ii) or else y_ℓ is universally quantified and the player assuming $x \in A$ claims (4.3) fails.

Otherwise we use the hypothesis $x \notin A$.

In case i points to the answer to an oracle query (4.3), the functions η_ℓ and the reductions R_{η_ℓ} incorporate the moves during the successive rounds of the game underlying (4.3). The reduction R_{η_ℓ}, together with the player's assumption about membership of x to A, determines the actual move $y_\ell(x)$ during the ℓth round, namely $R_{\eta_\ell}^{A \cup \{x\}}(x)$ if the ℓth round is played by the opponent assuming $x \in A$, and $R_{\eta_\ell}^{A \setminus \{x\}}(x)$ otherwise. The condition on the right-hand side of (4.5) guarantees that the existentially quantified variables are determined by the opponent claiming the query (4.3) is a true formula, and the universally quantified ones by the other opponent. In particular, (4.5) ensures that the opponent with the correct claim about (4.3) has a winning strategy. Provided it exists, the function η_ℓ defines a winning move during the ℓth round of the game for the opponent playing that round, given the way the previous rounds were actually played (as described by the $y(x)$'s). For odd ℓ, i.e., y_ℓ is existentially quantified, it tries to set y_ℓ such that the remainder of (4.3) holds; otherwise it tries to set y_ℓ such that the remainder of (4.3) fails. The actual move may differ from the one given by η_ℓ in case the player's assumption about $x \in A$ is incorrect. The opponent with the correct assumption plays according to η_ℓ. Since that opponent also makes the correct claim about (4.3), he will win the game. In any case, $N(q, y_1, y_2, \ldots, y_k)$ will hold iff (4.3) holds.

Finally, we define the functions ν and τ, which have a similar job as the functions μ and σ, respectively, but this time for the computation of $N(q, y_1, y_2, \ldots, y_k)$ instead of the computation of $M_k^{\mathrm{TQBF}}(x)$. More precisely, $\nu(\langle x, r \rangle)$ equals the rth bit of the computation of $N(q, y_1(x), y_2(x), \ldots, y_k(x))$, where the $y_\ell(x)$'s are defined by

```
if $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) = R_\mu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$
  then accept iff $R_\mu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) = 1$
  else $i \leftarrow R_\sigma^{A\cup\{x\}}(x)$
    if the $i$th bit of $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is not the answer to an oracle query
      then accept iff $e_M(x; i, R_\mu^{A\setminus\{x\}}(\langle x, g_M(x; i, 1)\rangle), R_\mu^{A\setminus\{x\}}(\langle x, g_M(x; i, 2)\rangle),$
                          $\ldots, R_\mu^{A\setminus\{x\}}(\langle x, g_M(x; i, c_M)\rangle)) = 0$
      else if $R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) = R_\nu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$
        then accept iff $R_\mu^{A\setminus\{x\}}(\langle x, i\rangle) \neq R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$
        else $r \leftarrow R_\tau^{A\cup\{x\}}(x)$
          accept iff
            $e_N(q, y_1, y_2, \ldots, y_k; r, R_\nu^{A\setminus\{x\}}(\langle x, g_N(q, y_1, y_2, \ldots, y_k; r, 1)\rangle),$
                 $R_\nu^{A\setminus\{x\}}(\langle x, g_N(q, y_1, y_2, \ldots, y_k; r, 2)\rangle),$
                 $\ldots, R_\nu^{A\setminus\{x\}}(\langle x, g_N(q, y_1, y_2, \ldots, y_k; r, c_N)\rangle)) = 0$
          where $q$ denotes the query described in $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$
                to which the $i$th bit in this computation is the answer
            and
            $y_\ell = \begin{cases} R_{\eta_\ell}^{A\cup\{x\}}(x) & \text{if } \ell \equiv R_\mu^{A\cup\{x\}}(\langle x, i\rangle) \bmod 2, \\ R_{\eta_\ell}^{A\setminus\{x\}}(x) & \text{otherwise} \end{cases}$
      end if
    end if
end if
```

FIG. 4.2. *Autoreduction for the set A of Theorem* 4.4 *on input x.*

(4.5), and the bit with index $i = R_\sigma^{A\cup\{x\}}(x)$ in the computation $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is the answer to the oracle query (4.3). We define $\tau(x)$ to be the first r (if any) for which $R_\nu^{A\setminus\{x\}}(\langle x, r\rangle) \neq R_\nu^{A\cup\{x\}}(\langle x, r\rangle)$, provided the bit with index $i = R_\sigma^{A\cup\{x\}}(x)$ in the computation $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is the answer to an oracle query.

Now that we have these functions and corresponding reductions, we can describe an autoreduction for A. On input x, it works as described in Figure 4.2. We next argue that the algorithm correctly decides A on input x. Checking the other properties required of an autoreduction for A is straightforward.

We only consider the cases where $R_\mu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) \neq R_\mu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$ and i points to the answer to an oracle query in $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$. We refer to the analysis in the proof of Theorem 4.1 for the remaining cases.

(i) *Case* $R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) = R_\nu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$. If $x \in A$, variable i points to the first incorrect bit of $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$, which turns out to be the answer to an oracle query, say (4.3). Since $R_\nu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$ yields the correct oracle answer to (4.3),

$$R_\mu^{A\setminus\{x\}}(\langle x, i\rangle) \neq R_\nu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) = R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle),$$

and we accept x.

If $x \notin A$, both $R_\mu^{A\setminus\{x\}}(\langle x, i\rangle)$ and $R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$ give the correct answer to the oracle query i points to in the computation $R_\mu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$. Thus, they are equal, and we reject x.

(ii) *Case* $R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle) \neq R_\nu^{A\cup\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$. As described in Figure 4.2, we will use the local consistency test for $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ being the computation of $N(q, y_1(x), y_2(x), \ldots, y_k(x))$. Apart from bits in the purported computation $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$, this test may also need bits from q and from the $y_\ell(x)$'s. The $y_\ell(x)$'s can be computed straightforwardly using their definition (4.5). The bits from q we might need can be retrieved from $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$. This is because our encoding scheme for computations has the property that the query q is a substring of the prefix of the computation up to the position indexed by i. Since either $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is correct everywhere, or else i is the first position where it is incorrect, the description of q in $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is correct in any case. Moreover, we can easily compute a pointer to the beginning of the substring q of $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ from i.

If $x \in A$, $R_\nu^{A\setminus\{x\}}(\langle x, 2^{p^3(|x|)}\rangle)$ is incorrect, so $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ has an error as a computation of $N(q, y_1(x), y_2(x), \ldots, y_k(x))$. Variable r gets assigned the index of the first incorrect bit in this computation, so the local consistency check fails, and we accept x.

If $x \notin A$, $R_\nu^{A\setminus\{x\}}(\langle x, \cdot\rangle)$ is a valid computation of $N(q, y_1(x), y_2(x), \ldots, y_k(x))$, so every local consistency test is passed, and we reject x. $\square$

4.2. Nonadaptive autoreductions. Thus far, we have constructed autoreductions for $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets A. On input x we looked at the two candidate computations obtained by reducing to A, answering all oracle queries except for x according to A, and answering query x positively for one candidate, and negatively for the other. If the candidates disagreed, we tried to find out the right one, which always existed. We managed to get the idea to work for quite powerful sets A, e.g., $\mathcal{EXP}$-complete sets, by exploiting the local checkability of computations. That allowed us to figure out the wrong computation without going through the entire computation ourselves: With help from A, we first computed a pointer to the first mistake in the wrong computation, and then verified it locally.

We cannot use this adaptive approach for constructing nonadaptive autoreductions. It seems like figuring out the wrong computation in a nonadaptive way requires the autoreduction to perform the computation of the base machine itself, so the base machine has to run in polynomial time. Then checking the computation essentially boils down to verifying the oracle answers. Using the game characterization of the polynomial-time hierarchy along the same lines as in Theorem 4.4, we can do this for oracles from the polynomial-time hierarchy.

THEOREM 4.5. *For any integer $k \geq 0$, every $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete set for $\Delta_{k+1}^{\mathcal{P}}$ is nonadaptively autoreducible.*

Parallel to the adaptive case, an earlier version of this paper [6] stated Theorem 4.5 for unbounded k, i.e., for $\mathcal{PSPACE}$. However, we only get the proof to work for constant k. In section 5, we will see that proving Theorem 4.5 for $\mathcal{PSPACE}$ would separate $\mathcal{NL}$ from $\mathcal{NP}$.

The only additional difficulty in the proof is that in the nonadaptive setting, we do not know which player has to perform the even rounds, and which one the odd rounds in the k-round game underlying a query like (4.2). But we can just have them play both scenarios, and afterwards figure out the relevant run.

Proof of Theorem 4.5. Let A be a $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete set for $\Delta_{k+1}^{\mathcal{P}} = \mathcal{P}^{\Sigma_k^{\mathrm{P}}}$ accepted by the polynomial-time oracle Turing machine M with oracle TQBF$_k$. Without loss of generality there is a polynomial p and a polynomial-time Turing machine N such that on inputs of size n, M makes exactly $p(n)$ oracle queries q, all of the form

$$(4.6)\quad \exists\, y_1 \in \Sigma^{p(n)}, \forall\, y_2 \in \Sigma^{p(n)}, \ldots, Q_k\, y_k \in \Sigma^{p(n)} : N(q, y_1, y_2, \ldots, y_k) \text{ accepts},$$

where q has length $p^2(n)$. Let $q(x,i)$ denote the ith oracle query of M^{TQBF_k} on input x. Note that $q \in \mathrm{FP}^{\Sigma_k^{\mathcal{P}}}$.

Let $Q = \{\langle x,i\rangle \mid q(x,i) \in \mathrm{TQBF}_k\}$. The set Q belongs to $\Delta_{k+1}^{\mathcal{P}}$, so there is a $\leq_{\mathrm{tt}}^{\mathcal{P}}$-reduction R_Q from Q to A.

If for a given input x $R_Q^{A\cup\{x\}}$ and $R_Q^{A\setminus\{x\}}$ agree on $\langle x,j\rangle$ for every $1 \leq j \leq p(|x|)$, we are home: We can simulate the base machine M using $R_Q^{A\cup\{x\}}(\langle x,j\rangle)$ as the answer to the jth oracle query.

Otherwise, we will make use of the following functions $\eta_1, \eta_2, \dots, \eta_k$ computable in $\Delta_{k+1}^{\mathcal{P}}$, corresponding oracle Turing machines $R_{\eta_1}, R_{\eta_2}, \dots, R_{\eta_k}$ defining $\leq_{\mathrm{tt}}^{\mathcal{P}}$-reductions to A, and functions $y_1, y_2, \dots, y_k$ also computable in $\Delta_{k+1}^{\mathcal{P}}$. As in the proof of Theorem 4.4, we define η_ℓ and y_ℓ inductively for $\ell = 1, \dots, k$. They are defined for inputs x such that there is a smallest $1 \leq i \leq p(|x|)$ for which $R_Q^{A\setminus\{x\}}(\langle x,i\rangle) \neq R_Q^{A\cup\{x\}}(\langle x,i\rangle)$. The value of $\eta_\ell(x)$ equals the lexicographically least $y_\ell \in \Sigma^{p(|x|)}$ such that

$$\begin{aligned}&\chi[Q_{\ell+1}\, y_{\ell+1}, Q_{\ell+2}\, y_{\ell+2}, \dots, Q_k\, y_k :\\ &\quad N(q(x,i), y_1(x), y_2(x), \dots, y_{\ell-1}(x), y_\ell, y_{\ell+1}, \dots, y_k) \text{ accepts}] \equiv \ell \bmod 2;\end{aligned} \tag{4.7}$$

we set $\eta_\ell(x) = 0^{p(|x|)}$ if such string does not exist. The right-hand side of (4.7) is 1 iff y_ℓ is existentially quantified in (4.6).

$$y_\ell = \begin{cases} R_{\eta_\ell}^{A\cup\{x\}}(x) & \text{if } \ell \equiv R_Q^{A\cup\{x\}}(\langle x,i\rangle) \bmod 2,\\ R_{\eta_\ell}^{A\setminus\{x\}}(x) & \text{otherwise.}\end{cases} \tag{4.8}$$

The condition on the right-hand side of (4.8) means that we use the hypothesis $x \in A$ to compute $y_\ell(x)$ from R_{η_ℓ} in case

(i) either y_ℓ is existentially quantified in (4.6) and the assumption $x \in A$ leads to claiming that (4.6) holds,

(ii) or else y_ℓ is universally quantified and the assumption $x \in A$ leads to claiming that (4.6) fails.

The intuitive meaning of the functions η_ℓ and the reductions R_{η_ℓ} is similar to in the proof of Theorem 4.4: They capture the moves during the ℓth round of the game underlying (4.6) for $q = q(x,i)$. The function η_ℓ encapsulates an optimal move during round ℓ if it exists, and the reduction R_{η_ℓ} under the player's assumption regarding membership of x to A produces the actual move in that round. The condition on the right-hand side of (4.8) guarantees the correct alternation of rounds. We refer to the proof of Theorem 4.4 for more intuition.

Consider the algorithm in Figure 4.3. Note that the only queries to A the algorithm in Figure 4.3 needs to make are the queries of R_Q different from x on inputs $\langle x,j\rangle$ for $1 \leq j \leq p(|x|)$ and the queries of R_{η_ℓ} different from x on input x for $1 \leq \ell \leq k$. Since R_Q and the R_{η_ℓ}'s are nonadaptive, it follows that Figure 4.3 describes a $\leq_{\mathrm{tt}}^{\mathcal{P}}$-reduction to A that does not query its own input. A similar but simplified argument as in the proof of Theorem 4.4 shows that it accepts A. Thus, A is nonadaptively autoreducible. ∎

Next, we consider more restricted reductions. Using a different technique, we arrive at the following theorem.

THEOREM 4.6. *For any complexity class $\mathcal{C}$, every $\leq_{2-\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{C}$ is $\leq_{2-\mathrm{tt}}^{\mathcal{P}}$-autoreducible, provided $\mathcal{C}$ is closed under exponential-time reductions that only ask one query which is smaller in length.*

```
if $R_Q^{A\setminus\{x\}}(\langle x,j\rangle) = R_Q^{A\cup\{x\}}(\langle x,j\rangle)$ for every $1 \le j \le p(|x|)$
  then accept iff $M$ accepts $x$ when the $j$th oracle query is answered $R_Q^{A\cup\{x\}}(\langle x,j\rangle)$
  else $i \leftarrow$ first $j$ such that $R_Q^{A\setminus\{x\}}(\langle x,j\rangle) \neq R_Q^{A\cup\{x\}}(\langle x,j\rangle)$
      accept iff $N(q, y_1, y_2, \dots, y_k) = R_Q^{A\cup\{x\}}(\langle x,i\rangle)$
                 where $q$ denotes the $i$th query of $M$ on input $x$
                       when the answer to the $j$th oracle query is given by $R_Q^{A\cup\{x\}}(\langle x,j\rangle)$
                 and
                 $y_\ell = \begin{cases} R_{\eta_\ell}^{A\cup\{x\}}(x) & \text{if } \ell \equiv R_Q^{A\cup\{x\}}(\langle x,i\rangle) \bmod 2, \\ R_{\eta_\ell}^{A\setminus\{x\}}(x) & \text{otherwise} \end{cases}$
  end if
```

FIG. 4.3. *Nonadaptive autoreduction for the set A of Theorem* 4.5 *on input x.*

```
case truth-table of $M_i$ on input $\langle 0^i, x\rangle$ with the truth-value of query $x$ set to $A(x)$
  constant:
    accept iff $M_i^A$ rejects $\langle 0^i, x\rangle$
  of the form "$y \notin A$":
    accept iff $x \notin A$
  otherwise:
    accept iff $x \in A$
  end case
```

FIG. 4.4. *Algorithm for the set D of Theorem* 4.6 *on input $\langle 0^i, x\rangle$.*

In particular, Theorem 4.6 applies to $\mathcal{C} = \mathcal{EXP}$, $\mathcal{EXPSPACE}$, and $\mathcal{EEXPSPACE}$. In view of Theorems 3.1 and 3.3, this implies that Theorems 3.1, 3.3, and 4.6 are optimal.

The proof exploits the ability of $\mathcal{EXP}$ to simulate all polynomial-time reductions to construct an auxiliary set D within $\mathcal{C}$ such that any $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-reductions of D to some fixed complete set A has a property that induces an autoreduction on A.

Proof of Theorem 4.6. Let $M_1, M_2, \dots$ be a standard enumeration of $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-reductions such that M_i runs in time n^i on inputs of size n. Let A be a $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{C}$.

Consider the set D that only contains strings of the form $\langle 0^i, x\rangle$ for $i \in \mathbb{N}$ and $x \in \Sigma^*$, and is decided by the algorithm of Figure 4.4 on such an input. Except for deciding $A(x)$, the algorithm runs in exponential time. Therefore, under the given conditions on $\mathcal{C}$, $D \in \mathcal{C}$, so there is a $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-reduction M_j from D to A.

The construction of D diagonalizes against every $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-reduction M_i of D to A whose truth-table on input $\langle 0^i, x\rangle$ would become constant once we filled in the membership bit for x. Therefore, for every input x, one of the following cases holds for the truth-table of M_j on input $\langle 0^j, x\rangle$.

(i) The reduced truth-table is of the form "$y \in A$" with $y \neq x$. Then $y \in A \Leftrightarrow M_j$ accepts $\langle 0^j, x\rangle \Leftrightarrow x \in A$.

(ii) The reduced truth-table is of the form "$y \notin A$" with $y \neq x$. Then $y \notin A \Leftrightarrow M_j$ accepts $\langle 0^j, x\rangle \Leftrightarrow x \notin A$.

(iii) The truth-table depends on the membership to A of two strings different from x. Then M_j^A does not query x on input $\langle 0^j, x\rangle$, and accepts iff $x \in A$.

The above analysis shows that the algorithm of Figure 4.5 describes a $\le_{2-\mathrm{tt}}^{\mathcal{P}}$-

```
if $|Q_{M_j}(\langle 0^j, x\rangle) \setminus \{x\}| = 2$
   then accept iff $M_j^A$ accepts $\langle 0^j, x\rangle$
   else { $|Q_{M_j}(\langle 0^j, x\rangle) \setminus \{x\}| = 1$ }
         $y \leftarrow$ unique element of $Q_{M_j}(\langle 0^j, x\rangle) \setminus \{x\}$
         accept iff $y \in A$
   endif
```

FIG. 4.5. *Autoreduction constructed in the proof of Theorem* 4.6.

reduction of A. ∎

4.3. Probabilistic and nonuniform autoreductions. The previous results in this section trivially imply that the $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets for the Δ-levels of the exponential-time hierarchy are probabilistically autoreducible, and the $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for the Δ-levels of the polynomial-time hierarchy are probabilistically nonadaptively autoreducible. Randomness allows us to prove more in the nonadaptive case.

First, we can establish Theorem 4.5 for $\mathcal{EXP}$.

THEOREM 4.7. *Let f be a constructible function. Every $\leq_{f(n)-\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{EXP}$ is probabilistically $\leq_{O(f(n))-\mathrm{tt}}^{\mathcal{P}}$-autoreducible. In particular, every $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{EXP}$ is probabilistically nonadaptively autoreducible.*

Proof. Let A be a $\leq_{f(n)-\mathrm{tt}}^{\mathcal{P}}$-complete set for $\mathcal{EXP}$. We will apply the PCP Theorem for $\mathcal{EXP}$ [2] to A.

LEMMA 4.8 (see [2]). *There is a constant k such that for any set $A \in \mathcal{EXP}$, there is a polynomial-time Turing machine V and a polynomial p such that for any input x:*

(i) *If $x \in A$, then there exists a proof oracle π such that*

$$\Pr_{|r|=p(|x|)}[V^{\pi}(x,r) \text{ accepts }] = 1. \tag{4.9}$$

(ii) *If $x \notin A$, then for any proof oracle π*

$$\Pr_{|r|=p(|x|)}[V^{\pi}(x,r) \text{ accepts }] \leq \frac{1}{3}.$$

Moreover, V never makes more than k proof oracle queries, and there is a proof oracle $\tilde{\pi} \in \mathcal{EXP}$ independent of x such that (4.9) *holds for $\pi = \tilde{\pi}$ in case $x \in A$.*

Translating Lemma 4.8 into our terminology, we obtain the following lemma.

LEMMA 4.9. *There is a constant k such that for any set $A \in \mathcal{EXP}$ there is a probabilistic $\leq_{k-\mathrm{tt}}^{\mathcal{P}}$-reduction N, and a set $B \in \mathcal{EXP}$ such that for any input x:*

(i) *If $x \in A$, then $N^B(x)$ always accepts.*

(ii) *If $x \notin A$, then for any oracle C, $N^C(x)$ accepts with probability at most $\frac{1}{3}$.*

Let R be a $\leq_{f(n)-\mathrm{tt}}^{\mathcal{P}}$-reduction of B to A, and consider the probabilistic reduction M^A that on input x, runs N on input x with oracle $R^{A\cup\{x\}}$. M^A is a probabilistic $\leq_{k\cdot f(n)-\mathrm{tt}}^{\mathcal{P}}$-reduction to A that never queries its own input. The following shows it defines a reduction from A:

(i) If $x \in A$, then $R^{A\cup\{x\}} = R^A = B$, so $M^A(x) = N^B(x)$ always accepts.

(ii) If $x \notin A$, then for $C = R^{A\cup\{x\}}$, $M^A(x) = N^C(x)$ accepts with probability at most $\frac{1}{3}$. ∎

Note that Theorem 4.7 makes it plausible why we did not manage to scale down Theorem 3.2 by one exponent to $\mathcal{EXPSPACE}$ in the nonadaptive setting, as we were able to do for our other results in section 3 when going from the adaptive to the nonadaptive case: This would separate $\mathcal{EXP}$ from $\mathcal{EXPSPACE}$.

We suggest the extension of Theorem 4.7 to the Δ-levels of the exponential-time hierarchy as an interesting problem for further research.

Second, Theorem 4.5 also holds for $\mathcal{NP}$.

THEOREM 4.10. *All $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for $\mathcal{NP}$ are probabilistically nonadaptively autoreducible.*

Proof. Fix a $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete set A for $\mathcal{NP}$. Let R_A denote a length nondecreasing $\leq_{\mathrm{m}}^{\mathcal{P}}$-reduction of A to SAT.

Define the set

$$W = \{\langle \phi, 0^i \rangle \mid \phi \text{ is a formula with say } m \text{ variables and } \exists\, a \in \Sigma^m : [\phi(a) \text{ and } a_i = 1]\}.$$

Since $W \in \mathcal{NP}$, there is a $\leq_{\mathrm{tt}}^{\mathcal{P}}$-reduction R_W from W to A.

We will use the following probabilistic algorithm by Valiant and Vazirani [17].

LEMMA 4.11 (see [17]). *There exists a polynomial-time probabilistic Turing machine N that on input a Boolean formula φ with n variables, outputs another quantifier free Boolean formula $\phi = N(\varphi)$ such that:*

(i) *If φ is satisfiable, then with probability at least $\frac{1}{4n}$, ϕ has a unique satisfying assignment.*

(ii) *If φ is not satisfiable, then ϕ is never satisfiable.*

Now consider the following algorithm for A: On input x, run N on input $R_A(x)$, yielding a Boolean formula ϕ with, say m variables, and it accepts iff

$$\phi(R_W^{A\cup\{x\}}(\langle \phi, 0\rangle), R_W^{A\cup\{x\}}(\langle \phi, 00\rangle), \ldots, R_W^{A\cup\{x\}}(\langle \phi, 0^i\rangle), \ldots, R_W^{A\cup\{x\}}(\langle \phi, 0^m\rangle))$$

evaluates to true. Note that this algorithm describes a probabilistic $\leq_{\mathrm{tt}}^{\mathcal{P}}$-reduction to A that never queries its own input. Moreover, we have the following:

(i) If $x \in A$, then with probability at least $\frac{1}{4|x|}$, the Valiant–Vazirani algorithm N produces a Boolean formula ϕ with a unique satisfying assignment $\tilde{a}_\phi$. In that case, $(R_W^{A\cup\{x\}}(\langle \phi, 0\rangle), R_W^{A\cup\{x\}}(\langle \phi, 00\rangle), \ldots, R_W^{A\cup\{x\}}(\langle \phi, 0^i\rangle), \ldots, R_W^{A\cup\{x\}}(\langle \phi, 0^m\rangle))$ equals $\tilde{a}_\phi$, and we accept x.

(ii) If $x \notin A$, any Boolean formula ϕ which N produces has no satisfying assignment, so we always reject x.

Executing $\Theta(n)$ independent runs of this algorithm, and accepting iff any of them accepts, yields a probabilistic nonadaptive autoreduction for A. ∎

Thus, for probabilistic autoreductions, we get similar results as for deterministic ones: Low end complexity classes turn out to have the property that their complete sets are autoreducible, whereas high end complexity classes do not. As we will see in more detail in the next section, this structural difference yields separations.

If we allow nonuniformity, the situation changes dramatically. Since probabilistic autoreducibility implies nonuniform autoreducibility [5], all our positive results for small complexity classes carry over to the nonuniform setting. But, as we will see next, the negative results do not, because also the complete sets for large complexity classes become autoreducible, both in the adaptive and in the nonadaptive case. Thus, uniformity is crucial for separating complexity classes using autoreducibility, and the Razborov–Rudich result [14] does not apply.

Feigenbaum and Fortnow [7] define the following concept of $\#\mathcal{P}$-robustness, of which we also consider the nonadaptive variant.

Table 5.1
Separation results using autoreducibility.

question	yes	no
Are all $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets for $\mathcal{EXPSPACE}$ autoreducible?	$\mathcal{NL} \neq \mathcal{NP}$	$\mathcal{PH} \neq \mathcal{PSPACE}$
Are all $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets for $\mathcal{EEXP}$ autoreducible?	$\mathcal{NL} \neq \mathcal{NP}$ $\mathcal{P} \neq \mathcal{PSPACE}$	$\mathcal{PH} \neq \mathcal{EXP}$
Are all $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for $\mathcal{PSPACE}$ $\leq_{\mathrm{tt}}^{\mathcal{P}}$-autoreducible?	$\mathcal{NL} \neq \mathcal{NP}$	$\mathcal{PH} \neq \mathcal{PSPACE}$
Are all $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for $\mathcal{EXP}$ $\leq_{\mathrm{tt}}^{\mathcal{P}}$-autoreducible?	$\mathcal{NL} \neq \mathcal{NP}$ $\mathcal{P} \neq \mathcal{PSPACE}$	$\mathcal{PH} \neq \mathcal{EXP}$
Are all $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for $\mathcal{EXPSPACE}$ probabilistically $\leq_{\mathrm{tt}}^{\mathcal{P}}$-autoreducible?	$\mathcal{NL} \neq \mathcal{NP}$	$\mathcal{P} \neq \mathcal{PSPACE}$

Definition 4.12. *A set A is* $\#\mathcal{P}$-robust *if* $\#\mathcal{P}^A \subseteq \mathcal{FP}^A$; *A is* nonadaptively $\#\mathcal{P}$-robust *if* $\#\mathcal{P}_{\mathrm{tt}}^A \subseteq \mathcal{FP}_{\mathrm{tt}}^A$.

Nonadaptive $\#\mathcal{P}$-robustness implies $\#\mathcal{P}$-robustness. For the usual deterministic and nondeterministic complexity classes containing $\mathcal{PSPACE}$, all $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets are $\#\mathcal{P}$-robust. For the deterministic classes containing $\mathcal{PSPACE}$, it is also true that the $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets are nonadaptively $\#\mathcal{P}$-robust.

The following connection with nonuniform autoreducibility holds.

Theorem 4.13. *All $\#\mathcal{P}$-robust sets are nonuniformly autoreducible. All nonadaptively $\#\mathcal{P}$-robust sets are nonuniformly nonadaptively autoreducible.*

Proof. Feigenbaum and Fortnow [7] show that every $\#\mathcal{P}$-robust language is random-self-reducible. Beigel and Feigenbaum [5] prove that every random-self-reducible set is nonuniformly autoreducible (or "weakly coherent," as they call it). Their proofs carry over to the nonadaptive setting. □

It follows that the $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for the usual deterministic complexity classes containing $\mathcal{PSPACE}$ are all nonuniformly nonadaptively autoreducible. The same holds for adaptive reductions, in which case the property is also true of nondeterministic complexity classes containing $\mathcal{PSPACE}$. In particular, we get the following corollary.

Corollary 4.14. *All $\leq_{\mathrm{T}}^{\mathcal{P}}$-complete sets for $\mathcal{NEXP}$, $\mathcal{EXPSPACE}$, $\mathcal{EEXP}$, $\mathcal{NEEXP}$, $\mathcal{EEXPSPACE}$,... are nonuniformly autoreducible. All $\leq_{\mathrm{tt}}^{\mathcal{P}}$-complete sets for $\mathcal{PSPACE}$, $\mathcal{EXP}$, $\mathcal{EXPSPACE}$,... are nonuniformly nonadaptively autoreducible.*

5. Separation results.

In this section, we will see how we can use the structural property of all complete sets being autoreducible to separate complexity classes. Based on the results of sections 3 and 4, we only get separations that were already known: $\mathcal{EXPH} \neq \mathcal{EEXPSPACE}$ (by Theorems 4.4 and 3.1), $\mathcal{EXP} \neq \mathcal{EEXPSPACE}$ (by Theorems 4.7 and 3.2), and $\mathcal{PH} \neq \mathcal{EXPSPACE}$ (by Theorems 4.5 and 3.3, and also by scaling down $\mathcal{EXPH} \neq \mathcal{EEXPSPACE}$). However, settling the question for certain other classes would yield impressive new separations.

We summarize the implications in Table 5.1.

Theorem 5.1. *In Table* 5.1, *a positive answer to a question from the first column implies the separation in the second column, and a negative answer implies the separation in the third column.*

Most of the entries in Table 5.1 follow directly from the results of the previous sections. In order to finish the table, we use the next lemma.

LEMMA 5.2. *If $\mathcal{NP} = \mathcal{NL}$, we can decide the validity of QBF-formulae of size t and with γ alternations on a deterministic Turing machine M_1 in time $t^{O(c^\gamma)}$ and on a nondeterministic Turing machine M_2 in space $O(c^\gamma \log t)$, for some constant c.*

Proof. Since $co\mathcal{NP} = \mathcal{NP}$, by Cook's theorem we can transform in polynomial time a Π_1-formula with free variables into an equivalent Σ_1-formula with the same free variables, and vice versa. Since $\mathcal{NP} = \mathcal{P}$, we can decide the validity of Σ_1-formulae in polynomial-time. Say both the transformation algorithm T and the satisfiability algorithm S run in time n^c for some constant c.

Let ϕ be a QBF-formula of size t with γ alternations. Consider the following algorithm for deciding ϕ: Repeatedly apply the transformation T to the largest suffix that constitutes a Σ_1- or Π_1-formula until the whole formula becomes Σ_1, and then run S on it.

This algorithm correctly decides the truth of ϕ. Since the number of alternations decreases by one during every iteration, it makes at most γ calls to T, each time at most raising the length of the formula to the power c. It follows that the algorithm runs in time $t^{O(c^\gamma)}$.

Moreover, since $\mathcal{P} = \mathcal{NL}$, a padding argument shows that DTIME$[\tau]$ is contained in NSPACE$[\log \tau]$ for any time constructible function τ. Therefore, the result holds. ∎

This allows us to improve Theorems 3.2 and 3.3 as follows under the hypothesis $\mathcal{NP} = \mathcal{NL}$.

THEOREM 5.3. *If $\mathcal{NP} = \mathcal{NL}$, there is a $\leq^{\mathcal{P}}_{2-\mathrm{T}}$-complete set for $\mathcal{EXPSPACE}$ that is not probabilistically autoreducible. The same holds for $\mathcal{EEXP}$ instead of $\mathcal{EXPSPACE}$.*

Proof. Combine Lemma 5.2 with the probabilistic extension of Lemma 3.5 used in the proof of Theorem 3.2. ∎

THEOREM 5.4. *If $\mathcal{NP} = \mathcal{NL}$, there is a $\leq^{\mathcal{P}}_{3-\mathrm{tt}}$-complete set for $\mathcal{PSPACE}$ that is not nonadaptively autoreducible. The same holds for $\mathcal{EXP}$ instead of $\mathcal{PSPACE}$.*

Proof. Combining Lemma 5.2 with Lemma 3.7 for $\alpha(n) = n$ yields the result for $\mathcal{EXP}$. The one for $\mathcal{PSPACE}$ follows, since $\mathcal{NP} = \mathcal{NL}$ implies that $\mathcal{EXP} = \mathcal{PSPACE}$. ∎

Now, we have all ingredients for establishing Table 5.1.

Proof of Theorem 5.1. The $\mathcal{NL} \neq \mathcal{NP}$ implications in the "yes"-column of Table 5.1 immediately follow from Theorems 5.3 and 5.4 by contraposition.

By Theorem 3.1, a positive answer to the second question in Table 5.1 would yield $\mathcal{EEXP} \neq \mathcal{EEXPSPACE}$, and by Theorem 3.3, a positive answer to the fourth question would imply $\mathcal{EXP} \neq \mathcal{EXPSPACE}$. By padding, both translate down to $\mathcal{P} \neq \mathcal{PSPACE}$.

Similarly, by Theorem 4.4, a negative answer to the second question would imply $\mathcal{EXPH} \neq \mathcal{EEXP}$, which pads down to $\mathcal{PH} \neq \mathcal{EXP}$. A negative answer to the fourth question would yield $\mathcal{PH} \neq \mathcal{EXP}$ directly by Theorem 4.5. By the same token, a negative answer to the first question results in $\mathcal{EXPH} \neq \mathcal{EXPSPACE}$ and $\mathcal{PH} \neq \mathcal{PSPACE}$, and a negative answer to the third question in $\mathcal{PH} \neq \mathcal{PSPACE}$. By Theorem 4.7, a negative answer to the last question implies $\mathcal{EXP} \neq \mathcal{EXPSPACE}$ and $\mathcal{P} \neq \mathcal{PSPACE}$. ∎

We note that we can tighten all of the separations in Table 5.1 a bit, because we can apply Lemmas 3.5 and 3.7 to smaller classes than in Theorems 3.1 and 3.3, respectively. One improvement along these lines that warrants attention is replacing "$\mathcal{NL} \neq \mathcal{NP}$" in Table 5.1 with "$co\mathcal{NP} \not\subseteq \mathcal{NP} \cap \mathrm{NSPACE}[\log^{O(1)} n]$." This is because that condition suffices for Theorems 5.3 and 5.4, since we can strengthen Lemma 5.2 as follows.

LEMMA 5.5. *If co$\mathcal{NP} \subseteq \mathcal{NP} \cap \mathrm{NSPACE}[\log^{O(1)} n]$, we can decide the validity of QBF-formulae of size t and with γ alternations on a deterministic Turing machine M_1 in time $t^{O(c^\gamma)}$ and on a nondeterministic Turing machine M_2 in space $O(d^\gamma \log^d t)$, for some constants c and d.*

6. Conclusion.

We have studied the question of whether all complete sets are autoreducible for various complexity classes and various reducibilities. We obtained a positive answer for lower complexity classes in section 4 and a negative one for higher complexity classes in section 3. This way we separated the lower complexity classes from the higher ones by highlighting a structural difference. The resulting separations were not new, but we argued in section 5 that settling the very same question for intermediate complexity classes would provide major new separations.

We believe that refinements to our techniques may lead to these separations, and we would like to end with some thoughts in that direction.

One does not have to look at complete sets only. Let $\mathcal{C}_1 \subseteq \mathcal{C}_2$. Suppose we know that all complete sets for $\mathcal{C}_2$ are autoreducible. Then it suffices to construct, e.g., along the lines of Lemma 3.5, a hard set for $\mathcal{C}_1$ that is not autoreducible, in order to separate $\mathcal{C}_1$ from $\mathcal{C}_2$.

As we mentioned at the end of section 5, we can improve Theorem 3.1 a bit by applying Lemma 3.5 to smaller space-bounded classes than $\mathcal{EEXPSPACE}$. We cannot hope to gain much, though, since the coding in the proof of Lemma 3.5 seems to be $\mathrm{DSPACE}[2^{n^{\beta(n)}}]$-complete because of the $\mathrm{QBF}_{2\log\beta(n)}$-formulae of size $2^{O(n^{\beta(n)})}$ involved for inputs of size n. The same holds for Theorem 3.3 and Lemma 3.7.

Generalizations of autoreducibility may allow us to push things further. For example, one could look at $k(n)$-autoreducibility where $k(n)$ bits of the set remain unknown to the querying machine. Theorem 4.4 goes through for $k(n) \in O(\log n)$. Perhaps one can exploit this leeway in the coding of Lemma 3.5 and narrow the gap between the positive and negative results. As discussed in section 5, this would yield interesting separations.

Finally, one may want to look at other properties than autoreducibility to realize Post's program in complexity theory. Perhaps another concept from computability theory or a more artificial property can be used to separate complexity classes.

Acknowledgments. We would like to thank Manindra Agrawal and Ashish Naik for very helpful discussions. We are also grateful to Carsten Lund and Muli Safra for answering questions regarding the PCP theorem. We thank the anonymous referees for their nice suggestions on how to present our results and for their meticulous proofreading.

REFERENCES

[1] K. AMBOS-SPIES, *P-mitotic sets*, in Logic and Machines: Decision Problems and Complexity, E. Börger, G. Hasenjäger, and D. Roding, eds., Lecture Notes in Comput. Sci. 171, Springer-Verlag, Berlin, New York, 1984, pp. 1–23.

[2] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and the hardness of approximation problems*, J. ACM, 45 (1998), pp. 501–555.

[3] J. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* II, Monogr. Theoret. Comput. Sci. EATCS Ser. 22, Springer-Verlag, Berlin, 1990.

[4] J. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* I, Monogr. Theoret. Comput. Sci. EATCS Ser. 11, Springer-Verlag, Berlin, 1995.

[5] R. Beigel and J. Feigenbaum, *On being incoherent without being very hard*, Comput. Complexity, 2 (1992), pp. 1–17.

[6] H. Buhrman, L. Fortnow, and L. Torenvliet, *Using autoreducibility to separate complexity classes*, in Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1995, pp. 520–528.

[7] J. Feigenbaum and L. Fortnow, *Random-self-reducibility of complete sets*, SIAM J. Comput., 22 (1993), pp. 994–1005.

[8] L. Fortnow, *The role of relativization in complexity theory*, Bull. European Assoc. Theoret. Comput. Sci., 52 (1994), pp. 229–244.

[9] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[10] R. Ladner, *Mitotic recursively enumerable sets*, J. Symbolic Logic, 38 (1973), pp. 199–211.

[11] P. Odifreddi, *Classical Recursion Theory*, Stud. Logic Found. Math. 125, North–Holland, Amsterdam, 1989.

[12] C. Papadimitriou, *Computational Complexity*, Addison–Wesley, Reading, MA, 1994.

[13] E. Post, *Recursively enumerable sets of positive integers and their decision problems*, Bull. Amer. Math. Soc., 50 (1944), pp. 284–316.

[14] A. Razborov and S. Rudich, *Natural proofs*, J. Comput. System Sci., 55 (1997), pp. 24–35.

[15] R. Soare, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, New York, 1987.

[16] B. Trakhtenbrot, *On autoreducibility*, Dokl. Akad. Nauk SSSR, 192 (1970), pp. 1224–1227 (in Russian); Soviet Mathematics–Doklady, 11 (1970), pp. 814–817 (in English).

[17] L. Valiant and V. Vazirani, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.

[18] A. Yao, *Coherent functions and program checkers*, in Proceedings of the 22nd ACM Symposium on the Theory of Computing, Baltimore, MD, 1990, pp. 84–94.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1521–1549

MAINTENANCE OF 2- AND 3-EDGE-CONNECTED COMPONENTS OF GRAPHS II*

HAN LA POUTRÉ†

Abstract. Data structures and algorithms are presented to efficiently maintain the 2- and 3-edge-connected components of a general graph, under insertions of edges and nodes in the graph. At any moment, the data structure can answer whether two nodes are 2- or 3-edge-connected. The algorithms run in $O(n+m.\alpha(m,n))$ time, where m is the total number of queries and edge insertions. Furthermore, a linear-time algorithm is presented for maintaining the 2-edge-connected components in case the initial graph is connected. Finally, a new solution is presented for the 2-vertex-connected components of a graph.

Key words. analysis of algorithms, dynamic data structures, edge connectivity, vertex connectivity

AMS subject classifications. 68P05, 68Q20, 68R10

PII. S0097539793257770

1. Introduction. A graph algorithm is called dynamic if it maintains some information related to a graph while the graph is being changed. Dynamic algorithms are known for several graph problems. Examples are, e.g., maintenance of transitive closures [16, 17, 18, 27], minimal spanning trees [7, 8], planarity testing [3, 4, 6, 25, 33], shortest paths [1, 2, 29], k-connectivity [4, 5, 8, 9, 14, 21, 23, 24, 28, 32, 34, 35], and nearest common ancestors in trees [12].

The general problem of maintaining the k-edge- or k-vertex-connected components of a graph ($k \geq 1$) starts with an "empty" graph of n nodes[1] (i.e., a graph with no edges) and allows subsequent edge insertions and queries that ask whether two nodes are k-edge/vertex-connected.[2] For these problems, a lower bound of $\Omega(n+m.\alpha(m,n))$ [32] exists,[3] which is induced by lower bounds for set merging algorithms [10, 20]. Here m is the number of insertions and queries. So it is important to know whether there exist algorithms that actually run in this time. For $k=2$, Westbrook and Tarjan [32] obtained the optimal running time of $O(n+m.\alpha(m,n))$. For 3-edge-connectivity, however, only combinatorial and special-case results exist. In our companion paper [21], we developed combinatorial and special-case results for 2- and 3-edge-connectivity; we will further use these in this paper. These special-case results concern maintaining the 3-edge-connectivity relation in 2-edge-connected graphs and give an implementation in $O((n+m).\alpha(m,n))$ time. In [14], Galil and Italiano obtained comparable results with this complexity for maintaining the 3-edge-connectivity relation in connected graphs.

*Received by the editors October 27, 1993; accepted for publication (in revised form) October 4, 1996; published electronically March 15, 2000. This work was done when the author was a researcher at Utrecht University and Leiden University, The Netherlands. The work was partially supported by the ESPRIT Basic Research Actions 3075 and 7141 of the EC, and partially by the Royal Netherlands Academy of Sciences and Arts (KNAW). Earlier versions of this paper occurred in Tech. report RUU-CS-90-27, Utrecht University, 1990; *Dynamic Graph Algorithms and Data Structures*, Ph.D. thesis, Utrecht University, The Netherlands, 1991.

http://www.siam.org/journals/sicomp/29-5/25777.html

†CWI (Centre for Mathematics and Computer Science), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (hlp@cwi.nl).

[1]In this paper, n is the number of nodes.

[2]For definitions, see subsection 2.5.

[3]$\alpha(m,n)$ is the inverse Ackermann function.

Although [21] presents a solution for 3-edge-connectivity which works in time $O(n \log n + m)$, it still leaves the problem whether the α-bound is achievable for the general case. The general problem appears to be substantially more difficult than the special-case problems, which can make use of the preset combinatorial structure of the special graphs. This is also sustained by a coresult of this paper, which shows that maintaining 2-edge-connected components in connected graphs can be done in linear time, while in general graphs this cannot. Therefore, the important issue remains whether the (actually more involved) 3-edge-connectivity relation can be maintained within the α-bound.

The main objectives of this paper are presenting algorithms and data structures that maintain the 3-edge-connectivity relation in general graphs with a running time of $O(n + m.\alpha(m, n))$. To achieve this, we develop extended combinatorial structures (augmented cycle forests and basic cluster trees), and we present new data structures (fractionally rooted trees). We thus construct a solution consisting of different data structure layers to maintain the 3-edge-connectivity relation. For practical applications, however, the structures seem very well suited for implementation. Furthermore, we also present a linear-time solution for maintaining 2-edge-connected components in connected graphs. Since there is a nonlinear lower bound of $\Omega(n + m.\alpha(m, n))$ for maintaining 2-*vertex*-connectivity in connected graphs, this seems to be the first result that reveals a difference in computational complexity between 2-edge- and 2-vertex-connectivity. Finally, we also give new solutions[4] for maintaining the 2-edge- and 2-vertex-connected components of a graph, which also makes use of the above data structures and with a similar running time of $O(n + m.\alpha(m, n))$. This integrates the approaches for 2- and 3-edge-connectivity and also connects those for 2- and 3-vertex-connectivity (see [24]). We remark that all our results allow the insertion of nodes as well.

The paper is organized as follows. Section 2 contains the preliminaries. In section 3, the specifications of the operations on a new data structure, called fractionally rooted trees, are given. In section 4, the maintenance of 2-edge-connected components is considered, including the special case for connected graphs. In section 5–7, the fractionally rooted tree is presented. To be precise, observations and ideas are given in section 5; the building elements for fractionally rooted trees, called division trees, are described in section 6; and the fractionally rooted trees themselves are presented in section 7. Their complexity is considered in section 8. The final results for fractionally rooted trees are in section 9. In section 10, the optimal solution for maintaining the 3-edge-connected components is presented. Furthermore, in section 11, the maintenance of 2-vertex-connected components is briefly considered. Finally, section 12 contains concluding remarks. Readers interested in the main outlines of the paper can skip sections 6.2, 8.1, and 9 (except for the theorems in section 9). Readers interested in 3-edge-connectivity only can skip subsection 4.2.

2. Preliminaries.

2.1. Graphs and terminology. Let $G = \langle V, E \rangle$ be an undirected graph with V the set of vertices and E the set of edges. We denote an edge as a triple (e, x, y), where e is a unique edge name and x and y are the end nodes of the edge. A graph is called *empty* if it consists of nodes without edges. We use the standard terminology (see also [15]). A path is *simple* if no node occurs twice in it. Two paths are called *edge disjoint* if they do not have a common edge. Two (different) paths are called

[4]Obtained independently from [32].

vertex disjoint if they do not have a common vertex except for their end vertices. Two nodes are called *connected* if there exists a path between them. An (elementary) *cycle* is a path of which the end nodes are equal and in which no edge occurs twice. A cycle is *simple* if no node occurs twice except for the end nodes.

We extend the terminology. Consider a tree T. A set of nodes of T *induces a subtree* of T if these nodes are the nodes of a subtree of T; this is similar for a set of edges. Suppose the vertex set of T is partitioned into disjoint subsets, where each set induces a subtree of T. Let each induced subtree of T be contracted to a new node, called *contraction node*. For an edge (e,x,y), where x and y are contracted to p and q, $p \neq q$, the edge (e,p,q) is called the *contraction (edge)* of (e,x,y), and (e,x,y) is called the *original* of (e,p,q). (Both edges are given the same name.)

The tree CT consisting of the contraction nodes and the contraction edges is called *a contraction tree of* T. For a class D of edges in T, the class of edges in CT *inherited from* D consists of the contractions of edges in D. When we consider classes of nodes in a graph, we often refer to a class that is represented by a node c by "class c." A *singleton* class, set, or tree is a class, set, or tree that consists of one element or node, respectively. For a set or list L, $|L|$ denotes the number of elements in L. (If a sublist is attached to each element in L, then these sublists are not considered for $|L|$.)

Consider a tree T that is rooted at node r. (This just means that node r is a distinguished node.) The *father node of an edge* is the end node of the edge that is closest to the root. Then *father edge of a node* x is the edge between x and the father node of x. The *father edge of an edge* is the father edge of the father node of that edge. For a subtree S of T, the *maximal* node of that subtree is the (unique) node that is nearest to the root. We call an edge of subtree S a *maximal edge* if it is incident with the maximal node of S.

2.2. Connectivity. Two nodes x and y are *k-edge-connected* ($k \geq 1$) iff there exist k edge-disjoint paths between x and y, and x and y are *k-vertex-connected* iff there exist k different vertex-disjoint paths between x and y (Menger; see [26]). It is well known that k-edge-connectivity is an equivalence relation on the set of nodes of a graph.

Henceforth, we will usually call an equivalence class for 2-edge-connectivity a 2*ec-class*, and an equivalence class for 3-edge-connectivity a 3*ec-class*. The 2-edge-connected components of a graph $G = \langle V, E\rangle$ are the subgraphs of G that are induced by the 2ec-classes, i.e., subgraph $\langle C, \{(e,x,y) \in E | x,y \in C\}\rangle$ for each 2ec-class C.

Lemma 2.1 (see [21]). *Let $G = \langle V, E\rangle$ be a graph. Let H be a 2-edge-connected component of G. Then H is a 2-edge-connected graph. Moreover, nodes $x, y \in H$ are k-edge-connected in H iff they are k-edge-connected in G ($k \geq 1$).*

The notion of a 3-edge-connected component can be defined such that Lemma 2.1 holds for 3-edge-connectivity too. We refer to [21]. In our observations, we will represent the 2ec-classes and the 3ec-classes of a graph by means of a "super" graph. To this end, we use the notion of a *class node*, which is a new node (or "name") that represents a class.

Lemma 2.2 (see [21]). *Let $G = \langle V, E\rangle$ be a graph and let $k \geq 1$. Let V be partitioned into classes, where any two nodes in the same class are k-edge-connected. Let a new class node be related to each class. Let k' satisfy $1 \leq k' \leq k$. Then two nodes are k'-edge-connected in G iff the class nodes of their classes are k'-edge-connected in the graph obtained from G by contracting each class to its class node.*

We call a set S of at least two nodes a 2*vc-class* if the nodes are 2-vertex-connected, and if there does not exist a node outside S that is 2-vertex-connected with the nodes of S (i.e., the class is maximal). Furthermore we define a *quasi class* to be any set of two nodes that are the end nodes of a cut edge. The 2-*vertex-connected components* of a graph G are the subgraphs of G that are induced by the 2vc-classes of nodes. (Note that the 2-vertex-connected components and the subgraphs induced by quasi classes as we defined them are usually called the blocks of a graph.)

In the sequel, we will often denote 2-edge-connectivity by "2ec-," etc., when we consider components or relations. For example, 3ec-components denotes 3-edge-connected components, and 2vc-relation denotes 2-vertex-connectivity relation.

2.3. Problem description. The problems that we consider in this paper are as follows. Let a graph be given. Then the following operations may be applied on the graph.

insert$((e,x,y))$. Insert the edge (e,x,y) in the graph.

2*ec-comp*(x). Output the name of the 2ec-component (2ec-class) which contains x.

3*ec-comp*(x). Output the name of the 3ec-component (3ec-class) which contains x.

*Is*2*vc*(x,y). Output whether x and y are two nodes in the graph that are 2-vertex-connected and output the name of the 2vc-component (2vc-class) in which they both are contained (if any).

We call a problem the 2*ec-problem* if operations *insert* and 2*ec-comp* are considered; the 3*ec-problem* if operations *insert*, 2*ec-comp*, and 3*ec-comp* are considered; and the 2*vc-problem* if operations *insert* and $Is2vc$ are considered. In these problems, we normally start with an empty graph with n nodes (unless stated otherwise). In addition, the above collection of operations can be extended with the insertion of a new (isolated) node in the graph. We will consider this operation only in the last steps of our solutions.

We call the insertion of an edge an *essential insertion* for a given problem, if in the graph either the connectivity relation changes or, for the 2ec-problem, the 2ec-relation changes, or, for the 3ec-problem, the 2ec- or 3ec-relation changes, or, for the 2vc-problem, the 2vc-relation changes. An insertion is called *nonessential* otherwise. Note that nonessential insertions can be omitted, which is known after a proper couple of queries. (Thus such an insertion does not need to take more than the time for those queries.)

2.4. The Ackermann function. The *Ackermann function* A is defined as follows. For $i, x \geq 0$ function A is given by

$$\begin{array}{rcll} A(0,x) & = & 2x & \text{for } x \geq 0, \\ A(i,0) & = & 1 & \text{for } i \geq 1, \\ A(i,x) & = & A(i-1, A(i,x-1)) & \text{for } i \geq 1,\ x \geq 1. \end{array} \tag{1}$$

The *row inverse* a of A and the *functional inverse* α of A are defined in correspondence to [11, 12, 19, 23] by

$$a(i,n) = \min\{x \geq 0 | A(i,x) \geq n\} \quad (i \geq 0,\ n \geq 0), \tag{2}$$

$$\alpha(m,n) = \min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\} \quad (m \geq 0,\ n \geq 1). \tag{3}$$

Here we take $\lceil 0 \rceil = 1$. For more technical insight on these functions, we refer to [19]. Here we quote that

$$a(i, A(i,x)) = x \quad (i \geq 0,\ x \geq 0), \tag{4}$$

and also that for *any* practical n, we have $\alpha(m,n) \leq 3$. Also, $A(i,1) = 2$ and $A(i,2) = 4$ $(i \geq 0)$, and $A(0,x) = 2x$, $A(1,x) = 2^x$, and $A(2,x) = \left.2^{2^{2^{\cdot^{\cdot^{2}}}}}\right\}^{x \text{ 2s}}$.
Similarly we have

$$\begin{array}{rclcl}
a(0,n) & = & \lceil \frac{n}{2} \rceil, & & \\
a(1,n) & = & \lceil \log n \rceil & = & \min\{j | \lceil \frac{n}{2^j} \rceil = 1\}, \\
a(2,n) & = & \log^* n & = & \min\{j | \lceil \log^{(j)} n \rceil = 1\}, \\
a(3,n) & = & & & \min\{j | \log^{*^{(j)}} n = 1\},
\end{array}$$

where the superscript (j) denotes the j consecutive applications. For simplicity, we extend the Ackermann function by $A(i,-1) = 0$ for all $i \geq 0$.

2.5. Representation and data structures. The algorithms and data structures that we present (except for the algorithm in subsection 4.2) can be implemented on *both* a pointer machine and a random access machine (RAM) with the same complexity. Nodes and edges of a graph are represented in memory by records, which we will consider to be the actual nodes and edges. Each vertex has an incidence list consisting of pointers to the incident edges. Also, each edge contains pointers to its two end nodes. If we consider a tree T rooted at some node r, then for each node in T, its father node and father edge are related to it by appropriate pointers. An edge that has to be inserted is given by its record with the pointers to its end nodes as input for the algorithms.

In the following, the Union-Find structure is used to maintain the equivalence classes for connectivity, 2-edge-, and 3-edge-connectivity. These structures are denoted by UF_c, UF_{2ec}, and UF_{3ec}, respectively, where the corresponding Finds on elements x are denoted by $c(x)$, $2ec(x)$, and $3ec(x)$, respectively. Many solutions have been proposed for the Union-Find problem [19, 30, 31]: these solutions all take $O(n + m.\alpha(m,n))$ time for all Unions and m Finds on n elements, which is optimal [10, 20]. The solution of [19] ensures that, in addition, the fth Find can be done in $O(\alpha(f,n))$ worst-case time. We call such structures *α-UF structures.* In this paper, we will also make use of a class of structures UF(i) ($i \geq 1$), as defined in [19].

THEOREM 2.3 (see [19]). *Structure UF(i) takes $O(n.a(i,n))$ time for all Unions on n elements, where a Find takes $O(i)$ worst-case time $(i \geq 1)$.*

We consider the *connectivity problem* for edge insertions. Let $G = \langle V, E \rangle$ be a graph. Suppose a sequence of edge insertions in G and queries about whether two nodes are connected are performed. If an edge (e,x,y) is inserted, there are two cases. If $c(x) = c(y)$, then nothing needs to be done. Otherwise, if $c(x) \neq c(y)$, then x and y are not connected yet and the equivalence classes $c(x)$ and $c(y)$ are joined. Since apart from these Unions, each insertion takes $O(1)$ time, it follows that all insertions and queries can be performed in $O(|E|)$ time plus the time needed for the Union and Find operations. In the sequel, we use this algorithm for maintaining connectivity, but we will not make the above computations explicit any more.

3. Fractionally rooted trees: Concept and operations. We give a formal description of the operations supported by the data structure called *fractionally rooted tree*, without considering the data structure itself yet. Let a forest F be given. Suppose the collection of edges is partitioned into disjoint classes such that each class induces some subtree of F. Such a partition is called an *admissible partition.*

We first define some notions. Let x and y be two nodes in the same tree of F, and let P be the tree path between x and y. By "edge classes on P," we mean the edge classes of which an edge is on P. An edge class is *incident* with node x if it contains an edge with x as end node. We call a node x on P a *boundary node* of P if it is incident with two classes on P or if it is one of the end nodes of P. We call a node of P an *internal node* otherwise. A *boundary edge set* for a boundary node z on P is a set of (0, 1, or 2) edges incident with z: one from each class that is incident with z and that is on P. (See Figure 1, where path P is drawn with heavy lines, C_1 and C_2 are two different edge classes, $\{e_1, e_2\} \subseteq C_1$ and $\{e_3, e_4\} \subseteq C_2$, and where $\{e_1, e_3\}$, $\{e_1, e_4\}$, $\{e_2, e_3\}$, and $\{e_2, e_4\}$ are the possible boundary edge sets for z on P.) A *boundary list* for the two nodes x and y is a list consisting of the boundary nodes of P, where each boundary node has a sublist that contains a boundary edge set for it on P. (Note that in a boundary list for x and y with $x \neq y$, all nodes have a sublist with two edges except for nodes x and y that each have one edge in their sublist.) We say that x and y are *related nodes*, denoted by $x \sim y$, if $x = y$ or if all the edges on P are in the same edge class. (Hence $x \sim y$ iff x and y are the only nodes in a boundary list for x and y.)

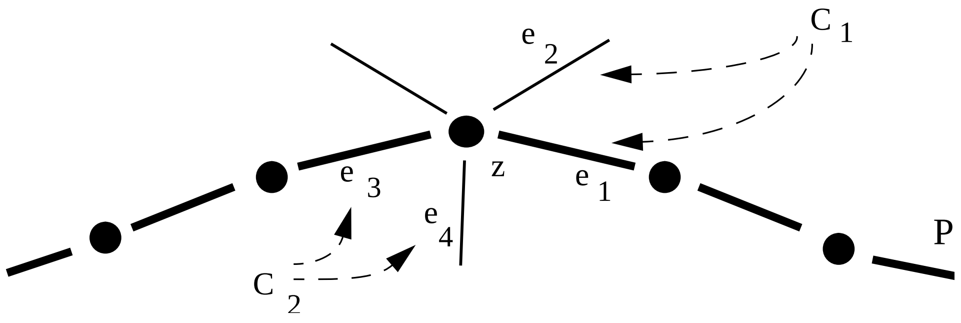


FIG. 1. *Boundary edge sets.*

We say that an edge class *occurs* in a list consisting of sublists of edges if an edge of it occurs in some sublist. A *joining list* J is a list of nodes with sublists of edges such that the union of the classes occurring in J induces some subtree in F. (Hence it yields a new admissible partition of the edge set.) In addition, the nodes in J must be the nodes incident with at least two classes occurring in J, and the sublist for each node contains an edge for each class in J incident with the node.

The following operations, called *FRT operations*, may be performed on a forest F.

link$((e, x, y))$. Let x and y be nodes in different trees of forest F. Then link the two trees containing x and y by inserting the edge (e, x, y), where (e, x, y) forms a new singleton class.

boundary(x, y). Let x and y be in the same tree of F with $x \neq y$. Then output a boundary list for x and y.

joinclasses(J). Let J be a joining list. Then join all the edge classes of which an edge occurs in the list.

candidates(x, y). Return an edge incident with x and return an edge incident with y; these edges (the candidates) are in the same class if such edges exist (i.e., $x \sim y$). Return the names of the edge classes in which the edges are contained.

Finally, we define some notions that will be used in the sequel. We say that a call *boundary*(x, y) is *essential* if $\neg(x \sim y)$ and it is *nonessential* if $x \sim y$. An *essential sequence* is a sequence of *link*, *boundary*, and *joinclasses*, where every call of *boundary* is essential and is followed by a call *joinclasses*(J) such that all the edge classes occurring in the output of *boundary* also occur in J. A *matching sequence* is a sequence of FRT-operations where the subsequence of calls of *link*, essential calls of *boundary*, and calls of *joinclasses* forms an essential sequence.

4. Two-edge-connectivity.

In this section, we consider the problem of maintaining the 2ec-components in a graph, and we will present algorithms that run in $O(n + m.\alpha(m, n))$ time for n nodes and m queries and insertions using fractionally rooted trees. Thus we present a solution that is different from that given in [32] but whose approach is closer to the approach for maintaining the 3ec-relation in general graphs (section 10), and that we will use there too. We also present a linear-time solution for maintaining the 2ec-components in case the initial graph is connected.

4.1. Graph observations.

In this subsection, we recall from the companion paper [21] the observations for inserting edges in a graph $G = \langle V, E\rangle$. The set V can be partitioned into equivalence classes for 2-edge-connectivity: the 2ec-classes. Let each 2ec-class C be represented by a new (distinct) node c, called the *class node* of C. Let $2ec(x)$ be the class node of the 2ec-class in which the node x is contained. We define the contracted graph $2ec(G)$ as follows:

$$2ec(G) = \langle 2ec(V), \{(e, 2ec(x), 2ec(y)) | (e, x, y) \in E \wedge 2ec(x) \neq 2ec(y)\}\rangle.$$

For example, $2ec(G)$ is the graph that is obtained if we contract each 2ec-class into one class node. By Lemma 2.2, $2ec(G)$ is a forest (for a figure, see [21]). An edge (e, x, y) in G is called an *interconnection edge* between (classes) $2ec(x)$ and $2ec(y)$ if $2ec(x) \neq 2ec(y)$.

We consider the 2ec-relation under edge insertions by means of the graph $2ec(G)$. Suppose a new edge $(e, x, y) \notin E$ is inserted in graph $G = \langle V, E\rangle$. We distinguish three cases and apply Lemma 2.2.

1. $c(x) \neq c(y)$. Then $(e, 2ec(x), 2ec(y))$ connects two trees in $2ec(G)$ that have to be joined into one tree.
2. $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. Then edge $(e, 2ec(x), 2ec(y))$ connects $2ec(x)$ and $2ec(y)$ in a tree of $2ec(G)$, and all class nodes on the tree path P from $2ec(x)$ to $2ec(y)$ become 2-edge-connected in $2ec(G)$. Thus all the classes "on" P must be joined.
3. $2ec(x) = 2ec(y)$. Then nothing happens.

4.2. Algorithms for initially connected graphs.

We consider the 2ec-problem in case the initial graph is connected. We represent the graph $2ec(G)$ by means of a spanning tree of G, denoted by $ST(G)$. Note that a 2ec-class induces a subtree in $ST(G)$. Since the tree $ST(G)$ can be constructed in advance, we can use the Union-Find algorithms of [13] to maintain the 2-edge-connected classes: this algorithm runs in $O(n + m)$ time for m Finds for this special case. (It runs on a RAM but not on a pointer machine.) Moreover, as remarked in [19], a Find can be performed in $O(1)$ worst-case time.

We give the algorithms in case the graph G initially is a tree. We implement the tree as a rooted tree and initialize the Union-Find structure of [13] accordingly. We recall from [13] that the name of a set in the Union-Find structure is the (unique) node in the set that is closest to the root. Suppose an edge (e, x, y) is inserted. If $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$, then the tree path between $2ec(x)$ and $2ec(y)$ is obtained as in [21] by traversing the root paths of $2ec(x)$ and $2ec(y)$ in $2ec(G)$ stepwise in an alternating way, where we use $ST(G)$ with the Union-Find structure as representation for $2ec(G)$. This is stopped if a class name *top* has been visited by both traversals; path P between $2ec(x)$ and $2ec(x)$ consists of the two parts of these root paths up to and including *top*.

We consider the time complexity. A computation of a tree path P is done in $O(|P|)$ time, since one of the two traversals contains nodes of P only. Since the number of classes decreases by $|P| - 1$, all path computations take $O(n)$ time altogether. All Unions and m Finds take $O(n+m)$ time. Finally, each insertion takes two Finds and $O(1)$ time, apart from the above cost.

In case the initial graph is connected but it is not a tree, then we do the following. First obtain a spanning tree of the graph, and initialize the structure for this tree. Then insert the edges of the graph that are not in the tree by means of the above algorithm. Then the actual insertions can be performed.

THEOREM 4.1. *The* $2ec$*-problem for graphs that are initially connected can be solved such that a sequence of* m *insert operations takes* $O(n + m)$ *time, where a query takes* $O(1)$ *time. The structure can be initialized* $O(e_0)$ *time and takes* $O(n)$ *space, where* e_0 *is the number of edges in the initial graph.*

The above theorem can be augmented to allow attachment of a single new node by an edge connecting it with an existing node in the graph, within the same time complexity. (Thus, the graph remains connected.) This can be done by [13, section 3].

4.3. Algorithms and data structures for general graphs. In this subsection, we will give a solution for the general 2ec-problem with a time complexity of $O(n + m.\alpha(m, n))$ for n nodes and m queries and insertions.

We represent the structure $2ec(G)$ by means of a forest of spanning trees of G. We denote the forest together with additional information (defined below) by $SF(G)$. $SF(G)$ is augmented with *edge classes* induced by the 2ec-relation.

Let (e, x, y) be an edge in $SF(G)$. If $2ec(x) = 2ec(y)$, then (e, x, y) is in the edge class named $2ec(x)$. Otherwise, edge (e, x, y) forms a singleton class on its own, which we call a *quasi class*.

An edge class that is not a quasi class is called a *real class*. Note that interconnection edges form quasi classes and vice versa.

As observed in subsection 4.2, a 2ec-class (of nodes) induces some subtree in $SF(G)$. Therefore, each edge class induces a subtree in $SF(G)$. Also, if each subtree in $SF(G)$ induced by a real edge class is contracted to some node, then we obtain the forest $2ec(G)$, where the quasi edge classes in $SF(G)$ correspond to the edges in $2ec(G)$.

We consider the insertion of edge (e, x, y). If x and y are in different trees of $SF(G)$, then these trees need to be linked. Now suppose x and y are in the same tree T of $SF(G)$. Let P be the tree path in T between x and y. We use the terminology of section 3. By the definition of edge classes, a boundary node of P is either one of the end nodes x or y, or it is a node for which its two neighbors on P are not both in the same 2ec-class as itself. The two neighbors of an internal node z on P are inside class $2ec(z)$ too. Therefore, if we compute the boundary nodes of P only, then we obtain one or two nodes of each 2ec-class (of nodes) that need to be joined.

We need some tree representation to compute boundary sequences efficiently while trees are linked from time to time. One solution is to use rooted trees and, in the case of linkings of trees, to redirect the smallest one of the two trees that are linked.

However, this takes $O(n.\log n)$ for the linkings. To improve the time complexity, we use the fractionally rooted trees (FRT) structure.

We solve the 2ec-problem by the so-called 2EC structure, which is given as follows. We use the above forest $SF(G)$ with the 2ec-classes and the above edge classes. A node x in $SF(G)$ that is not in a singleton 2ec-class has a pointer *assoc* to an edge that is incident with x and that is in the class named $2ec(x)$. (Such an edge exists.) We call such an edge an *associated edge* for x. Forest $SF(G)$ is implemented as a FRT structure, denoted by FRT_{2ec}. Moreover, all 2ec-classes of nodes (in $SF(G)$) are implemented by a Union-Find structure, denoted by UF_{2ec}. All connected components of nodes are implemented by a Union-Find structure, denoted by UF_c.

The initialization and the queries are straightforward. The insertion of edge (e, x, y) in graph G is done by procedure $insert_{2ec}((e, x, y))$ as follows.

1. If $c(x) \neq c(y)$, then $link((e, x, y))$ is performed, and the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).
2. If $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$, then the following is done. First, operation $boundary(x, y)$ is performed, returning boundary list BL. All the (node) classes in which the boundary nodes are contained are joined in UF_{2ec}. For each node z in BL, the associated edge e of z (if any) is obtained. If edge e is not in an edge class occurring in the sublist of z in BL, then e is inserted in its sublist. (This ensures that $2ec(e) = 2ec(z)$ remains true after subsequent joinings.) The end nodes of BL are removed in case their sublists contain one edge only. Then, if $BL \neq \emptyset$, operation $joinclasses(BL)$ is performed. Finally, for each node z in BL without an associated edge, an edge in its (old) sublist is made its associated edge.
3. If $2ec(x) = 2ec(y)$, nothing is done.

Note that starting from a graph with n nodes, there are at most $2(n-1)$ essential insertions, since in each essential insertion at least two connected components or 2ec-classes are joined.

OBSERVATION 4.2. *In a 2EC structure, the time needed for a sequence of essential insertions is linear to the time for a matching sequence of $O(n)$ operations on n nodes in FRT_{2ec} and for $O(n)$ Unions and Finds in UF_c and UF_{2ec}. Each nonessential insertion takes time linear to $\theta(1)$ Finds in UF_c and UF_{2ec}.*

A $2EC(i)$ structure is the structure described above, where $FRT_{2ec} = FRT(i)$ (see section 9), $UF_{2ec} = UF(i)$, and $UF_c = UF(i)$.

THEOREM 4.3. *A $2EC(i)$ structure solves the 2ec-problem such that the total time for all essential insertions is $O(n.i.a(i, n))$, where a query and a nonessential insertion can be performed in $O(i)$ time, and where the data structure can be initialized in $O(n)$ time and takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. Theorem 4.3 can be proved by Observation 4.2, Theorem 2.3, and Theorem 9.1. ▯

We denote the Union-Find structures UF_{2ec} and UF_c together by UF. We consider the UF structures to be one structure on $O(n)$ elements. Now take FRT$(\alpha(n, n))$ as FRT_{2ec} for a graph with n nodes, where $\alpha(n, n)$ is obtained as in [19], and take for UF the α-UF structure (see subsection 2.5). Then we obtain the following.

THEOREM 4.4. *The 2ec-problem can be solved such that the time is $O(m.\alpha(m, n))$ in total (where m is the number of edge insertions and queries), where the fth query takes $O(\alpha(f, n))$ time. The data structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

Proof. Each query and nonessential insertion corresponds to $O(1)$ Finds in the

UF structures. Moreover, all essential insertions take at most $O(n)$ Finds. Hence by [19] the fth operation is performed in $O(\alpha(f,n))$ time if it is a query or nonessential insertion. The remaining statements follow by Theorem 9.1 (with $n' \leq \min\{2m, n\}$), by (3), and by subsection 2.5. $\square$

The above theorem can be augmented to allow insertion of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$: then α-FRT is used instead of FRT$(\alpha(n,n))$ (cf. section 9).

5. Fractionally rooted trees: Observations and ideas. We give some of the ideas and observations regarding *fractionally rooted trees*. We consider a forest F, with an admissible partition of the edge set (see section 3).

A tree T in F is partitioned into subtrees that all are (locally) rooted, i.e.; each subtree has its own root independent of the remainder of the tree and subtrees. Each subtree is contracted to a new node, which yields a contracted tree T'. The collection of edges of T' is partitioned into edge classes *inherited from* the edge classes of T.

A boundary list B between two nodes x and y in T can now be obtained as follows. Let c and d be the nodes in T' to which x and y are contracted, respectively. Suppose $c \neq d$. Let P be the tree path between x and y in T. Let P' be the tree path between c and d in T'. Since an edge class induces a subtree, it follows that each boundary node of P' contains a boundary node of P, and the other way around. For a boundary node b on P', let P_b be the part of P inside b, and let s and t be its end nodes. Then, obviously, for a node $z \notin \{s,t\}$ contained in b, z is a boundary node of P_b iff it is one of P. If we extend P_b to P_{bb} with the other edges e_s and e_t on P incident with s and t, respectively (if they exist), then it follows that *a node contained in b is a boundary node of P_{bb} iff it is one of P.*

Now suppose that e_s exists. Then the boundary set for b contains an edge that is in the same edge class as the contraction of e_s. Let f_s be the original of this edge. Then e_s and f_s are in the same class, and, hence, the tree path connecting them consists of edges in this class only. Therefore, if we change P_b by replacing the "end edge" e_s by f_s, the boundary nodes contained inside b remain unchanged. We can do the same for t. Hence, *the boundary nodes contained in b are those contained in the local tree path between the originals of the edges in the boundary edge set of b or x or y (if x or y are contained in b).* (See Figure 2 for an illustration within T, where the subtree of T that is contracted to b in T' is surrounded by an ellipsoid.)

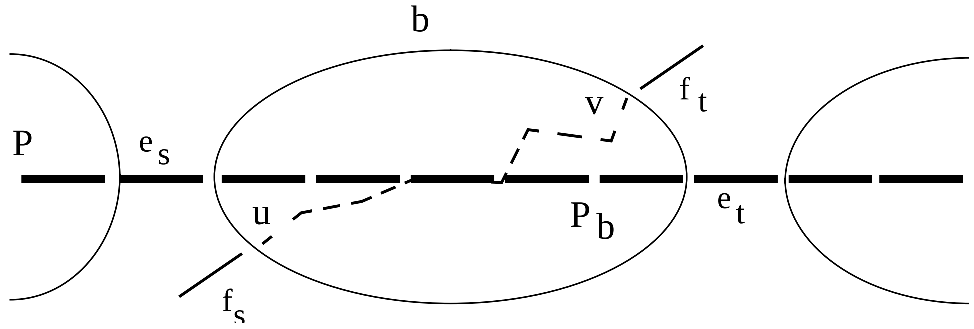


FIG. 2. *Boundary nodes in b.*

Hence we can *compute* a *boundary list B* for x and y as follows. First we compute a boundary list B' in T' for the nodes c and d. Then for each boundary node b in B', we obtain the nodes u and v in b that are x, y, or end nodes of the originals of the edges in the boundary edge set of b. Subsequently, we compute the "local" boundary list $bl(b)$ for u and v. Finally, we extend the sublists of the end nodes u and v with the appropriate originals of the edges in the boundary edge set of b; if u (or v) is not a boundary node for x and y after all, then it is removed from $bl(b)$. Then these local boundary lists $bl(b)$ together form B.

6. Division trees.

6.1. Description. Division trees form the base of the fractionally rooted trees. For the terminology regarding contractions we refer to section 2.1.

Let F be a forest with an admissible partition of the edge set into edge classes, distinguished as *global edge classes*. Let T be a tree in F. Let $CT(T)$ be a contraction tree of T, where $CN(T)$ is the collection of contraction nodes and for each $b \in CN(T)$, $tree(b)$ is the subtree of T that is contracted to b. Then T together with set $CN(T)$ and with subtrees $tree(b)$ is called a *division tree*. An edge is called *internal* if it is contained in some $tree(b)$, and *external* otherwise. For a contraction node b, the *extended tree* $extree(b)$ is $tree(b)$ extended with the external edges incident with $tree(b)$. The edge set of a $extree(b)$ is partitioned into *local edge classes* induced by the global edge classes of T. This yields an admissible partition. Tree $extree(b)$ is rooted at some node.

An external edge (e, x, y) may contain different information pertaining to the two extended subtrees in which it is contained. Therefore, we distinguish two representatives called *(edge) sides*, one for each of its end nodes $(e, x, y)_x$ and $(e, x, y)_y$. For external edge (e, x, y), $(e, x, y)_x$ is the representative for $extree(contr(x))$, where $contr(x)$ is the node to which x is contracted. For internal edges, both sides are considered to be identical. We often omit referring to the proper side, however.

The class of edge e in $extree(b)$ is denoted by $class(e)$. Every edge class contains at most one edge that is marked by a so-called *d-mark*, which must be an external edge and which contains a direct pointer $d(e)$ to the name of the edge class. (So, the class name can be obtained fast.) For each edge class C in $extree(b)$, the following edges are distinguished (with direct pointers to them):

- $\max(C)$ is a maximal edge of C in the rooted tree $extree(b)$. Such an edge is then called *the* maximal edge of that class and is marked by an *m-mark* (which is done implicitly).
- $ext(C)$ is an external edge in it (if there exists any).
- $direct(C)$ is the d-marked edge in it (if it exists).

For a node x in $extree(b)$, the father edge of x or an m-marked edge incident with x is called a *preferred edge for x*. Note that for node x and a class C incident with x, there is *exactly one* preferred edge for x in C.

We describe the operations that we want to perform on F.

basic-external-link$((e, x, y))$. Let x and y be nodes in two different trees T_x and T_y. Then link these trees by the edge (e, x, y), yielding tree T, where the partition of the node set remains unchanged. This means that $CN(T) = CN(T_x) \cup CN(T_y)$, and for each $b \in CN(T)$, $tree(b)$ is not affected by the operation. The new edge (e, x, y) forms a new singleton class on its own.

basic-internal-link$((e, x, y), y)$. Let x and y be nodes in two different trees T_x and T_y. Let $c = contr(x)$. Then link these trees by the edge (e, x, y), yielding tree T, where $tree(c)$ is extended with edge (e, x, y) and tree T_y. For example, $CN(T) = CN(T_x)$, and $tree(b)$ remain unchanged for $b \in CN(T_x)\backslash\{c\}$. The new edge (e, x, y) forms a new singleton class on its own. The edges in T_y are called *affected*.

basic-integrate(x, f). Let x be a node in tree T, and let f be a (possibly new) contraction node not occurring in $CN(T)$. Then change the partition of T such that it consists of just one subtree, with contraction node f. For example, afterward $CN(T) = \{f\}$. The edges in T are called *affected.*

basic-boundary(x, y). Let $contr(x) = contr(y)$. Then return a boundary list BL for x and y, where the edges in BL are preferred.

basic-joinclasses(J). Let J be a joining list with one node such that there is at most one edge class occurring in J that contains a d-marked edge. Then join the edge classes occurring in J.

Note that for affected edges, the father relations and m-marks of these edges (edge sides) may change during these calls.

6.2. Implementation. We implement the structures as follows. A tree T in F is implemented in the common way. Each node x in T contains a pointer $contr(x)$ to the contraction node in which it is contained, and, conversely, for each contraction node b, the list $nodes(b)$ consists of the nodes in $tree(b)$. Similarly, each external edge has a pointer to its contraction edge and vice versa. An edge is marked external or internal. The edge classes in $extree(b)$ are represented by a Union-Find structure, called the *local class Union-Find structure.* The initialization of a division tree with one contraction node is straightforward.

The operations are implemented as follows. We omit straightforward implementation details regarding, e.g., handling marks, (special) pointers, lists, etc. Note that converting an edge from external to internal may have consequences for classes, marks, and pointers.

basic-external-link$((e, x, y))$ *and basic-integrate*(x, f). The implementation of these operations is obvious. Note that maximal edges can be found by checking for each edge whether its father edge is in the same class.

basic-internal-link$((e, x, y), y)$. Let $c = contr(x)$. First, *basic-integrate*(y, c) is performed, edge (e, x, y) is inserted, and x is made the father of y. Then $\max(class(e))$ is set to (e, x, y).

basic-boundary(x, y). If $x = y$, then return the boundary list BL consisting of node x with empty sublist. Otherwise, the following is done. First, two boundary lists $s(x)$ and $s(y)$ for the root paths of x and y are stepwisely computed in an alternating way, until a node *top* has been visited by both computations. This is as follows. List $s(x)$ starts with visiting node x, and a step for $s(x)$ is as follows: obtain the father edge (e, z, z') of the node z that is visited (if any), obtain the edge $\max(class(e)) = (e', u, v)$, and visit the father node of e'. Shorten the lists $s(x)$ and $s(y)$ such that they are boundary lists for x and *top* and for y and *top* respectively. Boundary list BL is created from $s(x)$ and $s(y)$, where if $top \notin \{x, y\}$ and the two edges related to *top* are in the same edge class, then *top* is removed from the list (since it cannot be a boundary node of P).

basic-joinclasses(J). First a list CJ is created consisting of all (names of) edge classes occurring in J. Then the classes in CJ are joined, and the edges e_d, e_m, and e_{ex} (given below) are related to this new class appropriately. Edge e_m is the maximum edge of the class of the father edge of x if this class occurs in CJ, and of any class in CJ otherwise. Edge e_d is the (unique) d-marked edge in one of the classes in CJ (if it exists), and e_{ex} is the external edge of some class in CJ (if any).

7. Fractionally rooted trees: The data structure. We present the recursive data structure called the fractionally rooted trees. We consider a dynamic forest F_0 with an admissible partition of its edge set.

Let $i \geq 1$. Let F_i consist of contractions of a number of trees in F_0. The edge set of forest F_i is partitioned into the edge classes that are inherited from the edge classes of F_0 We introduce the structures FRT(i) for F_i for $i \geq 1$.

Each tree of F_i has a name in FRT(i) being some (new) unique node. For each tree in F_i, its data structure contains its tree name s and a collection of at most i layers, numbered from i in a decreasing order (say, down to $down(s)$). Each existing layer j consists of a division tree, denoted by $tree(s, j)$. For layer i, $tree(s, i)$ is the tree in F_i with name s. For existing layer $j < i$, $tree(s, j)$ is the contraction of $tree(s, j+1)$, and its global edge classes are inherited from those of $tree(s, j + 1)$. Finally, tree name s forms the contraction tree of $tree(s, down(s))$. (The above number $down(s)$ is only used in the description.) We denote by $tree_0(s)$ the original in F_0 of $tree(s, i)$ in F_i. To each tree name some parameters are associated, which will be given in the following sections. The structure FRT(i) allows the operations on F_i as described in section 3, where we add the parameter i to easily allow recursion. Thus we have (with the following modifications) the operations $link((e, x, y), s, t, i)$, where $x \in tree(s, i)$ and $y \in tree(t, i)$; $boundary(x, y, i)$, where the returned boundary list consists of *preferred* edges; $joinclasses(J, i)$; and $candidates(x, y, i)$, which does not return the names of edges classes, and where the returned edges are *preferred.* In addition, we have an operation $treename(x)$ that trivially outputs the name s of the tree in which a node x occurs.

For implementation purposes, we mention that the edge classes in F_0 are represented by a Union-Find structure UF_0. If FRT(i) is used as a complete structure, directly on F_0 (i.e., $F_i = F_0$), then $UF_0 = UF(i)$ (see subsection 2.5), and each operation $joinclasses(J, i)$ also joins all classes in F_0 occurring in J (in UF_0).

The structures FRT(i) are defined inductively (in terms of divisions trees). The method of induction has relations to those in [11, 12, 19, 23]. We start from a base structure FRT(1) that corresponds to the idea using ordinary rooted trees. This structure takes $O(n. \log n)$ time for an essential sequence of operations.

7.1. The structure FRT(1). Structure FRT(1) is a structure for a forest F_1 that satisfies the following conditions. For each tree name s, we have a parameter $weight(s, 1)$ that contains the number of nodes in $tree(s, 1)$. The local class Union-Find structure for F_1 is UF(1). FRT(1) is initialized as a forest of division trees with one contraction node each. The algorithms for the operations are as follows.

$link((e, x, y), s, t, 1)$. W.l.o.g. suppose that $weight(s, 1) \leq weight(t, 1)$. Then $basic\text{-}internal\text{-}link((e, x, y), x)$ is performed.

$boundary(x, y, 1)$. Boundary list BL is obtained by a call $basic\text{-}boundary(x, y)$.

$joinclasses(J, 1)$. The joining of classes is performed by calls $basic\text{-}joinclasses(J_x)$ for each node x in J, where J_x consists of x and its sublist in J.

$candidates(x, y, 1)$. Let e_x and e_y be the father edges of x and y, respectively (if they exist). Obtain the edges $m_x := \max(class(e_x))$ and $m_y := \max(class(e_y))$. If m_x is incident with y, then $e'_y := m_x$ (now e'_y is m-marked for y), otherwise $e'_y := e_y$; this is similar for e'_x. Output e'_x and e'_y. (Now e'_x and e'_y are preferred.)

We remark that procedure $candidates(x, y, 1)$ yields a correct pair of edges, since if x and y are incident with the same edge class C, at least one of the father edges of x and y must be in C, and if the father edge of, say, x is not in C, then the m-marked edge of C is incident with x.

7.2. The structure FRT(i) for $i > 1$. Let $i > 1$. Structure FRT(i) is a structure for a forest F_i that satisfies the following conditions. For each tree name

s, we keep a parameter $\mathit{weight}(s,i)$ that contains the number of nodes of $\mathit{tree}(s,i)$. Also, we have a parameter $\mathit{lowindex}(s,i)$ which is an integer ≥ -1 that satisfies

$$2.A(i, \mathit{lowindex}(s,i)) \leq \mathit{weight}(s,i). \tag{5}$$

(The parameter *lowindex* is incremented from time to time by the algorithms.) The Union-Find structure for local classes in F_i is UF(i).

Two cases are distinguished.

- If $\mathit{weight}(s,i) = 1$, then $\mathit{down}(s) = i$. Hence $CN(\mathit{tree}(s,1)) = \{s\}$.
- Otherwise, if $\mathit{weight}(s,i) > 1$, then $\mathit{down}(s) < i$. A contraction node $b \in CN(\mathit{tree}(s,i))$ satisfies (besides $\mid \mathit{nodes}(b) \mid \geq 2$)

  $$\mid \mathit{nodes}(b) \mid \geq 2.A(i, \mathit{lowindex}(s,i)). \tag{6}$$

  If layer i is removed, then the remaining part, starting from $\mathit{tree}(s, i-1)$ in layer $i-1$, is an FRT($i-1$)-structure. For an external edge (e,x,y) in $\mathit{tree}(s,i)$, side $(e,x,y)_x$ is *d-marked* if its contraction edge is preferred for $\mathit{contr}(x)$.

Note that every edge class C in $\mathit{extree}(b)$ for some $b \in CN(\mathit{tree}(s,i))$ contains at most one d-marked edge, since every edge class in $\mathit{tree}(s,i-1)$ contains at most one preferred edge incident with b.

7.2.1. Implementation. The initialization is done by initializing a forest of division trees with one contraction node each. For singleton trees, the contraction node is the tree name, where for nonsingleton trees, new tree names are recursively related to them in the next layer. All the corresponding *lowindex*-values are set to -1.

We give the algorithms for the operations. Note that, by (5), $\mathit{lowindex}(s,i) \geq 0$ implies that $\mathit{down}(s) < i$.

$\mathit{link}((e,x,y),s,t,i)$. W.l.o.g., we assume that $\mathit{lowindex}(s,i) \geq \mathit{lowindex}(t,i)$.

Let $\mathit{newweight} := \mathit{weight}(s,i) + \mathit{weight}(t,i)$ and let $\mathit{ls} := \mathit{lowindex}(s,i)$. There are three cases. (For more intuition behind this operation, we refer to the comments and figures in [19].)

- $\mathit{lowindex}(s,i) > \mathit{lowindex}(t,i)$. A call *basic-internal-link*$((e,x,y),y)$ is performed. (Now, $\mathit{tree}(t,i)$ is contracted to $\mathit{contr}(x)$.) Then t and its related layers j with $j < i$ are disposed.
- $\mathit{lowindex}(s,i) = \mathit{lowindex}(t,i) \wedge \mathit{newweight} \geq 2.A(i, \mathit{ls}+1)$. Then a *new* contraction node f is created in layer $i-1$. Then *basic-external-link*(e,x,y) and $\mathit{basic} - \mathit{integrate}(x,f)$ are called, and $\mathit{contr}(f) := s$. (Now, $\mathit{tree}(f)$ consists of the former $\mathit{tree}(s,i)$, $\mathit{tree}(t,i)$, and (e,x,y).) The old existing layers j related to s and t with $j < i$ are disposed, including tree name t. Finally, $\mathit{lowindex}(s,i) := \mathit{lowindex}(s,i) + 1$ and $\mathit{lowindex}(s,i-1) := -1$.
- $\mathit{lowindex}(s,i) = \mathit{lowindex}(t,i) \wedge \mathit{newweight} < 2.A(i, \mathit{ls}+1)$. Then we want to do the actual linking on a lower layer. Therefore, first *basic-external-link*$((e,x,y))$ is executed. Then the contraction edge (e,c,d) of (e,x,y) is created, and a recursive call $\mathit{link}((e,c,d),s,t,i-1)$ is performed, where all the affected edges in layer $i-1$ are obtained. For each edge (e',u,v) in layer i that is (e,x,y) or the original of an affected edge in layer $i-1$, the d-marks are updated: if its contraction edge is preferred for $\mathit{contr}(u)$, then $(e',u,v)_u$ is d-marked; otherwise, $(e',u,v)_u$ is un-d-marked. The same is done for v.

boundary(x, y, i). Perform *candidates*(x, y, i) yielding edges e_x and e_y. If e_x and e_y are in the same edge classes in F_0, then the desired boundary list consists of x and y with e_x and e_y in their sublists. Otherwise, let $c = contr(x)$ and $d = contr(y)$. Then the boundary list BB for c and d in layer $i-1$ is (recursively) computed: if $c = d$, then BB contains just c, and otherwise a recursive call *boundary*$(c, d, i-1)$ is performed, returning BB. For each boundary node b in BB, we obtain the nodes u and v in *tree*(b) that are x, y, or end nodes of the originals of the edges in the boundary edge set of b. Then a "local" boundary list for u and v in *tree*(b) is computed by *basic-boundary*(u, v), where the sublists of the end nodes u and v are extended with the originals of the (at most 2) appropriate edges in BB: if u (or v) is not a boundary node for x and y after all, then it is removed from the local list. These local boundary lists are concatenated, yielding the desired boundary list.

joinclasses(J, i). First a list JJ for layer $i-1$ is made, consisting of the nodes $contr(x)$ for nodes $x \in J$, where the sublist for c is the concatenation of all sublists for $x \in J$ with $contr(x) = c$. Then, for each node $c \in JJ$, the classes occurring in its sublist are obtained, and its sublist is replaced by a sublist that contains for each such class one external edge (if any). All nodes of JJ with a sublist containing at most one edge are removed. If $JJ \neq \emptyset$, then *joinclasses*$(JJ, i-1)$ is called. All the original edge sides of the edges that have been un-m-marked in layer $i-1$ are un-d-marked in layer i. Finally, for each node x in J, *basic-joinclasses*(J_x) is executed, where J_x contains x and its sublist in J.

candidates(x, y, i). Let $c = contr(x)$ and $d = contr(y)$. If $c = d$, then do the same as for $i = 1$. Otherwise, perform candidates$(c, d, i-1)$ that returns the (preferred) edges e_c and e_d. Let edge $e_1 \in extree(c)$ be the (d-marked) original of e_c. Let $e_2 := \max(d(e_1))$. If e_2 is incident with x, then $e_x := e_2$ (e_x is m-marked w.r.t. x); otherwise, e_x is the father edge of x. The same is done for y, yielding e_y. Return the edges e_x and e_y. This is a correct pair of edges, which follows by the specification of *candidates*$(i-1)$ and similar observations as for $i = 1$. Note that by using $d(e_1)$ instead of *class*(e_1), we need to follow *one* pointer only, instead of performing a Find.

We are left with the problem of how to obtain and store the values *weight*, *lowindex*, and the Ackermann values. All these values depend on both the tree name and the layer number. The values *lowindex*(s, j) and *weight*(s, j) for all relevant j are stored in a list for s. For further details and for the problem of how to obtain Ackermann values for all the structures (viz., by means of one "Ackermann net" for $2n$), we refer to [19].

8. Complexity of FRT(i).

We consider the time and space complexity of FRT(i) structures ($i \geq 1$). In the notation, we omit the procedure parameters except for the layer number i. Operations *treename* and *candidates*(i) can be performed in $O(i)$ time, and a nonessential call *boundary*(i) can be done in $O(i)$ time plus $O(1)$ Finds in UF_0. For *candidates*, this is seen as follows. If $contr(x) = contr(y)$, it takes one Find in UF(i), which is $O(i)$ time. Otherwise, all instructions except for the recursive call can be done in constant time (because of the d-marks and preferred edges), giving $O(i)$ time by induction. For a nonessential call *boundary*, we see that if $i = 1$ then $x \sim y$, and thus *basic-boundary*(x, y) is similar to *candidates*$(x, y, 1)$, while if $i > 1$, then *candidates*(x, y, i) is executed together with two Finds.

In the sequel, we consider the complexity of essential sequences (see section 3). We determine the time complexity in steps, where one *step* denotes a Find operation (in any involved Union-Find structure), a *candidates* operation, a nonessential *boundary* operation, or one ordinary elementary computation step not included in these three operations. Hence each *candidates* operation and each nonessential call of *boundary* takes 1 step.

We obtain the following result. The proof is given in subsection 8.1 (which can be skipped at first reading). Note that if $F_i = F_0$, then $UF_0 = UF(i)$ and the set unions take $O(n.a(i,n))$ time (see Theorem 2.3).

LEMMA 8.1. *An essential sequence in an FRT(i) structure with n nodes needs a total of $O(n.a(i,n))$ steps, except for joining edge classes in F_0 ($i \geq 1$, $n \geq 2$).*

8.1. Proof of Lemma 8.1. Lemma 8.1 is proved by induction in a way similar to the proof in [19]. We first consider the *net cost* of the basic operations, i.e., the cost of the operations *except* for the cost of set unions. Then *basic-integrate*(y, f) and *basic-internal-link*$((e, x, y), y)$ take net $O(|T_y|)$ steps, where T_y is the tree containing y; *basic-external-link*$((e, x, y))$ takes net $O(1)$ steps. A call *basic-boundary*(x, y) takes $O(|BL|)$ steps if BL is the resulting boundary list, and *basic-joinclasses*(J) takes $O(|E_J|)$ net steps, where E_J is the number of edges in J.

We now consider the complexity of the structures FRT(i). As in [19], we do not need to consider the complexity of storing and obtaining the information for each layer related to a tree name, since this can be charged easily to other operations. We show that an essential sequence in FRT(i) takes $O(n.a(i,n))$ steps on n nodes (except for the cost on F_0). Moreover, we show that the number of times that an edge becomes affected (see section 6) is at most $a(i,n)$. We prove all this by considering the procedures *link*(i), (essential) *boundary*(i), and *joinclasses*(i), where the cost of set union or essential recursive calls is considered separately. Here an *essential recursive call* is any recursive call of these procedures with the restriction that recursive *boundary* calls are essential.

8.1.1. FRT(1). We consider the cost of an essential sequence on n nodes ($n > 1$) in FRT(1) by determining for each procedure the cost of all its calls.

Procedure *link*$((e, x, y), s, t, 1)$ takes at most $O(|weight(t, 1)|)$ steps, where, w.l.o.g. *tree*(t, i) is the smallest of the two sets to be joined. Charge each node in *tree*$(t, 1)$ for $O(1)$ cost. Since the nodes in *tree*$(t, 1)$ become elements of a tree with at least double size, all calls take at most $\lfloor \log n \rfloor \leq a(1, n)$ steps together. Similarly, the number of times that an edge is affected is at most $a(1, n)$.

A call *boundary*$(x, y, 1)$ takes $O(|BL|)$ steps. Note that at least $|BL| - 1$ different classes occur in BL. Charge $O(1)$ cost to the encountered classes. In the essential sequence, all these classes are subsequently joined by a call *joinclasses*. This gives at most $O(n)$ steps.

Procedure call *joinclasses*$(J, 1)$ takes $O(1)$ steps for each class that is joined; thus the total amount of steps is $O(n)$ steps apart from the joinings.

Finally, since there are at most $2n$ edge sides, the time for set unions in UF(1) is $O(n.a(1,n))$ (Theorem 2.3).

Therefore, FRT(1) takes at most $d.n.a(1, n)$ steps for an essential sequence on n nodes ($n > 1$) for some constant d. Moreover, the number of times that a node is affected is at most $a(1, n)$.

8.1.2. FRT(i) for $i > 1$. We consider the cost for an essential sequence on n nodes ($n > 1$) in FRT(i) with $i > 1$. We perform the analysis by means of induction on i. Suppose FRT$(i - 1)$ takes at most $c.k.a(i - 1, k)$ steps in an essential sequence on k nodes ($k > 1$), where c is some arbitrary constant. Moreover, suppose that the number of times that an edge in the FRT$(i - 1)$ structure is affected is at most

$a(i-1,k)$. For each procedure or specific part of the computation, we determine the cost of all its calls.

For an essential call of $boundary(i)$, we have the following. First, the calls of $candidates(i)$ and the recursive call $boundary(i-1)$ take $O(1)$ net steps. (The call $boundary(i-1)$ takes net $O(1)$ steps if it is nonessential and it takes no steps if it is essential.) Then local boundary lists are computed and manipulated (but do not become empty, see section 5). Hence the net cost is $O(|BL|)$ steps. Since afterward all classes occurring in BL must be joined by a call of $joinclasses$ in the essential sequence, it follows that the total net amount of steps is $O(n)$.

Procedure $joinclasses(i)$ takes a net number of steps linear to the number of classes that will be joined, apart from the recursive call. Hence this is $O(n)$ in total.

We divide *link* $((e,x,y),s,t,i)$ into several parts and compute the net cost of each of these parts for all executions together. First, the removal of parts of structures can be charged to their creation. Second, the calls of procedure *basic-internal-link* and *basic-integrate* take at most $O($*the number of processed nodes*$)$ steps. Therefore, we charge these steps to the processed nodes. Note that in both cases the processed nodes will (henceforth) be contained in a new tree with higher $lowindex$ value and that there are at most $a(i,\lceil\frac{n+1}{2}\rceil)+2 \le 3.a(i,n)$ different $lowindex$ values (cf. (5)). Therefore, the total cost of these calls is $O(n.a(i,n))$ steps. Similarly, it follows that an edge is affected is at most $a(i,n)$ times. Third, the cost for changing d-marks of edges in procedure $link(i)$ is linear to the number of times that contraction edges are affected in the recursive call $link(i-1)$. In Observation 8.4 we will show that this is at most $\frac{1}{2}.n.a(i,n)$. Hence this takes $O(n.a(i,n))$ steps altogether. Last, the rest of the procedure requires $O(1)$ net time per call of link(i), which gives $O(n)$ time altogether. In conclusion, all calls of *link* take at most $O(n.a(i,n))$ steps net and affect an edge at most $a(i,n)$ times.

The required time for set unions in UF(i) is $O(n.a(i,n))$ (Theorem 2.3), since there are at most $2n$ edge sides.

Finally, we consider the essential recursive calls (performed on contraction nodes). We first have two observations (the latter can be proved as in [19]).

OBSERVATION 8.2. *The operations on contraction trees (for layer i) by procedure link*$((e,x,y),i)$ *are the creation of a singleton tree and the linking and removal of trees; procedures* $joinclasses(i)$ *and* $boundary(i)$ *only change edge classes in contraction trees.*

OBSERVATION 8.3. *In link*$((e,x,y),s,t,i)$*, a recursive call is performed only if*

$$\begin{aligned}&1 < lowindex(s,i) = lowindex(t,i) \le a(i,n)\\ &\wedge weight(s,i)+weight(t,i) < 2.A(i,lowindex(s,i)+1).\end{aligned}$$

For a contraction node $c \in CN(tree(s,i))$, we denote by $lowindex(c)$ the value $lowindex(s,i)$, which is fixed during its existence. We call c an l-contraction node if $lowindex(c) = l$. Similarly, we say that a recursive call $link((e,c,d),s,t,i-1)$ is an l-call if $l = lowindex(s,i) = lowindex(t,i)$. A recursive call $boundary(i-1)$ or $joinclasses(i-1)$ is an l-call if $l = lowindex(s,i)$, where s is the name of the tree on which the operation is applied. Obviously an l-call operates on l-contraction nodes only, and vice versa. We compute the cost of all l-calls for fixed value l, $-1 \le l \le a(i,n)$. Note that any tree of l-contraction nodes with $l \le 0$ consists of one contraction node. Hence an l-call of $boundary(i-1)$ and $joinclasses(i-1)$ occurs only if $l \ge 1$. By Observation 8.3 and since $|nodes(b)| \ge 2$ for each contraction node b, it follows in case of an l-call *link* $(s,t,i-1)$ that $l > 1$, and that $weight(s,i-1)+$

$weight(t, i-1) < A(i, l+1)$. By this and by Observation 8.2, the maximal size of any tree of l-contraction nodes is $< A(i, l+1)$.

Now let l be fixed number with $1 \leq l \leq a(i, n)$. Partition the total collection of all l-contraction nodes involved in l-calls into the existing maximal sets. Then the size of such a maximal set is at most $A(i, l+1)$. It easily follows that the sequence of essential recursive l-calls on the nodes of a maximal set in FRT$(i-1)$ is an essential sequence. For each such maximal set of k contraction nodes, the cost of all (previous) essential l-calls on these nodes in FRT$(i-1)$ is at most $c.k.a(i-1, k) \leq c.k.\ a(i-1, A(i, l+1))$. Hence the total cost of all essential l-calls in FRT$(i-1)$ on l-cluster nodes is at most $c.(\textit{number of l-cluster nodes}).\ a(i-1, A(i, l+1))$. By (6), there are at most $n/(2.A(i, l))$ l-contraction nodes. Therefore, this cost is at most $c.\frac{n}{2.A(i,l)}.\ a(i-1, A(i, l+1))$, which is at most $\frac{1}{2}c.n$ by using $i > 1$ and equations (1) (on $A(i, l+1)$) and (4), respectively.

Since at most $a(i, n)$ values l of *lowindex* occur, the cost of all these FRT$(i-1)$-calls is at most $\frac{1}{2}c.n.a(i.n)$.

Similar to the above, by the induction hypothesis, the number of times that l-contraction edges are affected in the l-calls *link*$(i-1)$ is at most $\frac{1}{2}.n$ for fixed l.

OBSERVATION 8.4. *The number of times that contraction edges are affected in recursive calls at most link$(i-1)$ is $\frac{1}{2}.n.a(i, n)$.*

Combining all the above results yields that the total number of steps is at most $c_1.n + c_2.n.a(i, n) + \frac{1}{2}c.n.a(i, n)$ for some c_1 and c_2 (independent of c). By taking $c = \max\{d, 2.(c_1 + c_2)\}$, it follows by induction that an essential sequence in FRT(i) takes at most $c.n.a(i, n)$ steps and affects an edge at most $a(i, n)$ times. This concludes the proof of Lemma 8.1.

9. FRT structures. We consider FRT(i)-structures with $F_i = F_0$ and express the operations of section 3 in terms of section 7. It is easily seen how to use the latter for the former; we may only need an additional call of *treename* or $O(1)$ Finds in UF$_0$ for the proper result. Thus Lemma 8.1 remains valid for the operations in section 3 (in order of magnitude). Note that by Theorem 2.3, a step, as defined in the previous subsection, is $O(i)$ time.

THEOREM 9.1. *An essential sequence in FRT(i) on n nodes needs a total time of $O(n.i.a(i, n))$ $(i \geq 1,\ n \geq 2)$. Each candidates operation and each nonessential call boundary takes $O(i)$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

Note that if n' is the number of nodes that are not still contained in singleton trees after the execution of the above sequence (thus $n' \leq n$), then the total time is even $O(n'.i.a(i, n'))$. Also, the theorem can be extended with the insertion of new (isolated) nodes in the structure with the same complexity bounds, where the insertion of a new node takes $O(1)$ time (see also [19]).

We define an α-FRT structure as follows. Initially, FRT$(\alpha(n, n))$ is used. From time to time, a transformation is performed, replacing an FRT(i) structure by an FRT$(i-1)$ structure, viz., each time that $\alpha(q, n)$ decreases by one, where at any moment q is the number of queries *candidates* performed until then. This is performed in a way similar to the proof of Theorem 5.2 of [19] (full paper), where now the query *candidates* plays the role of the Finds, and where *link* and *joinclasses* play the role of the Union operations. The building of a new FRT$(i-1)$ is done similar to Theorem 5.2 in [19], but instead of building just parts of FRT$(i-1)$ during *candidates* operations, we have for all pointers in F_0 two versions, and we build and handle FRT$(i-1)$ with the unused pointer version. (This duplication is only relevant in case we want a single *candidates* query to have $O(\alpha(q,n))$ worst-case time.) Then we obtain the following result.

THEOREM 9.2. *Let an α-FRT structure for an "empty" forest with n nodes be given. Then a matching sequence in α-FRT needs a total of $O((n+m).\alpha(m,n))$ time (where m is the number of operations candidates and boundary that is performed), where the qth call of candidates takes $O(\alpha(q,n))$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

Proof. The proof is related to the proof of Theorem 5.2 in [19]. We leave this as an exercise for the reader (and refer to [22, 23]). □

Note that if n' is defined as before (and, hence, the essential subsequence of the matching sequence consists of $\theta(n')$ operations), then the total time is even $O((n'+m).\alpha(m,n))$ time. Also, by using the same transformation techniques as in Theorem 6.2 in [19], the above theorem can be extended with the insertion of new (isolated) nodes in the structure with the corresponding complexity bound $O((n+m).\alpha(m,n))$ (where m and n denote the current number at the time of consideration), where the insertion of a new node takes $O(1)$ time. We will not give details here but refer to [22, 23]. (We want to remark that if at any time $m = O(n)$, as for the 2ec-and the 3ec-problem, then only rebuildings from FRT(i) to FRT($i+1$) are needed.)

In practice there is no need to perform transformations of FRT-structures or to compute Ackermann values [19]. This is because $\alpha(m,n) \leq 3$ for any practical n. Thus, structures FRT(i) with $i \in \{2,3\}$ are suited for all practical situations and only need the nontrivial Ackermann values $A(2,3) = 16$ and $A(2,4) = A(3,3) = 65536$. An essential sequence in FRT(2) takes $\leq c.2.n.a(2,n) = 2cn.\log^* n$ time, which is $\leq 8cn$ for $n \leq 2^{16}$ and $\leq 10cn$ for $n \leq 2^{65536}$, where c is not too large a constant (see section 8). Therefore, we conjecture that FRT(2) can be implemented as a fast structure for all *practical* situations, with constant-time queries.

10. 3-edge-connectivity.

We will now extend the results to the maintenance of 3ec-components in a graph, with a time complexity of $O(n + m.\alpha(m,n))$ for n nodes and m queries and insertions. In subsection 10.1 we consider maintaining the 3ec-relation within 2-edge-connected graphs and, subsequently, in subsection 10.2 we consider the problem for general graphs.

Let $G = \langle V, E\rangle$ be a graph. The set V can be partitioned into equivalence classes for 3-edge-connectivity, called *3ec-classes*. Each 3ec-class C is represented by a new node c, called the *class node* of C. Let $3ec(x)$ be the class node of the 3ec-class in which the vertex x is contained. We define the graph $3ec(G)$ as follows.

$$3ec(G) = \langle 3ec(V), \{(e, 3ec(x), 3ec(y)) | (e,x,y) \in E \wedge 3ec(x) \neq 3ec(y)\}\rangle.$$

Hence, $3ec(G)$ is the graph that is obtained if we contract each 3ec-class into one class node (see Figure 3 if G is 2-edge-connected). No two nodes in $3ec(G)$ are 3-edge-connected (by Lemma 2.2).

10.1. 2-edge-connected graphs.

In this subsection, we suppose that graph G is 2-edge-connected, and we state results from the companion paper [21]. Every two distinct class nodes must lie on a common elementary cycle in $3ec(G)$, while simple cycles in $3ec(G)$ cannot intersect in more than one class node.

Let $Cyc(3ec(G))$ be the graph that is constructed from $3ec(G)$ as follows. Each nontrivial simple cycle is represented by a distinct node called a *cycle node*. Let $cn(3ec(G))$ be the set of cycle nodes. For a cycle node s, let $cycle(s)$ be the set of all class nodes that are on the cycle s. The graph $Cyc(3ec(G))$ consists of the class nodes and cycle nodes of $3ec(G)$, where a class node c is adjacent to a cycle node s in $Cyc(3ec(G))$ iff c lies on cycle s in $3ec(G)$. Therefore, graph $Cyc(3ec(G))$ shows the incidence relation for class nodes and cycles. Moreover, graph $Cyc(3ec(G))$ is a tree called the *cycle tree* of G. The structure of $Cyc(3ec(G))$ is illustrated in Figure 3, where the cycle nodes are drawn as boxes.

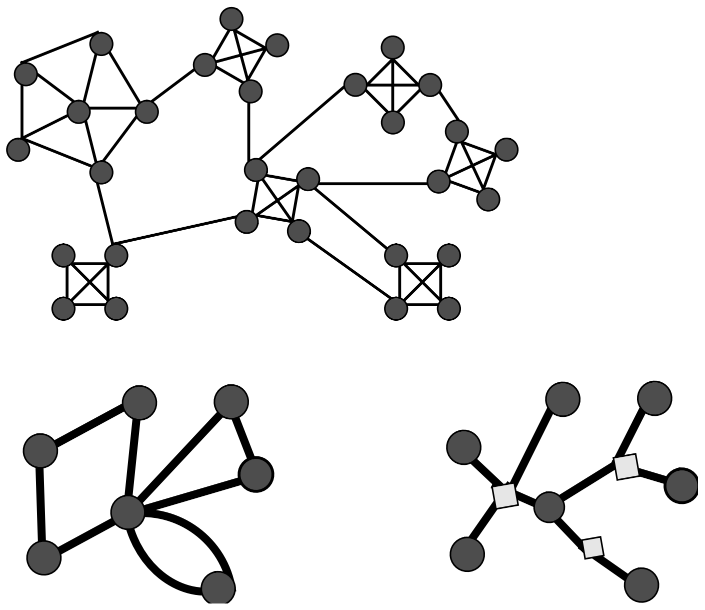


FIG. 3. *A* 2*-edge-connected graph* G *and the related graphs* $3ec(G)$ *and* $Cyc(3ec(G))$*.*

10.1.1. Edge insertions. We maintain the 3ec-relation under edge insertions by means of $Cyc(3ec(G))$. Suppose a new edge (e,x,y) is inserted in G. If $3ec(x) = 3ec(y)$, then by Lemma 2.2, the 3ec-relation, $3ec(G)$, and $Cyc(3ec(G))$ remain unchanged. So, we can assume that $3ec(x) \neq 3ec(y) \wedge 2ec(x) = 2ec(y)$. Then edge $(e, 3ec(x), 3ec(y))$ arises as a new edge in $3ec(G)$.

LEMMA 10.1 (see [21]). *Let G be a* 2*-edge-connected graph. Suppose that an edge* $(e, 3ec(x), 3ec(y))$ *is inserted to the graph* $3ec(G)$*. Then all the class nodes on the tree path P from* $3ec(x)$ *to* $3ec(y)$ *in* $Cyc(3ec(G))$ *become* 3*-edge-connected in* $3ec(G)$*, while the other pairs of distinct class nodes in* $3ec(G)$ *stay only* 2*-edge-connected.*

Thus, for all class nodes on P, all the corresponding classes form a new class (by Lemma 2.2). The update can now be performed as follows:

- obtain the tree path P between $3ec(x)$ and $3ec(y)$ in $Cyc(3ec(G))$,
- join all the classes "on" P into one new class C', and
- adapt the cycle tree $Cyc(3ec(G))$ accordingly.

The update heavily changes the structure of $Cyc(3ec(G))$. (For illustrations, we refer to [21].) The cycle tree changes as follows. Consider the simple cycle s and the class nodes c and d ($c \neq d$) such that s, c, and d are on P and $c, d \in cycle(s)$. Then classes c and d are joined into the new class c'. The original simple cycle s splits into two "smaller" simple cycles, each one consisting of the class node c' and of one of the two parts of the former cycle *between* c and d (we refer to [21]).

LEMMA 10.2 (see [21]). *Given a* 2*-edge-connected graph G of n nodes with a cycle tree, there exists a data structure for the* 3*ec-problem (that also maintains a* *cycle tree) such that the following holds. The total time for m insertions and queries is $O(m+n)$ time plus the time needed to perform $O(m+n)$ Finds and $O(n)$ Unions and Splits in a Union-Find or a Circular Split-Find structure for $O(n)$ elements. The data structure takes $O(n)$ space.*

Here, the *Circular Split-Find problem* [23] is a problem closely related to the Split-Find problem [11]. It deals with splitting cyclic lists into two new cyclic lists, determined by two splitting nodes. In [23], solutions for this problem are given with similar complexities as the UF-structures (cf. section 2.5). (They are closely related to [11].) We denote such structures similarly by GSF(i) and α-GSF structures. Later, we choose appropriate structures when applying Lemma 10.2.

10.2. General graphs.

10.2.1. Observations. We extend the solution of the previous section to general graphs. We first state observations of [21] (to which we refer for further details and figures). For detecting the 3ec-classes it suffices to detect the 3ec-classes inside the 2ec-components. Therefore, our algorithms for general graphs maintain the 2ec-classes (as in section 4), and they maintain the 3ec-classes within 2ec-components. We consider the forest of all cycle trees for the 2ec-components, called the *cycle forest* $Cyc(3ec(G))$ of G.

Suppose edge (e, x, y) is inserted in graph G yielding graph G'. If $c(x) \neq c(y)$, then the 2ec-classes and the 3ec-classes do not change. Otherwise, if $2ec(x) = 2ec(y)$, then (e, x, y) is inserted inside a 2ec-component and the changes as described in subsection 10.1.1 occur. Otherwise, we have $2ec(x) \neq 2ec(y) \wedge c(x) = c(y)$. Consider $2ec(G)$. Let P_2 be the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ (see subsection 4.1). Then the major changes are that

1. all 2ec-classes corresponding to class nodes on P_2 form one new 2ec-class,
2. for each 2ec-class C on P_2, the 3ec-classes inside C are changed, and
3. a new cycle s of 3ec-classes arises; the new cycle node s links the (updated) cycle trees that are contained in the 2ec-classes on P_2.

We consider the changes more precisely. The first part is identical to subsection 4.1. For the second part, we consider the changes of the 3ec-classes that occur in 2ec-classes on P_2. Consider a 2ec-class C on P_2 in $2ec(G)$. Let u and v be the two nodes in C that are x, y, or end nodes of interconnection edges between C and other classes on P. (We call u and v the *interconnection nodes* for C.) Then there is a new path between u and v in G' that does not intersect with C except for u and v, where such a path did not previously exist in G. Hence, considered within C only, this corresponds to inserting a temporary edge between the nodes u and v, since the 3ec-classes are completely determined by the 2ec-components in which they are contained (see Lemma 2.1). For the third part, now suppose all these "local" insertions are performed in the 2ec-classes on P_2. Then the two interconnection nodes in a 2ec-class C on P_2 are in the same (updated) 3ec-class in C, called the *interconnection* 3*ec-class* in C. All these interconnection 3ec-classes form a new cycle s. Then in the cycle forest s must be linked to these interconnection 3ec-classes, and thus it links the corresponding cycle trees.

10.2.2. Data structures and approach. We observe that when an edge (e, x, y) is inserted in a 2ec-component H, the changes in the 3ec-relation and $Cyc(3ec(H))$ are fully determined by just the 3ec-classes in which x and y are contained.

Consider a graph $G = \langle V, E \rangle$. We change the cycle forest $Cyc(3ec(G))$ by augmenting the collection of nodes of G and partitioning the thus-obtained 3ec-classes into subclasses. We do this as follows. Each 3ec-class in G may be extended with an arbitrary number of new, auxiliary nodes that are considered to be nodes in that 3ec-class (conceivably by means of artificial edges). The auxiliary nodes are not distinguished from the original nodes.

Each (extended) 3ec-class C of G is partitioned into *subclasses* of nodes. To each subclass a (new) distinct node is related called the *subclass node*. We call these the subclass nodes for C. The subclass node of the subclass to which x belongs is denoted by $sub(x)$. Now an *augmented cycle forest* AF_G for G is a forest on the subclass nodes and the cycle nodes of $Cyc(3ec(G))$ such that for each 3ec-class C of G the subclass nodes for C induce a subtree of AF_G and such that $Cyc(3ec(G))$ is obtained if for each 3ec-class C its subclass nodes are contracted into one node. We call an edge that links two subclass nodes of a 3ec-class C a *connector* for 3ec-class C. The set of all the connectors for C is called the *connector class* for C. Stated informally, AF_G can be obtained by replacing each class node in $Cyc(3ec(G))$ by some tree of subclass nodes and connectors. See Figure 4, where cycle nodes are drawn as boxes and (sub)class nodes as dots.

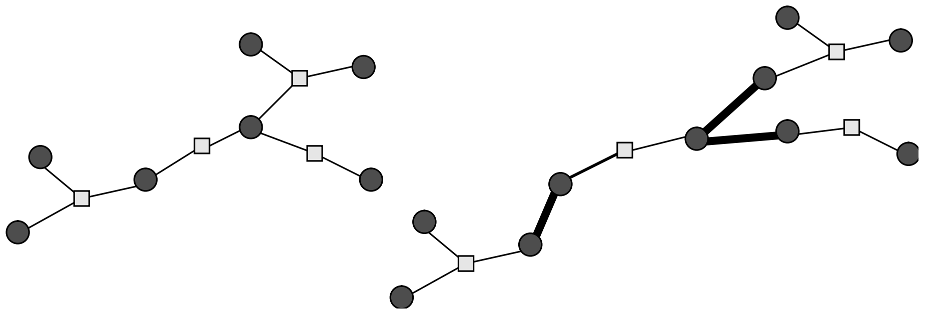


FIG. 4. *Graphs* $Cyc(3ec(G))$ *and* AF_G.

We consider the insertion of an edge (e, x, y) in a 2-edge-connected graph G in terms of AF_G. Let $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. The 3ec-classes on the tree path from $3ec(x)$ to $3ec(y)$ in $Cyc(3ec(G))$ correspond to the 3ec-classes that have at least one subclass on the tree path P between $sub(x)$ and $sub(y)$ in AF_G. Hence we can update the structure according to the following observations (also cf. subsection 10.1.1).

- Two successive subclass nodes on P (without a cycle node in between) correspond to the same class. Hence it suffices to obtain just the subclass nodes on P that are adjacent to a cycle node on P.
- All the classes of which a subclass node is "on" P must be joined into one new class C'.
- The augmented cycle tree AF_G must be adapted. Hence, all subclass nodes for C' must form a (sub)tree. This can be done by splitting each cycle s occurring on P and by joining the two subclasses that are the neighbors of s on P. (These updates can be performed locally as for cycle trees for each part of P without adjacent subclass nodes.)

Our goal structure is now as follows. To a graph G we relate a forest $bc(G)$ and an augmented cycle forest AF_G that satisfy the following. The graph $G = \langle V, E \rangle$ is extended with an (incremental) collection of auxiliary nodes, which are 3-edge-connected to least one original node. The (thus extended) vertex set is partitioned into disjoint sets, called *basic-clusters.* Each basic-cluster has a (new) unique node called *cluster node.* The nodes of forest $bc(G)$ are these cluster nodes. We call the edges of $bc(G)$ *bc-edges.* The following constraints are satisfied.

- Each 3ec-class C is partitioned into subclasses by intersecting C with the basic clusters. Then AF_G is an augmented cycle forest for G, based on this partition into subclasses.
- Each subclass node is considered to be contained in the basic cluster that contains its subclass. Then for a basic-cluster b, the subclass nodes that are contained in b together with appropriate cycle nodes of AF_G induce a subtree of AF_G denoted by $tree(b)$.
- Every connector in AF_G corresponds to exactly one edge in $bc(G)$ (and vice versa).

It follows that for a cluster b, $tree(b)$ does not have two adjacent subclass nodes. Therefore, $tree(b)$ is a cycle tree of some 2-edge-connected graph that has the nodes of basic-cluster b as its nodes together with a number of appropriate edges that induce the 3ec-relation as represented by $tree(b)$.

We observe that $bc(G)$ can be obtained from AF_G by contracting all subclass nodes in a basic-cluster b to cluster node b. Thus $bc(G)$ is an other contraction of AF_G (different from $Cyc(G)$). See Figure 5 for the example of Figure 4. We now define edge classes on $bc(G)$ as the classes inherited from the connector classes in AF_G (see section 2.1). Note that if two bc-edges incident with a cluster node b are in the same bc-edge class, then their originals in AF_G must have the same end node (a subclass node) in cluster b.

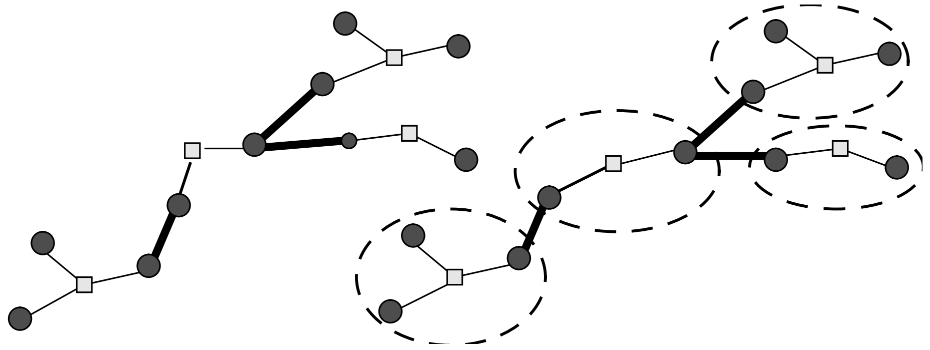


FIG. 5. *Forests AF_G and $bc(G)$.*

Now the strategy for inserting an edge (e, x, y) in 2-edge-connected graph G can be put in terms of $bc(G)$ as follows. Let c and d be the basic clusters containing x and y, respectively. Suppose that $c \neq d$.

- Let P' be the tree path in $bc(G)$ between c and d. Let P be the tree path in AF_G between $sub(x)$ and $sub(y)$. To obtain the relevant parts of P, it suffices to obtain a boundary list BL for c and d in $bc(G)$. This is seen as follows. The two incident bc-edges of an internal node b on P' are in the same bc-edge class. Hence their originals (which lie on P) have the same end node sb in b, which, therefore, is not adjacent to a cycle node on P. Note that for a boundary node $b \notin \{c, d\}$ of P', its two incident bc-edges on P' are not in the same bc-edge class, and thus cluster b contains at least two subclass nodes and one cycle node of P.

- For each such cluster b with $b \in BL$, a local update of the local cycle tree must be performed by joining all subclasses on the part P_b of P inside cluster b and by updating the local cycle tree correspondingly. The end nodes sb_1 and sb_2 of P_b are $sub(x)$, $sub(y)$, or the end nodes in cluster b of the originals of the bc-edges on P'. The latter can also be obtained from the bc-edges in the sublist of b in BL. The update thus corresponds to the update for inserting a temporary edge between any two nodes of G that are contained in subclasses sb_1 and sb_2.

10.2.3. Data structures and algorithms. In this section, we describe a data structure for the 3ec-problem, called *3EC structure*. We distinguish between the different layers of *representation*.

The representation of graph G is as follows. The vertex set of G may be extended from time to time with auxiliary nodes. There is a structure $2EC$ to maintain the 2ec-classes of G. This structure works on the regular nodes only, and, hence, the additional nodes are not involved. There is a "global" Union-Find structure UF_{3ec} for implementing the 3ec-classes of nodes of G. We recall that in the $2EC$ structure, there are Union-Find structures UF_c and UF_{2ec}.

Each node x has a pointer $clus(x)$ to the cluster node in which it is contained. Forest $bc(G)$ is implemented as a fractionally rooted tree structure (FRT) denoted by FRT_{3ec}.

The augmented cycle forest AF_G is not implemented as a whole. In fact, it is implemented in parts, viz., by cycle trees inside basic-clusters and by separate connectors. To be precise, we have the following implementation. Instead of a subclass node s as the end node of a connector, we take a node in subclass s as an end node. This is because subclasses are joined from time to time. Then the subclasses that are the ends of a connector (e, x, y) are $sub(x)$ and $sub(y)$. For a basic-cluster b, $tree(b)$ is implemented and maintained as a cycle tree as in Lemma 10.2. We refer to this as the *local structure*. The Union-Find and Circular Split-Find structures used in the local structure are denoted by UF_{loc} and GSF_{loc}. To each subclass, we relate a connector that has one of its end nodes in that subclass (if it exists) its *associated connector*.

The initialization for an empty graph is straightforward: each basic cluster contains one node. Also, a query corresponds to a Find in UF_{3ec}.

Suppose some new edge (e, x, y) is inserted in G, resulting in graph G'. Let the corresponding clusters for x and y be c and d. Then procedure $insert_3((e, x, y))$ updates the structure as follows if $3ec(x) \neq 3ec(y)$.

1. $c(x) \neq c(y)$. Then $insert_2((e, x, y))$ is performed
2. $c(x) = c(y) \wedge 2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. If $c = d$, then list BL is the list consisting of c with empty sublist; otherwise, $boundary(c, d)$ is performed in FRT_{3ec}, yielding boundary list BL in $bc(G)$. List BL is copied as list J but with empty sublists.
   For each basic cluster b in BL, the following is done. First, the nodes u and v in $tree(b)$ are obtained that are x, y, or end nodes of the originals of the bc-edges in the sublist of b (if any). If $3ec(u) \neq 3ec(v)$, then the following is done. A local insertion (Lemma 10.2) of a temporary edge (e', u, v) in basic-cluster b is performed to update $tree(b)$. Then an associated connector is obtained for each of the subclasses that are joined in cluster b, and the corresponding bc-edges are put in the sublist for b in J (since all classes must be joined later in the global structure). One of these connectors (if any) is assigned to the resulting subclass as its associated edge.

All the 3ec-classes of which a subclass was involved in the joinings are joined in UF_{3ec} (e.g., by taking a node from each subclass). Finally, the FRT_{3ec} structure is updated by means of call $joinclasses(J)$ (where if the sublist of node c or d is empty, then this node is removed from J.)

3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. First, the 2ec-classes that will be joined into one new class are determined. This is done as follows. A boundary list BL for x and y is computed in $2EC$ (this is only the first part of the call $insert_2((e, x, y))$). Subsequently, the names of the 2ec-classes are obtained. Then a linear list L is constructed that consists of these names and of those edges in BL with end nodes in different 2ec-classes: the names and edges alternate in L such that an edge is bracketed by the two corresponding 2ec-classes. Now L contains the tree path between $2ec(x)$ and $2ec(y)$ in $2ec(G)$ in the proper order.
For each 2ec-class C in L the following is done. We obtain the nodes u and v in C that are x, y, or the end nodes of the surrounding edges in L. If $3ec(u) \neq 3ec(v)$ then a temporary edge between u and v in C is inserted by a call $insert_3((e', u, v))$. Afterward, a new, auxiliary node z_C is created, and it is inserted in the interconnection 3ec-class $3ec(u)$. A connector (e'_C, z_C, z'_C) is created between z_C and some node z'_C of 3ec-class $3ec(u)$.
The nodes z_C $(C \in L)$ together form a new basic cluster b. Thus a cycle tree corresponding to the cycle of the new subclasses $\{z_C\}$ $(C \in L)$ is initialized in cluster b (in the same order as the 2ec-components C in L).
Cluster node b is linked with the involved trees in $bc(G)$ by means of new bc-edges as follows. For each z_C and connector (e'_C, z_C, z'_C), let $b' = clus(z'_C)$. Then a bc-edge (e'_C, b, b') is created for this connector, and a call $link((e'_C, b, b'))$ is performed in $bc(G)$. Edge (e'_C, z_C, z'_C) is associated with $sub(z_C)$ and $sub(z'_C)$. If $sub(z'_C)$ already had an associated edge (e'', z'', z'''), then (e'_C, b, b') is put in the class containing $(e'', clus(z''), clus(z'''))$ by a call of $joinclasses$.
The 2ec-classes in L are joined by performing a real call $insert_2((e, x, y))$ in $2EC$.

10.2.4. Complexity. We consider the complexity of the above algorithm. Regarding the creation of auxiliary nodes, suppose the initial graph G_0 has n (regular) nodes. The total number of new nodes created by the algorithm is at most $2n-1$, since a new node is created for each 2ec-class that is joined. Similarly, the total number of created clusters is at most $n-1$. Hence we only need FRT and Union-Find structures for $O(n)$ nodes. We denote all the Union-Find structures used independently in 3EC (not as part of FRT_{3ec} or FRT_{2ec}) by UF. We consider the UF structures to be one structure, on $O(n)$ elements. Obviously, a nonessential insertion takes time linear to $O(1)$ Finds.

LEMMA 10.3. *In a 3EC structure, the time for a sequence of essential insertions is at most linear to the time for a matching sequence of $O(n)$ operations on $O(n)$ nodes in FRT_{3ec} and FRT_{2ec} and for $O(n)$ Unions, Splits, and Finds in the UF and GSF_{loc} structures.*

Proof. We define a step to be an ordinary computational step or a Find operation in any UF or GSF_{loc} structure. We consider a collection of essential $insert_3$ operations, including the $insert_3$ calls during the execution of an $insert_3$ itself. Therefore, we do not consider the cost of an essential call $insert_3$ inside $insert_3$. Obviously, there are at most $O(n)$ essential insertions possible. So the essential $insert_3$ operations yield a matching sequence of $O(n)$ operations in FRT_{3ec}. Also, all calls $insert_2$

in the calls $insert_3$ are essential. Therefore, by Observation 4.2, the lemma holds for the operations in $2EC$.

We consider the *net cost* of the sequence of essential $insert_3$ calls: i.e., the cost of the parts of the computations apart from the computations considered above, from $O(1)$ steps per call $insert_3$, and from the Unions and Splits in the UF and GSF_{loc} structures.

1. Case $c(x) \neq c(y)$. Then there is no net cost.
2. Case $2ec(x) = 2ec(y) \wedge 3ec(x) \neq 3ec(y)$. We consider the net cost of a call $insert_3$. One part corresponds to the cost of essential local insertions inside clusters. This takes $O(1)$ steps for each such cluster and for each subclass that is joined. These $O(1)$ steps are considered to be included in the cost for joining two subclasses by a local insertion (at most 2 of these clusters have no subclasses that are joined).
   Since, in total, at most $O(n)$ essential local insertions can occur, the *net* cost is linear to $O(n)$ Finds in these structures (by Lemma 10.2).
3. $c(x) = c(y) \wedge 2ec(x) \neq 2ec(y)$. The computation of a boundary list in 2EC is a part of an essential call $insert_2$. The remainder takes $O(|L|)$ steps, plus a number of (other) $insert_3$ calls (this latter cost is included in case 2). Since the 2ec-classes in L are joined, the total net cost is $O(n)$ steps. ∎

A $3EC(i)$ *structure* is a 3EC structure where $FRT_{3ec} = FRT(i)$, $FRT_{2ec} = FRT(i)$, $UF = UF(i)$, and $GSF = GSF(i)$.

THEOREM 10.4. *A $3EC(i)$ structure solves the $3ec$-problem such that the following holds. The total time that is needed for all essential insertions is $O(n.i.a(i,n))$, whereas a query and nonessential insertion can be performed in $O(i)$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Proof. By Lemma 10.3, Theorems 2.3 and 9.1, and [23] (for GSF(i)), the theorem follows. ∎

The α-$3EC$ structure is a 3EC structure with $FRT_{3ec} = FRT(\alpha(n,n))$, $FRT_{2ec} = FRT(\alpha(n,n))$, $UF = \alpha$-UF, and $GSF = \alpha$-GSF, where in the latter structures the number of Finds is replaced by the number of *insert* operations and queries. Then, similarly as for Theorem 4.4, we obtain the following.

THEOREM 10.5. *The $3ec$-problem can be solved in $O(m.\alpha(m,n))$ total time (where m is the number of edge insertions and queries), where the fth query can be performed in $O(\alpha(f,n))$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

By using the α-$3EC$ structure where $FRT_{3ec} = \alpha$-FRT and $FRT_{2ec} = \alpha$-FRT instead, the above theorem can be augmented to allow insertions of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$ (cf. section 9).

11. A solution for 2-vertex-connectivity.

We consider the problem of maintaining the 2-vertex-connected components in a graph, and we will present algorithms with a time complexity of $O(n+m.\alpha(m,n))$ for n nodes and m queries and insertions using fractionally rooted trees. Similar to 2-edge-connectivity, we thus present a solution that is different from that given in [32] but whose approach is closer to the approach for maintaining the 3-vertex-connectivity relation in general graphs [24].

11.1. Graph observations.

Let $G = \langle V, E \rangle$ be a graph. We define the graph $2vc(G)$ as follows. For each 2vc-class or quasi class, there is a unique node related to that class called the class node. The vertices of $2vc(G)$ are the nodes of G together with these class nodes. For each node x, there is an edge between x and each class node c such that x is contained in 2vc-class c. (Thus we obtain a collection of trees corresponding to so-called block trees.) Hence $2vc(G)$ is a forest, where each tree in $2vc(G)$ corresponds to a connected component in G. For the insertion of an edge (e, x, y) in G, we have that all the classes of which the class node is on the tree path P between x and y in $2vc(G)$ form one new 2vc-class together (if P exists), while the other 2vc-classes and quasi classes remain unchanged.

We represent $2vc(G)$ by means of a spanning forest $SF(G)$ of G. We augment $SF(G)$ with edge classes on its set of edges. An edge class contains all the edges that connect two vertices that are in some 2vc-class or quasi class. An edge class consisting of a cut edge of G is called a *quasi* edge class, and a *real* edge class otherwise. Hence, a class of edges together with the end nodes of these edges induces a subtree in $SF(G)$, since for two 2-vertex-connected nodes x and y, all nodes on the tree path between x and y are 2-vertex-connected with them too. Also, two nodes x and y are 2-vertex-connected iff x and y are incident with 2 edges of the same real edge class. We use names of edge classes as the names of the corresponding 2vc-classes and quasi classes. Note that if each edge (e, x, y) in $SF(G)$ is replaced by two edges connecting class node C with x and y, where C is the edge class containing (e, x, y), then we obtain $2vc(G)$.

For the insertion of a new edge $(e, x, y) \notin E$ in G, we now have the following. If $c(x) \neq c(y)$, then (e, x, y) connects two connected components, and it thus connects two trees in $SF(G)$. If $\neg Is2vc(x, y) \wedge c(x) = c(y)$, then all edge classes occurring on the tree path between x and y in $SF(G)$ must be joined. Otherwise, we have $Is2vc(x, y) \wedge c(x) = c(y)$, and the insertion of (e, x, y) will not affect the 2vc-relation.

11.2. Algorithms. We use a fractionally rooted tree structure FRT on forest $SF(G)$, denoted by FRT_{2vc}. All quasi edge classes are marked as being quasi. All other classes are not marked. There is a Union-Find structure for connected components denoted by UF_c. The initialization for an empty graph is straightforward. A query $Is2vc(x, y)$ is performed by first performing a call *candidates*; then $false$ is returned if the returned edge-class names are distinct or correspond to a quasi edge class, while *true* and the (common) edge-class name are returned otherwise. For the insertion of a new edge (e, x, y) in G, we distinguish the two relevant cases.

1. $c(x) \neq c(y)$. Then $link((e, x, y))$ is performed, and the two connected components $c(x)$ and $c(y)$ are joined (in UF_c).
2. $\neg Is2vc(x, y) \wedge c(x) = c(y)$. Then a boundary list BL for x and y in $SF(G)$ is obtained by $boundary(x, y)$. If BL contains nodes x and y only, then x and y form a quasi class; then the edge class obtained in the call $Is2vc(x, y)$ is unmarked, reflecting that the edge class is real now. Otherwise, nodes x and y are deleted from BL (their sublists contain one edge only), and $joinclasses(BL)$ is called.

A $2VC(i)$ structure is the above structure where $FRT_{2vc} = FRT(i)$ and where $UF_c = UF(i)$. Then we obtain the following result in a way similar to subsection 4.3.

THEOREM 11.1. *A $2VC(i)$ structure solves the 2vc-problem such that the following holds. The total time that is needed for all essential insertions is $O(n.i.a(i, n))$, where a query and a nonessential insertion can be performed in $O(i)$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

Now take α-FRT as FRT_{2vc} for a graph with n nodes, and take α-UF for UF_c. Then we obtain the following result in a way similar to subsection 4, where now Theorem 9.2 is used instead of Theorem 9.1.

THEOREM 11.2. *The 2vc-problem can be solved in $O(m.\alpha(m, n))$ total time (where m is the number of edge insertions and queries), where the fth query can* *be performed in $O(\alpha(f,n))$ time. The structure can be initialized in $O(n)$ time and takes $O(n)$ space.*

The above theorem can be augmented to allow insertion of new nodes in the graph with a time complexity of $O(n + m.\alpha(m,n))$ (cf. section 9).

12. Concluding remarks. We have presented solutions for maintaining the 2-edge and the 3-edge-connected components of graphs under insertion of edges and vertices and for the 2-vertex-connected components. The solutions take $O(n + m.\alpha(m,n))$ time in total and are optimal on pointer machines and cell probe machines. The optimality follows from the $\Omega(n + m.\alpha(m,n))$ lower bound for k-edge/vertex-connectivity ($k \geq 1$) in general graphs [32]. Also, for all practical problem sizes, there is no need to perform transformations of FRT-structures; we recall that $a(2,n) = \log^* n$ and refer to section 9. Therefore, we conjecture that FRT(2), 2EC(2), 3EC(2), and 2VC(2) can be implemented as fast and easy structures in *practical* situations as well, with constant-time queries.

We have also presented linear-time algorithms for maintaining 2-*edge*-connectivity in a connected graph on a RAM. Since there is a nonlinear lower bound of $\Omega(n + m.\alpha(m,n))$ for maintaining 2-*vertex*-connectivity in connected graphs on a RAM [32], this shows an interesting difference in computational complexity between 2-edge-connectivity and 2-vertex-connectivity.

Finally, we remark that the problem of maintaining the 3-vertex-connected components of general graphs can be solved with the optimal complexity of $O(n + m.\alpha(m,n))$ time for m insertions and queries. This generalizes the special-case result in [4] for maintaining the 3-vertex-connectivity relation inside 2-vertex-connected graphs with such a time bound. We refer to [24].

REFERENCES

[1] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni, *Incremental algorithms for minimal length paths*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 12–21.

[2] R. F. Cohen and R. Tamassia, *Dynamic expression trees and their applications*, in Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1991, pp. 52–61.

[3] G. Di Battista and R. Tamassia, *Incremental planarity testing*, in Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS), 1989, pp. 436–441.

[4] G. Di Battista and R. Tamassia, *On-line algorithms with SPQR-trees*, in Proc. 17th Int. Colloq. on Automata, Languages, and Programming (ICALP), 1990, pp. 598–611.

[5] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, *Sparsification—A technique for speeding up dynamic graph algorithms*, J. ACM, 44 (1997), pp. 669–696.

[6] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer, *Separator based sparsification for dynamic planar graph algorithms*, in Proc. 25th Annual ACM Symposium on Theory of Computing (STOC), 1993, pp. 208–217.

[7] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, *Maintenance of a minimum spanning forest in a dynamic planar graph*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 1–11.

[8] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[9] G. N. Frederickson, *Ambivalent data structures for dynamic* 2*-edge-connectivity and k smallest spanning trees*, SIAM J. Comput., 26 (1997), pp. 484–538.

[10] M. L. Fredman and M. E. Saks, *The cell-probe complexity of dynamic data structures*, in Proc. 21st Annual ACM Symposium on Theory of Computing (STOC), 1989, pp. 345–354

[11] H. N. Gabow, *A scaling algorithm for weighted matching on general graphs*, in Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS), 1985, pp. 90–100.

[12] H. N. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 434–443.

[13] H. N. Gabow and R. E. Tarjan, *A linear time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209–221.
[14] Z. Galil and G. F. Italiano, *Maintaining the* 3*-edge-connected components of a graph on-line*, SIAM J. Comput., 22 (1993), pp. 11–28.
[15] F. Harary, *Graph Theory*, Addison–Wesley Publishing Company, Reading, MA, 1969.
[16] G. F. Italiano, *Amortized efficiency of a path retrieval data structure*, Theoret. Comput. Sci., 48 (1986), pp. 273–281.
[17] G. F. Italiano, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.
[18] J. A. La Poutré and J. van Leeuwen, *Maintenance of transitive closures and transitive reductions of graphs*, in Graph-Theoretic Concepts in Computer Science, H. Göttler and H. J. Schneider, eds., Lecture Notes in Comput. Sci. 314, Springer-Verlag, Berlin, 1987, pp. 106–120.
[19] J. A. La Poutré, *New techniques for the union-find problem*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 54–63.
[20] J. A. La Poutré, *Lower bounds for the union-find and the split-find problem on pointer machines*, J. Comput. System Sci., 52 (1996), pp. 87–99.
[21] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars, *Maintenance of* 2*- and* 3*-edge-connected components of graphs* I, Discrete Math., 114 (1993), pp. 329–359.
[22] J. A. La Poutré, *Maintenance of* 2*- and* 3*-Connected Components of Graphs, Part* II: 2*- and* 3*-Edge-Connected Components and* 2*-Vertex-Connected Components*, Tech. report RUU-CS-90-27, Utrecht University, The Netherlands, 1990.
[23] J. A. La Poutré, *Dynamic Graph Algorithms and Data Structures*, Ph.D. thesis, Utrecht University, The Netherlands, 1991.
[24] J.A. La Poutré, *Maintenance of triconnected components of graphs*, in Proc. 19th International Colloq. on Automata, Languages, and Programming (ICALP), 1992, pp. 354–365.
[25] J. A. La Poutré, *Alpha-algorithms for incremental planarity testing*, in Proc. 26th Annual ACM Symposium on Theory of Computing (STOC), 1994, pp. 706–715.
[26] K. Mehlhorn, *Data Structures and Algorithms* 2*: Graph Algorithms and NP-Completeness*, EATCS Monograph Series, Springer-Verlag, Berlin, 1984.
[27] F. P. Preparata and R. Tamassia, *Fully dynamic techniques for point location and transitive closure in planar structures*, in Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS), 1988, pp. 558–567.
[28] M. H. Rauch, *Fully dynamic biconnectivity in graphs*, Algorithmica, 13 (1995), pp. 503–538.
[29] H. Rohnert, *A dynamization of the all pairs least cost path problem*, in 2nd Annual Symposium on Theoretical Aspects of Computer Science, K. Mehlhorn, ed., Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, 1985, pp. 279–286.
[30] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.
[31] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.
[32] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.
[33] J. Westbrook, *Fast incremental planarity testing*, in Proc. 19th International Colloq. on Automata, Languages, and Programming (ICALP), 1992, pp. 342–353.
[34] Ye. Dinitz and J. Westbrook, *Maintaining the classes of y-edge-connectivity in a graph on-line*, Algorithmica, 20 (1998), pp. 242–276.
[35] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen, *On-line maintenance of the four-connected components of a graph*, in Proc. 32nd Annual Symp. on Foundations of Computer Science (FOCS), 1991, pp. 793–801.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1550–1567

RESTRUCTURING PARTITIONED NORMAL FORM RELATIONS WITHOUT INFORMATION LOSS[*]

MILLIST W. VINCENT[†] AND MARK LEVENE[‡]

Abstract. Nested relations in partitioned normal form (PNF) are an important subclass of nested relations that are useful in many applications. In this paper we address the question of determining when every PNF relation stored under one nested relation scheme can be transformed into another PNF relation stored under a different nested relation scheme without loss of information, referred to as the two schemes being data equivalent. This issue is important in many database application areas such as view processing, schema integration, and schema evolution. The main result of the paper provides two characterizations of data equivalence for nested schemes. The first is that two schemes are data equivalent if and only if the two sets of multivalued dependencies induced by the two corresponding scheme trees are equivalent. The second is that the schemes are equivalent if and only if the corresponding scheme trees can be transformed into the other by a sequence of applications of a local restructuring operator and its inverse.

Key words. database, nested relations, multivalued dependency, partitioned normal form

AMS subject classification. 68P15

PII. S0097539797326319

1. Introduction.

It is widely accepted now that the flat relational model, as defined by [5], needs to be extended or modified to cater to the demands of new application areas such as those involving textual, spatial, or scientific data [4, 17, 22, 25, 27]. One of the most important of these extensions is the *nested relational model*, originally proposed in [14], which removes the 1NF restriction that attributes values to be atomic and instead allows sets and relations as attribute values. Many aspects of nested relational databases have been investigated in the last decade including query languages and algebras, query optimization, normalization, and user interfaces [12, 13, 17, 19, 23, 24, 29]. Several research prototypes of nested databases have been implemented [20, 23, 26] and several major commercial database systems, such as ORACLE System 8 and ILLUSTRA [27], support nested relations.

Several subclasses of nested relations have also been investigated [29] and one subclass that has received considerable attention is the class of *partitioned normal form* (PNF) nested relations [1, 12, 21]. In a PNF relation the set of atomic attributes at every level of the nested relation must be a key. It has been shown that PNF relations possess several desirable properties that nested relations in general do not possess, such as nesting and unnesting operations commuting, which results in the semantics of PNF nested relations and their query languages being more transparent than for general nested relations [18]. It has been suggested that PNF relations are sufficient to model all practical applications of nested relations [7]. The issue that is investigated in this paper is determining when every PNF nested relation stored under one nested relation scheme can be transformed into another PNF nested relation stored under a different nested relation scheme without loss of semantic content, re-

[*]Received by the editors August 21, 1997; accepted for publication (in revised from) May 20, 1999; published electronically March 15, 2000.
http://www.siam.org/journals/sicomp/29-5/32631.html

[†]Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Adelaide, Australia 5095 (millist.vincent@unisa.edu.au).
[‡]Department of Computer Science, University College London, Gower Street, London, WC1E 6BT, UK (M.Levene@uk.ac.ucl.cs).

ferred to as the two schemes being *data equivalent*. Determining whether two schemes are data equivalent is an important issue in all data models, not only the nested relational model, and has impact on a variety of areas such as view processing, scheme integration, schema evolution, and schema translation in heterogeneous database systems [9, 10, 15, 16]. In particular, the recent work in [15, 16] has highlighted the importance of data equivalence, and the more general notion of data dominance, in the development of correct techniques for schema integration and schema translation and demonstrated that several existing methodologies were incorrect because of insufficient attention being paid to the issues of data equivalence and data dominance.

The approach to formalizing the notion of data equivalence used in this paper is based on the seminal work in [9, 11]. The essential idea of data equivalence is that two database schemes are data equivalent if there exists a bijection which maps from every instance of one scheme to an instance in the other. If this property is satisfied, then every instance of one scheme can be mapped to an instance in the other scheme and then recovered since the mapping is 1–1. If no restriction is placed on the instance mapping apart from the requirement that it be a bijection, then this notion of data equivalence is referred to as *absolute equivalence*. Several more restrictive notions of data equivalence were also introduce in [9, 11] based on further restricting the instance mapping. In particular, the other types of data equivalence introduced were (in increasing order of restrictiveness) *internal equivalence* (which does not allow the instance mapping "invent" more than a fixed set of constants), *generic equivalence* (which requires that the instance mapping commutes with permutations of the data values which leave the instances unchanged), and *query equivalence* (which requires that the instance mapping be a query in the data model). It has been shown that in the case of the relational model, the different notions of data equivalence are not identical, whereas in the format model introduced in [9, 11], absolute equivalence, internal equivalence, and generic equivalence are identical. In this paper, we will show that all four notions of data equivalence are identical for single nested relational schemes.

The main result of this paper is to provide two characterizations of absolute equivalence for PNF schemes. The first shows that two PNF schemes are absolutely equivalent if and only if the sets of *multivalued dependencies* (MVDs) induced by the corresponding scheme trees are equivalent. This result confirms the importance of considering dependencies in restructuring, an observation that had been previously made but not fully investigated [10]. For the second characterization, we introduce a local restructuring operator for the tree corresponding to a PNF scheme, called COMPRESS, which merges a parent node having a single child node with the child node. We then show that two PNF schemes are absolutely equivalent if and only if one tree can be transformed into the other by a sequence of applications of the COMPRESS operator and its inverse. Finally, we use these results to develop a polynomial time algorithm for testing whether two nested schemes are absolutely equivalent. We also point out that in the characterizing absolute equivalence we explicitly derive a 1–1 mapping between the PNF instances of the schemes expressed as a sequence of unnest and nest operations. The benefit of deriving an instance mapping, and especially one that is expressible as a query, is that the instance mapping can be used in database translation tools to translate any query expressed against one scheme into another query expressed against an equivalent scheme [9, 10, 15, 16]. We also note that the technique we use in proving the main result, using combinatorial methods and the path properties of balanced scheme trees, is a novel one and differs from the techniques used to characterize data equivalence in related data models [1, 2, 11].

The rest of the paper is organized as follows. In section 2 we introduce basic definitions and notations. In section 3 we derive the main result of the paper on characterizing absolute equivalence on nested schemes. In section 4 we discuss related work and section 5 contains concluding comments.

2. Definitions and notation.

We now introduce the definitions and notation to be used in later sections. More complete presentations can be found in standard references such as [3, 18].

2.1. Trees.

A *tree* is a finite, acyclic, directed graph in which there is a unique node, called the *root*, with indegree 0 and every other node has indegree 1. A node n' is a *child* of a node n (or equivalently, n is the *parent* of n') if there is a directed edge from n to n'. A pair of nodes are *siblings* if they are children of the same node. A node is a *leaf* if it has no children. A node n' is recursively defined to be a *descendant* of a node n (or equivalently, n is an *ancestor* of n') if either n' is a child of n, or n has a child n'' such that n' is a descendant of n''. The *height* of a tree T is the number of nodes on the longest path from the root to a leaf node.

A tree T' is a *subtree* of a tree T if the nodes of T' are a subset of those of T and for every pair of nodes n' and n, n' is a child of n in T if and only if n' is a child of n in T'. A subtree T' is a *principal subtree* of T if the root of T' is a child of the root of T. A tree T is *balanced* if every nonleaf node has at least two children.

2.2. Scheme trees and nested relations.

Let U be a fixed countable set of atomic attribute names. Associated with each attribute name $A \in U$ is a countably infinite set of values denoted by $\mathrm{DOM}(A)$ and the set $\mathbf{DOM}$ is defined by $\mathbf{DOM} = \cup\,\mathrm{DOM}(A_i)$ for all $A_i \in U$. We assume that $\mathrm{DOM}(A_i) \cap \mathrm{DOM}(A_j) = \emptyset$ if $i \neq j$. A *scheme tree* is a tree containing at least one node and whose nodes are labeled with nonempty sets of attributes that form a partition of a finite subset of U. If n denotes a node in a scheme tree T, then

$\mathrm{ATT}(n)$ is the set of attributes associated with n;
$\mathrm{ATT}(T)$ is the union of $\mathrm{ATT}(n)$ for all $n \in T$;
$\mathrm{ANC}(n)$ is the set of all ancestor nodes of n, including n;
$\mathrm{A}(n)$ is the union of $\mathrm{ATT}(n')$ for all $n' \in \mathrm{ANC}(n)$;
$\mathrm{DESC}(n)$ is the set of all descendant nodes of n, including n;
$\mathrm{D}(n)$ is the union of $\mathrm{ATT}(n')$ for all $n' \in \mathrm{DESC}(n)$;
$\mathrm{ROOT}(T)$ denotes the root node of T;
$\mathrm{HEIGHT}(T)$ denotes the height of T.

Figure 2.1 illustrates an example scheme tree defined over the set of attributes {STUDENT, MAJOR, CLASS, EXAM, PROJECT}.

If S and T are two scheme trees, then S and T are *isomorphic* if there exists a bijection τ from the nodes of S to those of T such that

(i) for any pair of nodes n' and n in S, n' is a child of n in T if and only if $\tau(n')$ is a child of $\tau(n)$ in S;

(ii) for every node $n \in S$, $\mathrm{ATT}(n) = \mathrm{ATT}(\tau(n))$.

We note that it follows from this definition that if scheme trees S and T are isomorphic, then $\mathrm{ATT}(S) = \mathrm{ATT}(T)$. It also follows that the scheme tree resulting from interchanging in a scheme tree of any pair of subtrees whose roots are siblings is isomorphic to the original tree. If a scheme tree T has leaf nodes $\{n_1, \ldots, n_m\}$, then the *path set* of T, $\mathrm{P}(T)$, is defined by $\mathrm{P}(T) = \{A(n_1), \ldots, A(n_m)\}$. For instance, the

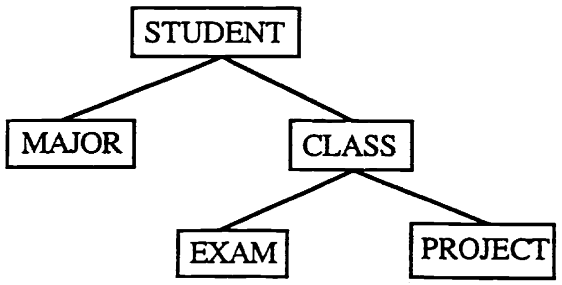


FIG. 2.1. *An example scheme tree.*

path set of the scheme tree in Figure 2.1 is {{STUDENT, MAJOR}, {STUDENT, CLASS, EXAM}, {STUDENT, CLASS, PROJECT}}.

A *nested relation scheme* (NRS) for a scheme tree T, denoted by $\mathrm{N}(T)$, is the set defined recursively by the following:

(i) if T consists of a single node u, then $\mathrm{N}(T) = \mathrm{ATT}(u)$;

(ii) if $A = \mathrm{ATT}(\mathrm{ROOT}(T))$ and $T_1, \ldots, T_k, k \geq 1$, are the principal subtrees of T, then $\mathrm{N}(T) = A \cup \{\mathrm{N}(T_1)\} \cup \cdots \cup \{\mathrm{N}(T_k)\}$.

For example, for the scheme tree T shown in Figure 2.1, $\mathrm{N}(T)$ = {STUDENT, {MAJOR},{CLASS, {EXAM}, {PROJECT}}}.

We now recursively define the domain of a scheme tree T, denoted by $\mathrm{DOM}(\mathrm{N}(T))$, by the following:

(i) if T consists of a single node n with $\mathrm{ATT}(n) = \{A_1, \ldots, A_n\}$, then $\mathrm{DOM}(\mathrm{N}(T)) = \mathrm{DOM}(A_1) \times \cdots \times \mathrm{DOM}(A_n)$;

(ii) if $A = \mathrm{ATT}(\mathrm{ROOT}(T))$ and $T_1, \ldots, T_p$ are the principal subtrees of T, then $\mathrm{DOM}(\mathrm{N}(T) = \mathrm{DOM}(A) \times \mathrm{P}(\mathrm{DOM}(\mathrm{N}(T_1))) \times \cdots \times \mathrm{P}(\mathrm{DOM}(\mathrm{N}(T_p)))$, where $\mathrm{P}(S)$ denotes the set of all *nonempty*, finite subsets of a set S.

We note in the above definition that, in contrast to the VERSO model defined in [1], we do not allow empty sets. A discussion of the effects of this assumption on our results is contained in section 4.

The set of atomic attributes in $\mathrm{N}(T)$, denoted by $\mathrm{Z}(\mathrm{N}(T))$, is defined by $\mathrm{Z}(\mathrm{N}(T)) = \mathrm{N}(T) \cap U$. The set of higher-order attributes in $\mathrm{N}(T)$, denoted by $\mathrm{H}(\mathrm{N}(T))$, is defined by $\mathrm{H}(\mathrm{N}(T)) = \mathrm{N}(T) - \mathrm{Z}(\mathrm{N}(T))$. For instance, for the example shown in Figure 2.1, $\mathrm{Z}(\mathrm{N}(T))$ = {STUDENT} and $\mathrm{H}(\mathrm{N}(T))$ = {{MAJOR}, {CLASS, {EXAM}, {PROJECT}}}. Also, if V is a subset of $\mathrm{N}(T)$ given by $V = \{A_1, \ldots, A_k, \{Y_1\}, \ldots, \{Y_p\}\}$, where $A_1, \ldots, A_k$ are the atomic attributes of V and $\{Y_1\}, \ldots, \{Y_p\}$ are the higher-order attributes, then the atomic attributes in V, denoted by $\mathrm{ATOM}(V)$, are recursively defined by $\mathrm{ATOM}(V) = \{A_1, \ldots, A_k\} \cup \mathrm{ATOM}(Y_1) \cdots \cup \mathrm{ATOM}(Y_p)$.

Finally we define a nested relation over a nested relation scheme $\mathrm{N}(T)$, denoted by $r(\mathrm{N}(T))$, or often simply by r when $\mathrm{N}(T)$ is understood, to be a finite nonempty set of elements from $\mathrm{DOM}(\mathrm{N}(T))$. If t is a tuple in r and Y is a nonempty subset of $\mathrm{N}(T)$, then $t[Y]$ denotes the restriction of t to Y and the restriction of r to Y is then the nested relation defined by $r[Y] = \{t[Y] \mid t \in r\}$. An example of a nested relation over the scheme tree shown in Figure 2.1 is shown in Table 2.1. The active domain of a nested relation r, denoted by $\mathrm{ACT}(r)$, is the subset of **DOM** containing atomic values appearing in r.

Table 2.1
A nested relation.

STUDENT	{MAJOR}	{CLASS	{EXAM}	{PROJECT}}
Anna	MATH	CS100	mid-year	Project A
	Computing		final	Project B
				Project C
Bill	Physics	P100	final	Prac Test 1
				Parc Test 2
	Chemistry	CH200	Test A	Experiment 1
			Test B	Experiment 2
			Test C	Experiment 3

2.3. Nested operators. We now define the two restructuring operators for nested relations, nest and unnest [28].

Let Y be a nonempty proper subset of $\mathrm{N}(T)$. Then the operation of *nesting* a relation r on Y, denoted by $\nu_Y(r)$, is defined to be a nested relation over the scheme $(\mathrm{N}(T) - Y) \cup \{Y\}$ and a tuple $t \in \nu_Y(r)$ if and only if (1) there exists $t' \in r$ such that $t[\mathrm{N}(T) - Y] = t'[\mathrm{N}(T) - Y]$ and (2) $t[Y] = \{t''[Y] \mid t'' \in r \text{ and } t''[\mathrm{N}(T) - Y] = t[\mathrm{N}(T) - Y]\}$.

Unnesting is defined as follows. Let $r(\mathrm{N}(T))$ be a relation and $\{Y\}$ an element of $\mathrm{H}(\mathrm{N}(T))$. Then the unnesting of r on $\{Y\}$, denoted by $\mu_{\{Y\}}(r)$, is a relation over the nested scheme $(\mathrm{N}(T) - \{Y\}) \cup Y$ and a tuple $t \in \mu_{\{Y\}}(r)$ if and only if there exists $t' \in r$ such that $t'[\mathrm{N}(T) - \{Y\}] = t[\mathrm{N}(t) - \{Y\}]$ and $t[Y] \in t'[\{Y\}]$.

More generally, one can define the total unnest of a relation, $\mu^*(r)$, as the flat relation defined recursively by the following:

(1) if r is a flat relation, then $\mu^*(r) = r$;

(2) otherwise $\mu^*(r) = \mu^*(\mu_{\{Y\}}(r))$, where $\{Y\}$ is a higher-order attribute in the NRS for r.

It can be shown [28] that the order of unnesting is immaterial and so μ^* is uniquely defined.

2.4. Constraints. We now define dependencies in a nested relation in a similar fashion to the way they are defined in flat relations. If $r(\mathrm{N}(T))$ is a nested relation and Y and Z are subsets of $\mathrm{N}(T)$, then r satisfies the *functional dependency* (FD) $Y \rightarrow Z$ if and only if for all tuples $t, t' \in r$, if $t[Y] = t'[Y]$, then $t[Z] = t'[Z]$. The MVD $Y \rightarrow\rightarrow Z$ is satisfied in r if and only if for all $t, t' \in r$ such that $t[Y] = t'[Y]$ there exists $s \in r$ such that $s[Y] = t[Y]$, $s[Z] = t[Z]$, and $s[\mathrm{N}(T) - Z] = t'[\mathrm{N}(T) - Z]$. Given a set Σ of FDs and MVDs and an FD $Z \rightarrow W$ (or MVD $Z \rightarrow\rightarrow W$), Σ *implies* the FD $Z \rightarrow W$ (or MVD $Z \rightarrow\rightarrow W$) if every relation in $\mathrm{SAT}(\Sigma)$ also satisfies $Z \rightarrow W$ (or $Z \rightarrow\rightarrow W$). The *closure* of a set Σ of a FDs and MVDs, denoted by Σ^+, is the set of FDs and MVDs implied by Σ. Two sets of dependencies, Σ and Ψ, are *equivalent*, written as $\Sigma \equiv \Psi$, if $\Sigma^+ = \Psi^+$. In some places in the rest of the paper we wish to consider only flat relations and we denote by $\mathrm{SAT}(\Sigma)$ the set of all *flat relations* which satisfy a set Σ of FDs and MVDs.

The set of MVDs induced by a scheme tree T is defined as follows [17]. If (n, n') is an edge in T, then the MVD associated with this edge is the MVD $\mathrm{A}(n) \rightarrow\rightarrow \mathrm{D}(n')$ and $\mathrm{MVD}(T)$ is the set of all such MVDs associated with all the edges in T. For example, if T is the scheme tree in Figure 2.1, then $\mathrm{MVD}(T) = \{$STUDENT $\rightarrow\rightarrow$ MAJOR, STUDENT $\rightarrow\rightarrow$ CLASS EXAM PROJECT, STUDENT CLASS $\rightarrow\rightarrow$ EXAM, STUDENT CLASS $\rightarrow\rightarrow$ PROJECT$\}$. A *join dependency* (JD), denoted by $\bowtie [R_1, \ldots, R_p]$, where $R_i \subseteq U$ for $1 \leq i \leq p$, is satisfied in a flat relation r if $r = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_p}(r)$, where π denotes the projection operator.

The subclass of nested relations we investigate in the paper, PNF relations, was introduced in [21]. If T is a scheme tree and $\mathrm{PNF}(T)$ denotes the set of all PNF nested relations over T, then a relation $r \in \mathrm{PNF}(T)$ if and only if the following conditions hold:

(i) r is a nested relation defined over $\mathrm{N}(T)$;

(ii) $\mathrm{Z}(\mathrm{N}(T))$ is a key for r, i.e., satisfies the FD $\mathrm{Z}(\mathrm{N}(T)) \rightarrow \mathrm{H}(\mathrm{N}(T))$;

(iii) for every $Y \in \mathrm{H}(\mathrm{N}(T))$ and $t \in r, t[Y]$ is in PNF.

3. Absolute equivalence for PNF schemes. In this section we establish the main results of the paper on characterizing absolute equivalence for scheme trees. First, we recall the definition of absolute equivalence between schemes [9].

DEFINITION 3.1. *Let T and S be scheme trees where $\mathrm{ATT}(T) = \mathrm{ATT}(S) = U$ and let $X \subseteq \mathbf{DOM}$. The set $\mathrm{DOM}_X(T)$ is defined by $\mathrm{DOM}_X(T) = \{r \mid r \in \mathrm{PNF}(T)$ and $\mathrm{ACT}(r) \subseteq X\}$. Then T is* absolutely equivalent *to S if there exists a positive integer N such that for all $X \subseteq \mathbf{DOM}$ such that $|X \cap \mathrm{DOM}(A_i)| \geq N$* [1] *for every $A_i \in U$, there exists a bijection from $\mathrm{DOM}_X(T)$ to $\mathrm{DOM}_X(S)$ ($|X|$ denotes the cardinality of a set X).*

A simple consequence of Definition 3.1 is the following alternative characterization [9] of absolute equivalence in terms of domain sizes. This characterization is fundamental to the later results in this paper.

LEMMA 3.2. *Let T and S be scheme trees where $\mathrm{ATT}(T) = \mathrm{ATT}(S)$ and let $X \subseteq \mathbf{DOM}$. Then T and S are absolutely equivalent if and only if there exists a positive integer N such that for all $X \subseteq \mathbf{DOM}$ such that $|X \cap \mathrm{DOM}(A_i)| \geq N$ for every $A_i \in U, |\mathrm{DOM}_X(T)| = |\mathrm{DOM}_X(S)|$.*

We now show that $|\mathrm{DOM}_X(T)|$ can be computed by the following recursively defined integer function.

DEFINITION 3.3. *If T is a scheme tree and $X \subseteq \mathbf{DOM}$, then the integer function $\alpha_X(T)$ is recursively defined as follows:*

(i) *if T consists of a single node with attributes $A_1, \ldots, A_k$, then*

$$\alpha_X(T) = 2^{x_1 \cdots x_k} - 1,$$

where $x_i = |X \cap \mathrm{DOM}(A_i)|, 1 \leq i \leq k$;

(ii) *otherwise if T has a root node n with attributes $A_1, \ldots, A_m$ and nonempty principal subtrees $T_1, \ldots, T_p$, then*

$$\alpha_X(T) = (\alpha_X(T_1) \cdots \alpha_X(T_p) + 1)^{x_1 \cdots x_m} - 1.$$

For example, given the scheme trees S and T shown in Figure 3.1,

$$\alpha_X(T) = 2^{x_1 x_2 x_3} - 1,$$

$$\alpha_X(S) = ((2^{x_2 - 1})(2^{x_3} - 1) + 1)^{x_1} - 1.$$

LEMMA 3.4. *If T is scheme tree and $X \subseteq \mathbf{DOM}$, then $|\mathrm{DOM}_X(T)| = \alpha_X(T)$.*

Proof. Part (i) of Definition 3.3 follows immediately from the fact that for any set with x elements, the number of nonempty subsets is $2^x - 1$. Part (ii) follows

[1]This is done to avoid combinatorial interaction that can occur with small domains [2].

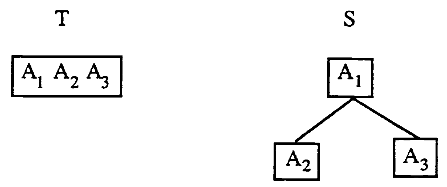


FIG. 3.1. *Example scheme trees.*

from the following combinatorial argument. There are $x_1 \cdots x_m$ distinct values for the set of attributes $A_1 \cdots A$ (we will also refer to this set of attributes as the key for T) and so, by the PNF assumption, any relation in PNF(T) can contain at most $x_1 \cdots x_m$ tuples. To count the number of possible relations, each key value is either not present in a relation or, if it is, it can be associated with $\alpha_X(T_1) \cdots \alpha_X(T_p)$ distinct values for the other attributes. There are thus $\alpha_X(T_1) \cdots \alpha_X(T_p) + 1$ different tuples involving a key value (the $+1$ for the case where the key value is missing from a relation). Then since there are $x_1 \cdots x_m$ possible tuples in a relation, there are $(\alpha_X(T_1) \cdots \alpha_X(T_p) + 1)^{x_1 \cdots x_m} - 1$ possible relations (the -1 is because we do not permit empty sets) [9]. $\square$

A straightforward corollary of the previous Lemma 3.2 is the following which characterizes absolute equivalence using $\alpha_X(T)$.

LEMMA 3.5. *Let T and S be scheme trees where* $\mathrm{ATT}(T) = \mathrm{ATT}(S)$ *and let* $X \subseteq \mathbf{DOM}$. *Then T is absolutely equivalent to S if and only if there exists a positive integer N such that for all* $X \subseteq \mathbf{DOM}$ *such that* $|X \cap \mathrm{DOM}(A_i)| \geq N$ *for every* $A_i \in U, \alpha_X(S) = \alpha_X(T)$.

Another interesting corollary of Lemma 3.4 is the following result which provides a characterization of the number of flat relations which satisfy MVD(T); a result that the reader can verify is not easy to obtain directly.

COROLLARY 3.6. *If* $\mathrm{DOM}_X(\mathrm{MVD}(T)) = \{r \mid r \in \mathrm{SAT}(\mathrm{MVD}(T))$ *and* $\mathrm{ACT}(r) \subseteq X$ *where* $X \subseteq \mathbf{DOM}\}$*, then* $|\mathrm{DOM}_X(\mathrm{MVD}(T))| = \alpha_X(T)$.

Proof. From Lemma 3.18 (see later), there exists a bijection from PNF(T) to SAT(MVD(T)) which preserves active values and so $|\mathrm{DOM}_X(\mathrm{MVD}(T))| = |\mathrm{DOM}_X(T)|$ and the result follows from Lemma 3.4. $\square$

We now introduce a restructuring operator for PNF scheme trees, called COMPRESS.[2] As will be shown shortly, the application of the COMPRESS operator yields a scheme tree which is absolutely equivalent to the original scheme tree. We will also show later, in the main result of the paper, that the COMPRESS operator is complete in the sense that two scheme trees are absolutely equivalent only if one can be transformed into the other by repeated application of the COMPRESS operator and its inverse.

DEFINITION 3.7. *Let T be a scheme tree and n a node in T with a single child node n' having children $n_1, \ldots, n_k$. Then the* COMPRESS *operator results in replacing the subtree with root n by a subtree with root containing* $\mathrm{ATT}(n) \cup \mathrm{ATT}(n')$ *and having children $n_1, \ldots, n_k$.*

The COMPRESS operator is illustrated in Figure 3.2.

Next, we show that the number of PNF instance of a scheme tree is invariant under the COMPRESS operator and so COMPRESS preserves absolute equivalence.

[2]The inverse of COMPRESS is referred to as elementary compaction in [1]

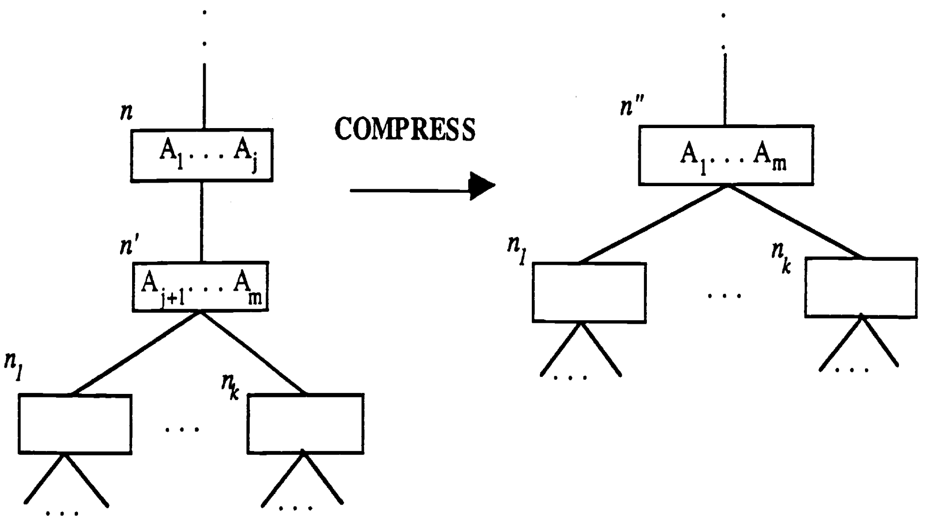


FIG. 3.2. *COMPRESS operator.*

LEMMA 3.8. *If T is a scheme tree, then $\alpha_X(T) = \alpha_X(\mathrm{COMPRESS}(T))$, where* COMPRESS$(T)$ *denotes the scheme tree obtained by applying the COMPRESS operator to an arbitrary node in the tree.*

Proof. From Definition 3.3, it suffices to show that $\alpha_X(T) = \alpha_X(S)$ where, from Figure 3.2, T is the subtree rooted at node n and S is the subtree rooted at node n''. From Definition 3.3 we have

$$\alpha_X(T) = (\alpha_X(T') + 1)^{x_1 \cdots x_j} - 1,$$

where T' is subtree at n'. If n' has no children, then $\alpha_X(T)$ can be written as

$$\begin{aligned}\alpha_X(T) &= (2^{x_{j+1} \cdots x_m} - 1 + 1)^{x_1 \cdots x_j} - 1\\ &= 2^{x_1 \cdots x_m} - 1\\ &= \alpha_X(S).\end{aligned}$$

Alternatively, if n' has children $n_1, \dots, n_k$ and corresponding subtrees $T_1, \dots, T_k$ rooted at these nodes, then $\alpha_X(T)$ can be written as

$$\begin{aligned}&= ((\alpha_X(T_1) \cdots \alpha_X(T_k) + 1)^{x_{j+1} \cdots x_m} - 1 + 1)^{x_1 \cdots x_j} - 1\\ &= (\alpha_X(T_1) \cdots \alpha_X(T_k) + 1)^{x_1 \cdots x_m} - 1\\ &= \alpha_x(S). \qquad \square\end{aligned}$$

It follows from this result that by repeatedly applying the COMPRESS operator to a scheme tree T until no more applications can be applied, a balanced tree is obtained which is absolutely equivalent to the original tree. In the terminology of [11], this tree can be considered as a *normal form tree* for T. We now show the crucial result that for balanced scheme trees, the absolute equivalence property coincides with the trees being isomorphic. It is interesting to note that the property of a scheme tree being balanced is unique to the context of PNF relations, since the property does not arise in the normal form trees developed in related nested data models which do not enforce PNF [2, 11].

THEOREM 3.9. *If S and T are balanced scheme trees such that* $\mathrm{ATT}(S) = \mathrm{ATT}(T)$*, then S and T are absolutely equivalent if and only if S and T are isomorphic.*

The proof of the "only if" part this theorem (the "if" part is trivial) requires several steps in the form of several preliminary lemmas and so we first summarize these steps. We first show that if S and T are not isomorphic, then there exist attributes A and B such that A and B belong to the same path in one scheme tree, say S, but not in T. We then establish lower and upper bounds for $\alpha_X(S)$ and $\alpha_X(T)$ and show that if we construct a domain X where the number of attribute values of any attribute in X, except for those of A and B, is fixed the number of A and B values in X are allowed to be large but identical, then $\alpha_X(S) > \alpha_X(T)$ and so S and T cannot be absolutely equivalent. To establish the first preliminary lemma on the relationship between isomorphisms and path sets in scheme trees, we need the following definitions.

DEFINITION 3.10. *For any pair of attributes A and B in a scheme tree $T, A \sim B$ denotes that there exists an element of* $\mathrm{P}(T)$ *to which they both belong, and $A \not\sim B$ denotes the converse. If S and T are scheme trees such that* $\mathrm{ATT}(S) = \mathrm{ATT}(T)$*, then S and T are defined to be* path equivalent*, denoted by $S \sim T$, if for all $A, B \in$* $\mathrm{ATT}(S)$*, $A \sim B$ in T if and only if $A \sim B$ in S. We denote by $S \not\sim T$ the fact that S and T are not path equivalent.*

For example, in the scheme tree S in Figure 3.1, $A_1 \sim A_2, A_1 \sim A_3$, but $A_2 \not\sim A_3$ and so $S \not\sim T$ since $A_2 \sim A_3$ in T. We now show that for two balanced scheme trees, path equivalence coincides with the trees being isomorphic.

LEMMA 3.11. *If T and S are balanced scheme trees such that* $\mathrm{ATT}(T) = \mathrm{ATT}(S)$*, then S and T are isomorphic if and only if $T \sim S$.*

Proof. If: the proof is by induction on Max $(\mathrm{HEIGHT}(T), (\mathrm{HEIGHT}(S))$. The result clearly holds when Max $(\mathrm{HEIGHT}(T), (\mathrm{HEIGHT}(S)) = 1$, so assume inductively that the result is true for all trees where Max $(\mathrm{HEIGHT}(T), (\mathrm{HEIGHT}(S)) < k, k > 1$, and we shall establish the result is true when Max $(\mathrm{HEIGHT}(T), (\mathrm{HEIGHT}(S)) = k$.

We first prove that the sets of attributes belonging to leaf nodes in S and T are the same. If not, there exists an attribute A which, without loss of generality, is in a leaf node of S but in a nonleaf node n in T. From the balanced assumption, n has at least two children and so there exists an attribute B in one child of n and an attribute C in another and so $B \not\sim C$ in T. This is a contradiction since by assumption $S \sim T$ and so $A \sim B$ and $A \sim C$, but this implies that $B \sim C$ in S since an attribute in a leaf node belongs to at most one element of the path set of a scheme tree.

Next we claim that there is a bijection ρ from the leaf nodes of S to the leaf nodes of T such that $\mathrm{ATT}(n) = \mathrm{ATT}(\rho(n))$ for every leaf node n in S. Otherwise, since the sets of attributes in leaf nodes are the same, there must exist a pair of attributes A and B such that A and B are in the same leaf node in S but in different leaf nodes of T thus contradicting the assumption that $S \sim T$.

Since the sets of leaf attributes are the same in both trees, the sets of attributes in nonleaf nodes are the same and so by the induction hypothesis the scheme trees S' and T' are isomorphic, where S' and T' are the trees obtained by removing the leaf nodes from S and T, respectively. Hence there exists a bijection τ from the nodes of S' to those of T'. Consider the mapping from the nodes of S to those in T defined by $\sigma(n) = \rho(n)$ if n is a leaf node in S and $\tau(n)$ otherwise. It is clear that σ satisfies property (ii) of a scheme tree isomorphism from the induction hypothesis and the property of ρ established above. Also, by the induction hypothesis, σ satisfies the property (i) for any pair of nodes in S' and so to complete the proof it suffices to show that a leaf node n' in S has a parent node n if and only if $\sigma(n')$ has parent node $\sigma(n)$ in T. If this is not the case, then by the induction hypothesis $\sigma(n)$ must be a leaf node in T' such that $\mathrm{ATT}(n) = \mathrm{ATT}(\rho(n))$ and so $A \sim B$ in S for any $A \in \mathrm{ATT}(n)$ and $B \in \mathrm{ATT}(n')$, but $A \not\sim B$ in T which completes the proof of the "if" part.

Only if: This part is immediate from the definition of isomorphism between scheme trees. $\square$

We note that the previous result holds only if the trees are balanced. For example, if S and T are scheme trees such that $\mathrm{N}(S) = \{A, \{B\}\}$ and $\mathrm{N}(T) = \{B, \{A\}\}$, then $T \sim S$ but S and T are not isomorphic. We next introduce two auxiliary functions and show that they are upper and lower bounds for $\alpha_X(T)$.

Definition 3.12. *If T is a scheme tree with* $\mathrm{ATT}(T) = \{A_1, \ldots, A_n\}$ *and* $X \subseteq$ **DOM**, *then the integer function $\beta_X(T)$ is defined by*

$$\beta_X(T) = 2^{p_X(T)},$$

where $p_X(T)$ is defined recursively by

$$p_X(T) = x_1 \cdots x_n \quad \textit{if } T \textit{ consists of a single node};$$

otherwise

$$p_X(T) = x_1 \cdots x_m (p_X(T_1) + \cdots + p_X(T_k) + 1),$$

where $\mathrm{ATT}(\mathrm{ROOT}(T)) = \{A_1, \ldots, A_m\}$, $x_i = |X \cap \mathrm{DOM}(A_i)|$, $1 \leq i \leq n$, *and* $T_1, \ldots, T_k$ *denote the principal subtrees of* T.

For example, for the scheme trees S and T shown in Figure 3.1,

$$\beta_X(T) = 2^{x_1 x_2 x_3} \quad \text{and} \quad \beta_X(S) = 2^{(x_2 + x_3 + 1)x_1}.$$

Definition 3.13. *If T is a scheme tree with* $\mathrm{ATT}(T) = \{A_1, \ldots, A_n\}$ *and* $X \subseteq$ **DOM**, *then the integer function $\delta_X(T)$ is defined by*

$$\delta_X(T) = 2^{q_X(T)}$$

and $q_X(T)$ is defined recursively by

$$q_X(T) = x_1 \cdots x_n - 1 \textit{ if } T \textit{ consists of a single node};$$

otherwise

$$q_X(T) = (x_1 \cdots x_m - 1)(q_X(T_1) + \cdots + q_X(T_k)),$$

where $\mathrm{ATT}(\mathrm{ROOT}(T)) = \{A_1, \ldots, A_m\}$, $x_i = |X \cap \mathrm{DOM}(A_i)|$, $1 \leq i \leq n$, *and* $T_1, \ldots, T_k$ *denote the principal subtrees of* T.

For example, for the scheme trees S and T shown in Figure 3.1,

$$\delta_X(T) = 2^{x_1 x_2 x_3 - 1} \quad \text{and} \quad \delta_X(S) = 2^{(x_2 - 1 + x_3 - 1)(x_1 - 1)}.$$

Lemma 3.14. *If T is a scheme tree where* $\mathrm{ATT}(T) = \{A_1, \ldots, A_n\}$ *and* $|X \cap \mathrm{DOM}(A_i)| > 1$, $1 \leq i \leq n$, *then* $\delta_X(T) < \alpha_X(T) < \beta_X(T)$.

Proof. We shall first show that $\alpha_X(T) < \beta_X(T)$ by induction on HEIGHT(T). The result clearly holds for the case where the height of tree is 1 so assume inductively that it holds for trees of height less than some positive integer p. Then

$$\begin{aligned}
\alpha_X(T) &= (\alpha_X(T_1)\cdots\alpha_X(T_k)+1)^{x_1\cdots x_m} - 1 && \text{by definition}\\
&< (\alpha_X(T_1)\cdots\alpha_X(T_k)+1)^{x_1\cdots x_m} \\
&< (\beta_X(T_1)\cdots\beta_X(T_k)+1)^{x_1\cdots x_m} && \text{by the induction hypothesis}\\
&= (2^{p_X(T_1)+\cdots+p_X(T_k)}+1)^{x_1\cdots x_m} && \text{by definition of } \beta_X(T)\\
&\le (2^{p_X(T_1)+\cdots+p_X(T_k)+1})^{x_1\cdots x_m} && \text{since } 2^a+1 \le 2^{a+1} \text{ for } 0 \le a\\
&= 2^{(p_X(T_1)+\cdots+p_X(T_k)+1)x_1\cdots x_m}\\
&= \beta_X(T).
\end{aligned}$$

We now show $\delta_X(T) < \alpha_X(T)$ again by induction on the HEIGHT(T). First, one can establish by a simple induction argument, which we omit, that $\alpha_X(T) > 1$. In the case that HEIGHT$(T) = 1, \delta_X(T) < \alpha_X(T)$ since $a^b - 1 > a^{b-1}$ when $a > 1$ and $b > 1$. Assume inductively that $\delta_X(T) < \alpha_X(T)$ holds for trees of height less than some positive integer p. Then

$$\begin{aligned}
\alpha_X(T) &= (\alpha_X(T_1)\cdots\alpha_X(T_k)+1)^{x_1\cdots x_m} - 1 && \text{by definition}\\
&> (\alpha_X(T_1)\cdots\alpha_X(T_k))^{x_1\cdots x_m} - 1\\
&> (\alpha_X(T_1)\cdots\alpha_X(T_k))^{x_1\cdots x_m - 1} && \text{since } \alpha_X(T) > 1 \text{ and } a^b - 1\\
& && \quad > a^{b-1} \text{ when } a > 1 \text{ and } b > 1\\
&> (\delta_X(T_1)\cdots\delta_X(T_k))^{x_1\cdots x_m - 1} && \text{by the induction hypothesis}\\
&= (2^{q_X(T_1)+\cdots+q_X(T_k)})^{x_1\cdots x_m - 1} && \text{from the definition of } \delta_X(T)\\
&= 2^{(q_X(T_1)+\cdots+q_X(T_k))(x_1\cdots x_m - 1)}\\
&= \delta_X(T). \qquad \square
\end{aligned}$$

We now use Lemma 3.14 to provide upper and lower bounds for $\alpha_X(T)$. $\square$

LEMMA 3.15. *Let T be a scheme tree,* ATT$(T) = \{A_1, \dots, A_n\}$*, and* $A_i, A_j \in$ ATT(T).

(i) *If $A_i \sim A_j$ in T and if $x_i = x_j = x$ and $x_k > 1$ for all k, $1 \le k \le n$, then* $\alpha_X(T) > 2^{(x-1)^2}$.

(ii) *If $A_i \not\sim A_j$ in T and if $x_i = x_j = x$ and $x_k > 1$ for all $k, 1 \le k \le n$, then there exist expressions C_1 and C_2, independent of x, such that $\alpha_X(T) < 2^{C_1 x + C_2}$.*

Proof. (i) We first note that one can easily see by an inductive argument that $q_X(T) > 1$, where $q_X(T)$ is defined in Definition 3.13. Next we prove by induction on the height of T that $q_X(T) > (x_i - 1)$ for all $i, 1 \le i \le n$. This result clearly holds when HEIGHT$(T) = 1$. Assume inductively then that the result holds for all trees of height up to k and consider the case where the height of T is $k+1$. If A_i is in the root of T, then

$$\begin{aligned}
q_X(T) &= (x_1 \cdots x_i \cdots x_m - 1)(q_X(T_1) + \cdots + q_X(T_k))\\
&> x_i - 1 && \text{since } q_X\text{'s} > 1 \text{ and all the } x\text{'s are} > 1 \text{ by assumption.}
\end{aligned}$$

If instead A_i is in a principal subtree T_p, then

$$\begin{aligned} q_X(T) &= (x_1 \cdots x_m - 1)(q_X(T_1) + \cdots + q_X(T_p) \ldots + q_X(T_k)) \\ &> (x_1 \cdots x_m - 1) q_X(T_p) \\ &> q_X(T_p) \\ &> x_i - 1 \qquad \text{by the induction hypothesis.} \end{aligned}$$

We now complete the proof of (i) by using induction on HEIGHT(T) to show that $p_X(T) > (x-1)^2$ from which (ii) follows using Lemma 3.14. In the case where HEIGHT(T) = 1, then A_i and A_j must both be in ROOT(T), and so

$$\begin{aligned} q_X(T) &= x_1 \cdots x_i \cdots x_j \cdots x_m - 1 \\ &> x^2 - 1 \qquad \text{since all the } x\text{'s are} > 1 \\ &> (x-1)^2 \qquad \text{since } x > 1. \end{aligned}$$

Assume inductively then that the result holds for all trees up to height k, and let T be of height $k+1$. If A_i and A_j are both in ROOT(T), then

$$\begin{aligned} q_X(T) &= (x_1 \cdots x_i \cdots x_j \cdots x_m - 1)(q_X(T_1) + \cdots + q_X(T_k)) \\ &> (x-1)^2 (q_X(T_1) + \cdots + q_X(T_k)) \\ &> (x-1)^2 \qquad \text{since } q_X(T) > 1 \text{ as noted above.} \end{aligned}$$

Alternatively, suppose that A_i is in ROOT(T) and A_j is in a principal subtree T_p. Then

$$\begin{aligned} q_X(T) &= (x_1 \cdots x_i \cdots x_m - 1)(q_X(T_1) + \cdots + q_X(T_p) \cdots + q_X(T_k)) \\ &> (x-1) q_X(T_p) \\ &> (x-1)^2 \qquad \text{since } q_X(T_p) > (x-1). \end{aligned}$$

Alternatively, suppose that A_i and A_j are both in a principal subtree T_p. Then $q_X(T) > q_X(T_p)$ from the definition of $q_X(T)$ and so $q_X(T) > (x-1)^2$ by the induction hypothesis. The other case, where A_i and A_j are in different principal subtrees cannot occur because $A_i \sim A_j$.

To prove (ii), we first note that one can prove, using a similar method to the one used above in proving $q_X(T) > (x_i - 1)$, that $p_X(T) = B_1 x_i + B_2$, where B_1 and B_2 are arithmetic expressions which do not involve x_i. We next show by induction on HEIGHT(T) that $p_X(T) < C_1 x + C_2$ from which (ii) follows using Lemma 3.14. Consider the case where HEIGHT(T) = 2. Then A_i and A_j must be in different children of the root and so, by Definition 3.12, $p_X(T) = D_1(D_2 x + D_3 x + 1)$ where D_1, D_2, and D_3 are arithmetic expressions which do not contain x and so, by simple rearrangement of terms, the induction hypothesis is satisfied. Assume inductively then that the result holds for all trees up to height k and let T be of height $k+1$.

If A_i and A_j belong to different principal subtrees of T, which without loss of generality we denote by T_i and T_j, respectively, then $p_X(T) = C(p_X(T_i) + p_X(T_j) + D)$, where C and D are expressions not containing x, and so the result follows using the above result that both $p_X(T_i)$ and $p_X(T_j)$ are equal to expressions which are linear in x. Alternatively, if A_i and A_j occur in the same principal subtree of T, then the result follows by the inductive hypothesis and rearranging terms in the expression for $p_X(T)$. ◻

Finally, we complete the proof of Theorem 3.9.

Proof of Theorem 3.9. *If*: This part is immediate.

Only if: We prove the contrapositive that if S and T are not isomorphic, then they are not absolutely equivalent. Let $\mathrm{ATT}(T) = \{A_1, \ldots, A_n\}$. If S and T are not isomorphic, then by Lemma 3.11 there exist attributes A_i and A_j such that, without loss of generality, $A_i \sim A_j$ in S but $A_i \not\sim A_j$ in T. So by Lemma 3.15, $\alpha_X(S) > 2^{(x-1)^2}$ and $2^{C_2 x + C_2} > \alpha_X(T)$. It follows by elementary properties of the exponential function that there exists a constant N such that $2^{(x-1)^2} > 2^{C_2 x + C_2}$ for $x > N$ and so $\alpha_X(S) > \alpha_X(T)$ when $x_k > 1$ for all k, $1 \leq k \leq n$, and $x_i = x_j = x$. This contradicts Lemma 3.5 and so S and T cannot be absolutely equivalent. ∎

We note that Theorem 3.9 is not valid if the scheme trees are not balanced. For example, if one takes scheme trees S and T, where $\mathrm{N}(S) = \{A, \{B\}\}$ and $\mathrm{N}(T) = \{A, B\}$, then it follows from Definition 3.3 and Lemma 3.5 that S and T are absolutely equivalent yet they are not isomorphic. It also follows from this theorem that the scheme trees S and T in Figure 3.1 are not absolutely equivalent since they are balanced but not isomorphic.

We now use Theorem 3.9 to show that the (balanced) normal form tree for T, denoted by $\mathrm{B}(T)$, is unique (up to an isomorphism). We write $S \Rightarrow T$ if there exists a sequence of scheme trees $S_1, \ldots, S_n$ such that $S_1 = S, \ldots, S_n = T$ and $S_i = \mathrm{COMPRESS}(S_{i-1}), n \geq i > 1$.

LEMMA 3.16. *If S is a scheme tree and S_1 and S_2 are balanced scheme trees such that $S \Rightarrow S_1$ and $S \Rightarrow S_2$, then S_1 and S_2 are isomorphic.*

Proof. It suffices to show that $\Rightarrow$ is Church–Rosser, i.e.,

(i) every sequence of COMPRESS operation terminates;

(ii) if $S \Rightarrow S_1$ and $S \Rightarrow S_2$, then there is a pair of isomorphic trees T_1 and T_2 such that $S_1 \Rightarrow T_1$ and $S_2 \Rightarrow T_2$.

Since COMPRESS reduces the number of nodes in the scheme tree by 1, (i) follows immediately. For (ii), it follows from (i) that there exist balanced scheme trees T_1 and T_2 such that $S \Rightarrow S_1 \Rightarrow T_1$ and $S \Rightarrow S_2 \Rightarrow T_2$. Applying Lemmas 3.5 and 3.8 shows that T_1 and T_2 are absolutely equivalent and so they are isomorphic by Theorem 3.9. ∎

To complete the preliminaries needed to prove the main result of the paper, we follow the approach of [29] and define a generalized nest operator and then show that it can be used to construct a bijection between PNF instances of two scheme trees whose induced sets of scheme trees are equivalent.

DEFINITION 3.17. *Let T be a scheme tree and s be a flat relation defined over U. Then nesting s according to T, denoted by $\nu^*_T(s)$, is defined by $\nu^*_T(s) = \nu_{X_k} \cdots \nu_{X_0}(s)$ such that*

(i) *for every nonroot node $n \in T$ there exist one and only one X_i such that $\mathrm{N}(T_n) = X_i$, where T_n is the subtree of T with root n;*

(ii) *if node n is a descendent of node n' in T, then the nest operation on the set corresponding to n is performed before the nest operation on the set corresponding to n'.*

For example, given the scheme tree T in Figure 2.1, then one nest sequence to construct $\nu^*_T(s)$ is $\nu_{\mathrm{MAJOR}}\ \nu_{\mathrm{CLASS},\{\mathrm{EXAM}\},\{\mathrm{PROJECT}\}}\ \nu_{\mathrm{EXAM}}\ \nu_{\mathrm{PROJECT}}(r)$. The following properties of $\nu^*_T(s)$ can be easily shown using the results in [28, 29].

LEMMA 3.18. *If s is a flat relation in $\mathrm{SAT}(\mathrm{MVD}(T))$, then*

(i) *$\nu^*_T(s)$ is uniquely defined;*

(ii) *if $r \in \mathrm{PNF}(T)$, then $\nu^*_T(\mu^*(r)) = r$;*

(iii) $\mu^*(\nu_T^*(s)) = s$.

We use this result to show that if MVD(S) and MVD(T) are equivalent, then there exists a bijection from PNF(S) to PNF(T).

LEMMA 3.19. *If S and T are scheme trees where* ATT(T) = ATT(S) *and* MVD(S) $\equiv$ MVD(T)*, then the mapping from* PNF(S) *to* PNF(T) *defined by* $F(r) = \nu_T^*(\mu^*(r))$*,* $r \in$ PNF(S)*, is a bijection.*

Proof. We shall verify that F has each of the properties of a bijection.

(i) *F is total.* If $r \in$ PNF(S), then $\mu^*(r) \in$ SAT(MVD(S)) (Theorem 7.5 in [18]) and so $\mu^*(r) \in$ SAT(MVD(T)) since MVD(S) $\equiv$ MVD(T) and hence $\nu_T^*(\mu^*(r)) \in$ PNF(T) (again by Theorem 7.5 in [18]).

(ii) *F is onto.* Let $r \in$ PNF(T) and consider the relation r_1 where $r_1 = \nu_S^*(\mu^*(r))$. From the same argument as in (i), $r_1 \in$ PNF(S). By Lemma 3.18(iii), $\mu^*(\nu_S^*(\mu^*(r))) = \mu^*(r)$ and so $F(r_1) = \nu_T^*(\mu^*(\nu_S^*(\mu^*(r)))) = \nu_T^*(\mu^*(r))$, and by Lemma 3.18(ii), $\nu_T^*(\mu^*(r)) = r$ and so F is onto.

(iii) *F is* 1–1. Suppose that there are relations r_1 and $r_2 \in$ PNF(S) such that $F(r_1) = F(r_2)$, i.e., $\nu_T^*(\mu^*(r_1)) = \nu_T^*(\mu^*(r_2))$. Then applying μ^* to both sides and using Lemma 3.18(iii) shows that $\mu^*(r_1) = \mu^*(r_2)$ and applying ν_S^* to both sides and using Lemma 3.18(ii) shows that $r_1 = r_2$ and so F is 1–1. □

Combining these preliminary results now leads to the main result of this paper which provides two characterization of absolute equivalence for nested schemes.

THEOREM 3.20. *If T and S are scheme trees where* ATT(S) = ATT(T)*, then the following statements are equivalent:*

(i) *S and T are absolutely equivalent;*

(ii) B(S) *and* B(T) *are isomrphic;*

(iii) MVD(S) $\equiv$ MVD(T).

Proof. We shall show that (i) $\Rightarrow$ (ii) $\Rightarrow$ (iii) $\Rightarrow$ (i).

(i) $\Rightarrow$ (ii). From Lemmas 3.5 and 3.8, if S and T are absolutely equivalent, then B(S) and B(T) are absolutely equivalent and so B(S) and B(T) are isomorphic by Theorem 3.9.

(ii) $\Rightarrow$ (iii). It follows directly from the definition of a scheme tree isomorphism that if B(S) and B(T) are isomorphic, then MVD(B(S)) = MVD(B(T)) and so it suffices to show that MVD(T') $\equiv$ MVD(COMPRESS(T')), where COMPRESS(T') denoted the tree obtained by any application of the COMPRESS operator to an arbitrary scheme tree T'. This follows immediately from the observation that P(T') = P(COMPRESS(T')) and the result [18] that MVD(T') $\equiv \bowtie$ [P(T')].

(iii) $\Rightarrow$ (i). This is immediate from Lemma 3.19 and noting that for any $r \in$ PNF(S), ACT(r) = ACT($F(r)$). □

Some observations on this theorem and its consequences are appropriate at this point. We first note that a consequence of the equivalence of (i) and (ii) is that the COMPRESS operator (and its inverse) are complete in the sense that two scheme trees are absolutely equivalent if and only if one tree can be obtained from the other by a sequence of applications of the COMPRESS operator and its inverse. Second, from (i) $\Rightarrow$ (iii) and Lemma 3.19 it follows that absolute equivalence implies query equivalence in the context of the nested query language of [28]. So all the types of data equivalence coincide for nested schemes since it is well known that query equivalence $\Rightarrow$ generic equivalence $\Rightarrow$ internal equivalence $\Rightarrow$ absolute equivalence [10].

The theorem also results in two polynomial time algorithms for testing scheme tree equivalence, one by checking for the equivalence of the sets of MVDs and the other by converting each scheme tree to a balanced tree and then checking if the trees are isomorphic. Using the MVD approach, it follows from the results in [6] that the implication problem of testing whether a set of MVDs Σ implies an MVD $Z \rightarrow\rightarrow W$ can be solved in at worst in $\mathrm{O}(N \log |U|)$ time, where N is the total number of occurrences of attributes in Σ. Testing whether two sets of MVDs are equivalent then reduces to running the membership test for each MVD in both sets and can be solved in at worst in $\mathrm{O}(|\Sigma| N \log |U|)$ time, assuming that $|\Sigma|$ and N are upper bounds for both MVD sets.

The alternative approach can be implemented by first imposing an arbitrary total ordering on U and thereafter sorting first the attributes in each node of the scheme tree and then the children of each node according to the first attribute in a child. This sort can be done in at worst in time $t = \Sigma_i (c_i \log c_i + k_i \log k_i)$ where the sum is taken over all the nodes in the tree and c_i and k_i represent the number of attributes in node i of the tree and the number of children of node i, respectively. Using the properties of the log function we can derive $t = \log(\Pi_i c_i^{c_i} k_i^{k_i})$ and so $t \leq \mathrm{O}(|U| \log |U|)$ because $\Sigma_i c_i = |U|$ and $\Sigma_i k_i \leq |U|$. After sorting the scheme tree, compressing them can be done in at worst in $\mathrm{O}(|U|)$ time since there are at most $|U|$ nodes in the tree. Finally, testing if the trees are isomorphic can be done in at worst $\mathrm{O}(|U|)$ time since each attribute needs to be checked at most once and appears only once in a scheme tree. Hence the total cost of testing for absolute equivalence using this method is at worst $\mathrm{O}(|U| \log |U|)$ time. This compares favorably with the MVD approach when $|U| < |\Sigma|\ N$, which one expects normally to hold. The next lemma summarizes this result.

LEMMA 3.21. *Testing whether two PNF scheme trees are absolutely equivalent can be done in* $\mathrm{O}(|U| \log |U|)$ *time.*

4. Related work.

The concept of absolute equivalence and the investigation of necessary and sufficient conditions for absolute equivalence for a nested-type model, called the format model, was first carried out in [11]. The format model uses three data constructors—collection (which corresponds to set construction), composition (which corresponds to tuple construction), and classification (an alternate construction similar to a variant record construction). A set of local restructuring operators was then defined and proven to be complete for preserving absolute equivalence between formats. The format model is a more general model than the nested model used in this paper in a number of aspects. First, the nested model does not use the classification construction; second, the nested model requires that the collection and composition operators alternate, which is not required in the format model; and last, the nested model requires the PNF condition, whereas the format model does not. As a consequence, while the essential problem addressed in this paper and [11] is the same, the results differ because of the different underlying models. In particular, none of the restructuring operators in [11] is equivalent to our COMPRESS operator and the balanced property of a normal form tree in this paper is not a property of the normal form trees in [11]. The work in [11] was later extended in [2], where the problem of restructuring and data equivalence of the format model was extended to include finite attribute domains. Also, restructuring operators which augment the data capacity of a scheme were introduced. Once again, the results in [2] differ from the results in this paper because of the different model assumptions.

The work in [1] examined data equivalence in the context of the VERSO model. The VERSO model is closer to the one used in this paper than those in [2, 11] but there are still two important differences. First, the VERSO model allows empty sets at all levels in a relation, whereas the model used in this paper, based on [28], does not permit empty sets. Second, the data manipulation language in VERSO differs from that of the nested model used in this paper and, in particular, does not include the nest and unnest operators for restructuring. Instead, the effect of restructuring an instance of a VERSO scheme is defined indirectly via a set of flat relations corresponding to the instance. As a result, the restructuring between two arbitrary VERSO schemes defined over the same set of atomic attributes may not be defined and so the VERSO restructuring operator is less powerful than the unnest and nest operators. For instance, given the scheme trees S and T with $\mathrm{N}(S) = \{A, B, C\}$ and $\mathrm{N}(T) = \{A, \{B\}, \{C\}\}$, then a relation defined over S can be transformed into one over T by two applications of the nest operator but S cannot be transformed into T with the VERSO restructuring [10]. As a result, the notion of data equivalence between two schemes in [1] is restricted to those schemes for which the VERSO restructuring operator is defined. In contrast, we place no restrictions on the nested schemes apart from the obvious requirement of them being defined over the same set of atomic attributes. It is interesting to then compare the results on data equivalence in [1] with ours. Their main result shows that two VERSO schemes are data equivalent if and only if they are identical apart from the trivial restructuring operations of permuting the order of attributes within a node of the scheme tree or permuting the left-to-right ordering of the children of a node. This is a more negative result than ours which allows nontrivial restructuring of the scheme tree using the COMPRESS operator and its inverse. The other difference between the work in [1] and ours is that they consider the issue of data dominance, a more general notion than data equivalence, whereas we have only considered data equivalence.

A different aspect to restructuring PNF relations was investigated in [8]. The motivation for the work was based on the observation that if empty sets are permitted, then restructuring using the nest and unnest operators may result in losing more information than is necessary. A restructuring operator was then defined which maps directly from one nested scheme to another without the use of the nest and unnest operator, using a similar but more general approach to that used in [1], with the aim of minimizing information loss in the restructuring. However, this paper did not address the question of characterizing when the new restructuring operator preserved information content and this was posed as an open problem.

In [12] two restructuring operations were considered in the context of nested relations not necessarily in PNF. The first operation, called *empty node insertion*, allows the insertion of a node with the empty set of attributes between two other nodes in the path of a scheme tree. Empty node insertion preserves absolute equivalence but is not considered herein, since nodes having an empty set of attributes are disallowed. The motivation for allowing such nods in [12] is the restructuring of a forest of scheme trees so that they be joinable. The second operation, called *root weighting*, is a generalization of the COMPRESS operator which allows us to remove a node in the scheme tree and add its attributes to one of its ancestor nodes. As a result of Theorem 3.20 it is easy to verify that any root weighting not corresponding to an application of the COMPRESS operator does not preserve absolute equivalence, resulting in dominance instead.

5. Conclusion.

In this paper we have investigated the problem of when two nested relational schemes are data equivalent and provided two characterizations of data equivalence. The first is that two nested schemes are data equivalent if and only if the sets of MVDs induced by the corresponding scheme trees are equivalent. The second is that two schemes are data equivalent if and only if the corresponding scheme trees can be transformed into each other by a sequence of applications of a tree restructuring operator called COMPRESS (and its inverse). The COMPRESS operator merges a parent and child nodes in a scheme tree if the parent has only one child node. We also used these characterizations to derive two polynomial time algorithms for determining when two nested schemes are data equivalent.

There are several other related topics that also warrant further investigation. Data equivalence can be viewed as a special case of data dominance [8] which holds if there is 1–1 (but not necessarily onto) mapping from the instances of one scheme to the instances of another scheme. Then finding a complete set of restructuring operators for a scheme tree which preserve data dominance is an important question that arises in contexts such as schema integration where one often wants to ensure that the integrated scheme dominates the original schemes. The approach in this paper has also assumed that attribute domains are disjoint and it would be useful to characterize data equivalence and dominance if this assumption is dropped. Another aspect that needs further investigation is to extend the approach in this paper, which has characterized data equivalence for only a single nested scheme, to characterizing absolute equivalence and dominance for nested database schemes.

Acknowledgments. Thanks to the anonymous referees whose constructive comments helped to improve the presentation of this paper.

REFERENCES

[1] S. Abiteboul and N. Bidoit, *Non first normal form relations: An algebra allowing data restructuring*, J. Comput. System Sci., 33 (1986), pp. 361–393.
[2] S. Abiteboul and R. Hull, *Restructuring hierarchical database objects*, Theoret. Comput. Sci., 62 (1988), pp. 3–38.
[3] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison–Wesley, Reading, MA, 1995.
[4] S. Abiteboul, M. Scholl, G. Gardarin, and E. Simon, *Towards DBMSs for supporting new applications*, in 12th International Conference on Very Large Data Bases, Kyoto, Japan, 1986, pp. 423–435.
[5] E. F. Codd, *A relational model of data for large shared data banks*, Comm. ACM, 13 (1970), pp. 377–387.
[6] Z. Galil, *An almost linear-time algorithm for computing a dependency basis in a relational database*, J. ACM, 29 (1982), pp. 96–102.
[7] M. Gyssens, J. Paredaens, and D. Van Gucht, *On a hierarchy of classes for nested databases*, Inform. Process. Lett., 36 (1990), pp. 259–266.
[8] G. Hulin, *On restructuring nested relations in partitioned normal form*, in 16th International Conference on Very Large Data Bases, Brisbane, Australia, 1990, pp. 626–637.
[9] R. Hull, *Relative information capacity of simple relational database schemata*, SIAM J. Comput., 15 (1986), pp. 856–886.
[10] R. Hull, *A survey of theoretic research on typed complex database objects*, in Databases, J. Paredaens, ed., Academic Press, London, 1987, pp. 193–256.
[11] R. Hull and C. K. Yap, *The format model: A theory of database organization*, J. ACM, 31 (1984), pp. 518–537.
[12] M. Levene, *The Nested Universal Relational Database Model*, Springer-Verlag, Berlin, 1992.
[13] M. Levene and G. Loizou, *NURQL: A nested universal relation query language*, Information Syst., 14 (1989), pp. 307–316.
[14] A. Makinouchi, *A consideration on normal form of not-necessarily-normalized relations in the relational data model*, in 3rd International Conference on Very Large Data Bases, Tokyo, Japan, 1977, pp. 447–453.
[15] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan, *Schema equivalence in heterogenous systems: Bridging theory and practice*, Information Syst., 19 (1994), pp. 3–13.
[16] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan, *The use of information capacity in schema integration and translation*, in 19th International Conference on Very Large Data Bases, Dublin, Ireland, 1993, pp. 120–133.

[17] Z. M. Ozsoyoglu and L.-Y. Yuan, *A new normal form for nested relations*, ACM Trans. Database System, 12 (1987), pp. 111–136.

[18] J. Paredaens, P. DeBra, M. Gyssens, and D. Van Gucht, *The Structure of the Relational Database Model*, Springer-Verlag, Berlin, 1989.

[19] J. Paredaens and D. Van Gucht, *Converting nested algebra expressions into flat algebra expressions*, ACM Trans. Database Systems, 17 (1992), pp. 65–93.

[20] P. Pistor, *The advanced information prototype (AIM-P): Advanced database technology for integrated applications*, IBM Systems J., 28 (1987), pp. 661–681

[21] M. A. Roth, H. F. Korth, and A. Silberschatz, *Extended algebra and calculus for nested relational databases*, ACM Trans. Database Systems, 13 (1988), pp. 389–417

[22] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, and J. Zobel, *ATLAS: A Nested Relational Database System for Text Applications*, Technical report CITRI/TR-92-52, Departments of Computer Science, RMIT and The University of Melbourne, Australia, 1992.

[23] H.-J. Schek, H. B. Paul, M. Scholl, and G. Weikum, *The DASDBS project: Objectives, experiences, and future prospects*, IEEE Transactions on Knowledge and Data Engineering, 2 (1990), pp. 25–43.

[24] H.-J. Schek and M. H. Scholl, *The relational model with relation-valued Attributes*, Information Syst., 11 (1986), pp. 137–147.

[25] H.-J. Schek and W. Waterfeld, *A database kernel system for geo-scientific applications*, in Symposium on Spatial Data Handling, 1986.

[26] M. Scholl, *VERSO: A database machine based on nested relations*, in Nested Relations and Complex Objects in Databases, S. Abiteboul, P. C. Fischer, and H.-J. Schek, eds., Springer-Verlag, Berlin, 1989, pp. 27–49.

[27] M. Stonebraker and D. Moore, *Object-Relational DBMSs*, Morgan–Kauffman, San Francisco, CA, 1996.

[28] S. J. Thomas and P. C. Fischer, *Nested relational structures*, in the Theory of Databases, P. C. Kanellakis, ed., JAI Press, Greenwich, CT, 1986, pp. 269–307.

[29] D. Van Gucht, *Multilevel nested relational structures*, J. Comput. System Sci., 36 (1988), pp. 77–105.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1568–1576

LINEAR-TIME APPROXIMATION ALGORITHMS FOR COMPUTING NUMERICAL SUMMATION WITH PROVABLY SMALL ERRORS*

MING-YANG KAO† AND JIE WANG‡

Abstract. Given a multiset $X = \{x_1, \ldots, x_n\}$ of real numbers, the *floating-point set summation* problem asks for $S_n = x_1 + \cdots + x_n$. Let E_n^* denote the minimum worst-case error over all possible orderings of evaluating S_n. We prove that if X has both positive and negative numbers, it is NP-hard to compute S_n with the worst-case error equal to E_n^*. We then give the first known polynomial-time approximation algorithm that has a provably small error for arbitrary X. Our algorithm incurs a worst-case error at most $2(\lceil \log(n-1) \rceil + 1)E_n^*$. (All logarithms log in this paper are base 2.) After X is sorted, it runs in $O(n)$ time. For the case where X is either all positive or all negative, we give another approximation algorithm with a worst-case error at most $\lceil \log \log n \rceil E_n^*$. Even for unsorted X, this algorithm runs in $O(n)$ time. Previously, the best linear-time approximation algorithm had a worst-case error at most $\lceil \log n \rceil E_n^*$, while E_n^* was known to be attainable in $O(n \log n)$ time using Huffman coding.

Key words. floating-point summation, error analysis, addition trees, combinatorial optimization, NP-hardness, approximation algorithms

AMS subject classifications. 65G05, 65B10, 68Q15, 68Q25, 68R05

PII. S0097539798341594

1. Introduction.

Summation of floating-point numbers is ubiquitous in numerical analysis and has been extensively studied (for example, see [2, 4, 5, 6, 7, 8, 10, 11, 12]). This paper focuses on the *floating-point set summation* problem which, given a multiset $X = \{x_1, \ldots, x_n\}$ of real numbers, asks for $S_n = x_1 + x_2 + \cdots + x_n$. Without loss of generality, let $x_i \neq 0$ for all i throughout the paper. Here X may contain both positive and negative numbers. For such a general X, previous studies have discussed heuristic methods and obtained statistical or empirical bounds for their errors. We take a new approach by designing efficient algorithms whose worst-case errors are provably small.

Our error analysis uses the standard model of floating-point arithmetic with unit roundoff $\alpha \ll 1$:

$$\mathrm{fl}(x + y) = (x + y)(1 + \delta_{xy}), \quad \text{where } |\delta_{xy}| \leq \alpha.$$

Since the operator $+$ is applied to two operands at a time, an ordering for adding X corresponds to a binary addition tree of n leaves and $n - 1$ internal nodes, where a leaf is an x_i and an internal node is the sum of its two children. Different orderings yield different addition trees, which may produce different computed sums $\hat{S}_n$ in floating-point arithmetic. We aim to find an optimal ordering that minimizes the

*Received by the editors July 6, 1998; accepted for publication (in revised form) February 24, 1999; published electronically March 15, 2000. A preliminary version of this work appeared as part of *Efficient minimization of numerical summation errors*, in Proceedings of the 25th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 1443, K. G. Larsen, S. Skyum, and G. Winskel, eds., Springer-Verlag, New York, NY, 1998, pp. 375–386.
http://www.siam.org/journals/sicomp/29-5/34159.html

†Department of Computer Science, Yale University, New Haven, CT 06520 (kao@cs.yale.edu). This author was supported in part by NSF grant CCR-9531028.

‡Department of Mathematical Sciences, University of North Carolina at Greensboro, Greensboro, NC 27412 (wang@uncg.edu). This author was supported in part by NSF grant CCR-9424164.

error $E_n = |\hat{S}_n - S_n|$. Let $I_1, \ldots, I_{n-1}$ be the internal nodes of an addition tree T over X. Since α is very small even on a desktop computer, any product of more than one α is negligible in our consideration. Using this approximation,

$$\hat{S}_n \approx S_n + \sum_{i=1}^{n-1} I_i \delta_i.$$

Hence $E_n \approx |\sum_{i=1}^{n-1} I_i \delta_i| \leq \alpha \sum_{i=1}^{n-1} |I_i|$, giving rise to the following definitions.

- The *worst-case error* of T, denoted by $E(T)$, is $\alpha \sum_{i=1}^{n-1} |I_i|$.
- The *cost* of T, denoted by $C(T)$, is $\sum_{i=1}^{n-1} |I_i|$.

Our task is to find a fast algorithm that constructs an addition tree T over X such that $E(T)$ is small. Since $E(T) = \alpha \cdot C(T)$, minimizing $E(T)$ is equivalent to minimizing $C(T)$. We further adopt the following notations.

- E_n^* (respectively, C_n^*) is the minimum worst-case error (respectively, minimum cost) over all orderings of evaluating S_n.
- $T_{\min}$ denotes an optimal addition tree over X, i.e., $E(T_{\min}) = E_n^*$ or, equivalently, $C(T_{\min}) = C_n^*$.

In section 2, we prove that if X contains both positive and negative numbers, it is NP-hard to compute a $T_{\min}$. In light of this result, we design an approximation algorithm in section 3.1 that computes a tree T with $E(T) \leq 2(\lceil \log(n-1) \rceil + 1)E_n^*$. After X is sorted, this algorithm takes only $O(n)$ time. This is the first known polynomial-time approximation algorithm that has a provably small error for arbitrary X. For the case where X is either all positive or all negative, we give another approximation algorithm in section 3.2 that computes a tree T with $E(T) \leq (1 + \lceil \log \log n \rceil)E_n^*$. This algorithm takes only $O(n)$ time even for unsorted X. Previously [5], the best linear-time approximation algorithm had a worst-case error at most $\lceil \log n \rceil E_n^*$, while E_n^* was known to be attainable in $O(n \log n)$ time using Huffman coding [9].

2. Minimizing the worst-case error is NP-hard.

If X contains both positive and negative numbers, we prove that it is NP-hard to find a $T_{\min}$. We first observe the following properties of $T_{\min}$.

LEMMA 2.1. *Let z be an internal node in $T_{\min}$ with children z_1 and z_2, sibling u, and parent r.*

1. *If $z > 0$, $z_1 \geq 0$, and $z_2 > 0$, then $u \geq 0$ or $r < 0$.*
2. *If $z < 0$, $z_1 \leq 0$, and $z_2 < 0$, then $u \leq 0$ or $r > 0$.*

Proof (By symmetry). We prove only the first statement. $C(T_r) = |r| + |z| + C_f$, where $C_f = C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. Assume to the contrary that $u < 0$ and $r \geq 0$. Then $z \geq |u|$. We swap T_{z_1} with T_u. Let $z' = u + z_2$. Now r becomes the parent of z' and z_1. This rearrangement of nodes does not affect the value of node r, and the costs of T_{z_1}, T_{z_2}, and T_u remain unchanged. Let T'_r be the new subtree with root r. Let T' be the entire new tree from the swapping. Since u and z_2 have the opposite signs, $|z'| < \max\{|u|, z_2\} \leq z$. Hence, $C(T'_r) = r + |z'| + C_f < r + z + C_f = C(T_r)$. Thus, $C(T') < C(T_{\min})$, contradicting the optimality of $T_{\min}$. This completes the proof. $\square$

For the purpose of proving that finding a $T_{\min}$ is NP-hard, we restrict all x_i to nonzero integers and consider the following optimization problem.

MINIMUM ADDITION TREE (MAT).

Input: A multiset X of n nonzero integers $x_1, \ldots, x_n$.

Output: Some $T_{\min}$ over X.

The following problem is a decision version of MAT.

ADDITION TREE (AT).

Instance: A multiset X of n nonzero integers $x_1, \dots, x_n$, and an integer $k \geq 0$.

Question: Does there exist an addition tree T over X with $C(T) \leq k$?

LEMMA 2.2. *If* MAT *is solvable in time polynomial in n, then* AT *is also solvable in time polynomial in n.*

Proof. The proof is straightforward. □

In light of Lemma 2.2, to prove that MAT is NP-hard, it suffices to reduce the following NP-complete problem [3] to AT.

3-PARTITION (3PAR).

Instance: A multiset B of $3m$ positive integers $b_1, \dots, b_{3m}$, and a positive integer K such that $K/4 < b_i < K/2$ and $b_1 + \cdots + b_{3m} = mK$.

Question: Can B be partitioned into m disjoint sets $B_1, \dots, B_m$ such that for each B_i, $\sum_{b \in B_i} b = K$? (B_i must therefore contain exactly three elements from B.)

Given an instance (B, K) of 3PAR, let

$$W = 100(5m)^2K; \qquad a_i = b_i + W; \qquad A = \{a_1, \dots, a_{3m}\}; \qquad L = 3W + K.$$

LEMMA 2.3. *(A, L) is an instance of* 3PAR. *Furthermore, it is a positive instance if and only if (B, K) is a positive instance.*

Proof. Since $K/4 < b_i < K/2$, $K/4 + W < a_i < K/2 + W$, and thus $L/4 < a_i < L/2$. Next, $a_1 + a_2 + \cdots + a_{3m} = 3mW + mK = mL$. This completes the proof of the first statement. The second statement follows from the fact that $b_i + b_j + b_k = K$ if and only if $a_i + a_j + a_k = L$. □

Write

$$\epsilon = \frac{1}{400(5m)^2}; \qquad h = \lfloor 4\epsilon L \rfloor; \qquad H = L + h;$$

$$h = \beta_0 H; \quad a_i = \left(\frac{1}{3} + \beta_i\right) H; \quad a_i = \left(\frac{1}{3} + \epsilon_i\right) L; \quad a_M = \max\{a_i : i = 1, \dots, 3m\}.$$

LEMMA 2.4.

1. $|\epsilon_i| < \epsilon$ *for* $i = 1, \dots, 3m$.
2. $0 < \beta_0 < 4\epsilon$, *and* $|\beta_i| < 4\epsilon$ *for* $i = 1, \dots, 3m$.
3. $3a_M < H$.

Proof. Statement 1. Note that $b_i + W = (1/3 + \epsilon_i)(3W + K)$. Thus, $b_i = K/3 + \epsilon_i(300(5m)^2 + 1)K$. Since $K/4 < b_i < K/2$, $-1/12 < \epsilon_i(300(5m)^2 + 1) < 1/6$. Hence $4(5m)^2|\epsilon_i| < 10^{-2}$, i.e., $|\epsilon_i| < \epsilon$.

Statement 2. Since $4\epsilon L > 1$, we have $\beta_0 > 0$. Also, since $H > L$ and $\beta_0 H = \lfloor 4\epsilon L \rfloor$, we have $\beta_0 < 4\epsilon$. Next, for each a_i, we have $\beta_i = (\epsilon_i L - h/3)/(L + h)$. Then by the triangular inequality and Statement 1, $|\beta_i| < 7\epsilon/3 < 4\epsilon$.

Statement 3. By Statement 1, $a_i < (1/3 + \epsilon)L$. Thus $3a_M < L + 3\epsilon L$. Then, since $3\epsilon L < 3K \leq h$, $3a_M < L + h = H$. □

To reduce (A, L) to an instance of AT, we consider a particular multiset

$$X = A \cup \{-H, \dots, -H\} \cup \{h, \dots, h\}$$

with m copies of $-H$ and h each. Given a node s in $T_{\min}$, let T_s denote the subtree rooted at s. For convenience, also let s denote the value of node s. Let $v(T_{\min})$ denote the value of the root of $T_{\min}$, which is always 0. For brevity, we use λ with or without scripts to denote the sum of at most $5m$ numbers in the form of $\pm\beta_i$. Then all nodes are in the form of $(N/3 + \lambda)H$ for some integer N and some λ. Since by Lemma 2.4,

$|\lambda| \le (5m)(4\epsilon) = (500m)^{-1}$, the terms N and λ of each node are uniquely determined. The nodes in the form of λH are called the *type*-0 nodes. Note that $T_{\min}$ has m type-0 leaves, i.e., the m copies of h in X.

LEMMA 2.5. *In $T_{\min}$, type-0 nodes can only be added to type-0 nodes.*

Proof. Assume to the contrary that a type-0 node z_1 is added to a node z_2 in the form of $(\pm N/3 + \lambda)H$ with $N \ge 1$. Then $|z_1 + z_2| \ge (1/3 + \lambda')H$ for some λ'. Let z be the parent of z_1 and z_2. Since $v(T_{\min}) = 0$, z cannot be the root of $T_{\min}$. Let u be the sibling of z. Let r be the parent of z and u. Let t be the root of $T_{\min}$. Let P_r be the path from t to r in $T_{\min}$. Let m_r be the number of nodes on P_r. Since $T_{\min}$ has $5m - 1$ internal nodes, $m_r < 5m - 1$.

We rearrange $T_{\min}$ to obtain a new tree T' as follows. First, we replace T_z with T_{z_2}; i.e., r now has subtrees T_{z_2} and T_u. Let T'' be the remaining tree; i.e., T'' is $T_{\min}$ after removing T_{z_1}. Next, we create T' such that its root has subtrees T_{z_1} and T''. This tree rearrangement eliminates the cost $|z_1 + z_2|$ from T_r but may result in a new cost in the form of λH on each node of P_r. The total of these extra costs, denoted by C_λ, is at most $m_r(5m)(4\epsilon)H < (5m-1)(5m)(4\epsilon)H$. Then, $C(T') = C(T_{\min}) - |z_1 + z_2| + C_\lambda \le C(T_{\min}) - (1/3 + \lambda')H + C_\lambda < C(T_{\min}) + (-1/3 + (5m)^2(4\epsilon))H = C(T_{\min}) + (-1/3 + 10^{-2})H < C(T_{\min})$, contradicting the optimality of $T_{\min}$. This completes the proof. ∎

LEMMA 2.6. *Let z be a node in $T_{\min}$.*

1. *If $z < 0$, then $|z| \le H$.*
2. *If $z > 0$, then $z < H$.*

Proof. Statement 1. Assume that the statement is untrue. Then, since all negative leaves have values $-H$, some negative internal node z has an absolute value greater than H and two negative children z_1 and z_2. Since $v(T_{\min}) = 0$, some z has a positive sibling u. We pick such a z at the lowest possible level of $T_{\min}$. Let r be the parent of z and u. By Lemma 2.1(1), $r > 0$. Then $u > |z| > H$. Since all positive leaves have values less than H, u is an internal node with two children u_1 and u_2. Since $u > 0$, $z < 0$, and $r > 0$, by Lemma 2.1(1), u must have a positive child and a negative child. Without loss of generality, let u_1 be positive and u_2 be negative. Then $u = u_1 - |u_2|$. Since z is at the lowest possible level, $|u_2| \le H$, for otherwise we could find a z at a lower level under u_2. We swap T_z with T_{u_2}. Let T'_r be the new subtree rooted r. Let $u' = u_1 + z$. Since $u_2 + u' = r > 0$ and $u_2 < 0$, we have $u' > 0$. Since $|u_2| \le H < |z|$, we have $u' = u_1 - |z| < u_1 - |u_2| = u$. Let $C_f = C(T_z) + C(T_{u_1}) + C(T_{u_2})$. Then $C(T'_r) = r + u' + C_f < r + u + C_f = C(T_r)$, which contradicts the optimality of $T_{\min}$ because the costs of the internal nodes not mentioned above remain unchanged.

Statement 2. Assume that this statement is false. Then, since all positive leaves have values less than H, some internal node z has a value of at least H as well as two positive children. Since $v(T_{\min}) = 0$, some such z has a negative sibling u. By Statement 1, $|u| \le H$. Hence $z + u \ge 0$, contradicting Lemma 2.1(1). ∎

The following lemma strengthens Lemma 2.6.

LEMMA 2.7.

1. *Let z be a node in $T_{\min}$. If $z > 0$, then z is in the form of λH, $(1/3 + \lambda)H$, or $(2/3 + \lambda)H$.*
2. *Let z be an internal node in $T_{\min}$. If $z < 0$, then z is in the form of λH, $(-1/3 + \lambda)H$, or $(-2/3 + \lambda)H$.*

Proof. Statement 1. By Lemma 2.6, $z < H$. Thus $z = (N/3 + \lambda)H$ with $0 \le N \le 3$. To rule out $N = 3$ by contradiction, assume $z = (1 + \lambda)H$ with $\lambda < 0$. Since by Lemma 2.4 all positive leaves have values less than $(1/3 + 4\epsilon)H$, z is an internal node. By Lemmas 2.5 and 2.6, z has two children $z_1 = (2/3 + \lambda')$ and $z_2 = (1/3 + \lambda'')$. Since $v(T_{\min}) = 0$, z is not the root, and by Lemmas 2.5 and 2.6, z has a negative sibling u. By Lemma 2.6, $|u| \le H$. Let r be the parent of z and u. Then $C(T_r) = |r| + z + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. We swap T_{z_2} with T_u. Let z' be the parent of z_1 and u. Now r is the parent of z' and u. Let T'_r be the new subtree rooted at r after the swapping. Since r remains the same, $C(T'_r) = |r| + |z'| + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. If $|u| \ge z_1$, then $|z'| = |u| - z_1 \le H - z_1 = (1/3 - \lambda')H < z_1 < z$; otherwise, $|u| < z_1$ and thus $|z'| = z_1 - |u| < z_1 < z$. In either case, $C(T'_r) < C(T_r)$, contradicting the optimality of $T_{\min}$.

Statement 2. The proof is similar to that of Statement 1. By Lemma 2.6, $z = (-N/3 + \lambda)H$ with $0 \le N \le 3$. To rule out $N = 3$ by contradiction, assume $z = (-1+\lambda)H$ with $\lambda < 0$. By Lemmas 2.5 and 2.6, z has a positive sibling $u < H$ and two children $z_1 = (-2/3 + \lambda')H$ and $z_2 = (-1/3 + \lambda'')H$. Let r be the parent of z and u. Then $C(T_r) = |r| + |z| + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. We swap T_{z_2} with T_u. Let z' be the parent of z_1 and u. Now r is the parent of z' and u. Let T'_r be the new subtree rooted at r after the swapping. Since r is the same, $C(T'_r) = |r| + |z'| + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. If $u \ge |z_1|$, then $|z'| = u - |z_1| < (1/3 - \lambda')H < |z|$; otherwise, $u < |z_1|$ and thus $|z'| = |z_1| - u < |z_1| < |z|$. So $C(T'_r) < C(T_r)$, contradicting the optimality of $T_{\min}$. ∎

The following lemma supplements Lemma 2.7(1).

Lemma 2.8. *Let z be a node in $T_{\min}$. If $z = (1/3 + \lambda)H$, then z is a leaf.*

Proof. Assume to the contrary that $z = (1/3 + \lambda)H$ is not a leaf. By Lemmas 2.5 and 2.7, z has two children $z_1 = (2/3 + \lambda_1)H$ and $z_2 = (-1/3 + \lambda_2)H$. By Lemmas 2.5 and 2.7, z_1 has two children $z_3 = (1/3 + \lambda_3)H$ and $z_4 = (1/3 + \lambda_4)H$, contradicting Lemma 2.1(1). ∎

The following lemma strengthens Lemma 2.7(2).

Lemma 2.9. *Let z be an internal node in $T_{\min}$. If $z < 0$, then z can be only in the form of λH or $(-1/3 + \lambda)H$.*

Proof. To prove the lemma by contradiction, by Lemma 2.7, we assume $z = (-2/3 + \lambda)H$. Let z_1 and z_2 be the two children of z. Let u be the sibling of z; by Lemmas 2.5 and 2.7, $u = (2/3 + \lambda')H$ or $(1/3 + \lambda')H$. Let r be the parent of z and u. Then $C(T_r) = |r| + |z| + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. By Lemmas 2.5 and 2.7, there are two cases based on the values of z_1 and z_2 with the symmetric cases omitted.

Case 1. $z_1 = (-1/3 + \lambda_1)H$ and $z_2 = (-1/3 + \lambda_2)H$. Swap T_u with T_{z_2}. Let z' be the new parent of z_1 and u. Then r is the parent of z' and u. Let T'_r be the new subtree rooted at r. Then $C(T'_r) = |r| + |z'| + C(T_{z_1}) + C(T_{z_2}) + C(T_u)$. Whether $u = (2/3 + \lambda')H$ or $(1/3 + \lambda')H$, we have $|z'| < |z|$ and thus $C(T'_r) < C(T_r)$, which contradicts the optimality of $T_{\min}$.

Case 2. $z_1 = (1/3 + \lambda_1)H$ and $z_2 = -H$. There are two subcases based on u.

Case 2A. $u = (2/3 + \lambda')H$. We swap T_{z_1} with T_u. Let z' be the new parent of z_2 and u. Then $|z'| < |z|$.

Case 2B. $u = (1/3 + \lambda')H$. We swap T_{z_2} with T_u. Let z' be the new parent of z_1 and u. By Lemma 2.8, both z_1 and u are leaves, and thus by Lemma 2.4, $2z_1 + u < H$. Therefore, $|z'| = z_1 + u < H - z_1 = |z|$.

Therefore, in either subcase of Case 2, the swapping results in an addition tree over X with a smaller cost than $T_{\min}$, reaching a contradiction. ∎

Lemma 2.10. *$C(T_{\min}) \ge m(H + h)$. Moreover, $C(T_{\min}) = m(H + h)$ if and only if (A, L) is a positive instance of* 3PAR .

Proof. By Lemmas 2.5, 2.7, 2.8, and 2.9, each $a_i \in A$ can be added only to some $a_j \in A$ or to some $z_1 = (-1/3 + \lambda_1)H$. In turn, z_1 can be only the sum of $-H$ and some $z_2 = (2/3 + \lambda_2)H$. In turn, z_2 is the sum of some a_k and $a_\ell \in A$. Hence in $T_{\min}$, $2m$ leaves in A are added in pairs. The sum of each pair is then added to a leaf node $-H$. This sum is then added to a leaf node in A. This sum is a type-0 node with value $-|\lambda'|H$, which can be added only to another type-0 node. Let $a_{p,1}, a_{p,2}$, and $a_{p,3}$ be the three leaves in A associated with each $-H$ and added together as $((a_{p,1}+a_{p,2})+(-H))+a_{p,3}$ in $T_{\min}$. The cost of such a subtree is $2H-(a_{p,1}+a_{p,2}+a_{p,3})$. There are m such subtrees R_p. Their total cost is $2mH - \sum_{i=1}^{3m} a_i = mH+mh$. Hence $C(T_{\min}) \geq mH + mh$.

If (A, L) is not a positive instance of 3PAR, then for any $T_{\min}$, there is some subtree R_p with $a_{p,1} + a_{p,2} + a_{p,3} \neq L$. Then, the value of the root r_i of R_p is $a_{p,1} + a_{p,2} + a_{p,3} - H \neq -h$. Since r_i is a type-0 node, it can be added only to a type-0 node. No matter how the m root values r_k and the m leaves h are added, some node resulting from adding these $2m$ numbers is nonzero. Hence $C(T_{\min}) > mH + mh$.

If (A, L) is a positive instance of 3PAR, let $\{a_{p,1}, a_{p,2}, a_{p,3}\}$ with $1 \leq p \leq m$ form a 3-set partition of A; i.e., A is the union of these m 3-sets and for each p, $a_{p,1} + a_{p,2} + a_{p,3} = L$. Then each 3-set can be added to one $-H$ and one h as $(((a_{p,1} + a_{p,2}) + (-H)) + a_{p,3}) + h$, resulting in a node of value zero and contributing no extra cost. Hence $C(T_{\min}) = mH + mh$. This completes the proof. ∎

THEOREM 2.11. *It is NP-hard to compute an optimal addition tree over a multiset that contains both positive and negative numbers.*

Proof. By Lemma 2.2, it suffices to construct a reduction f from 3PAR to AT. Let $f(B, K) = (X, mH + mh)$, which is polynomial-time computable. By Lemma 2.10, $(X, mH + mh)$ is a positive instance of AT if and only if (A, L) is a positive instance of 3PAR. Then, by Lemma 2.3, f is a desired reduction. ∎

3. Approximation algorithms.

In light of Theorem 2.11, for X with both positive and negative numbers, no polynomial-time algorithm can find a $T_{\min}$ unless P = NP [3]. This motivates the consideration of approximation algorithms.

3.1. Linear-time approximation for general X.

This section assumes that X contains at least one positive number and one negative number. We give an approximation algorithm whose worst-case error is at most $2(\lceil \log(n-1) \rceil + 1)E_n^*$. If X is sorted, this algorithm takes only $O(n)$ time.

In an addition tree, a leaf is *critical* if its sibling is a leaf with the opposite sign. Note that if two leaves are siblings, then one is critical if and only if the other is critical. Hence an addition tree has an even number of critical leaves.

LEMMA 3.1. *Let T be an addition tree over X. Let $y_1, \ldots, y_{2k}$ be its critical leaves, where y_{2i-1} and y_{2i} are siblings. Let $z_1, \ldots, z_{n-2k}$ be the noncritical leaves. Let $\Pi = \sum_{i=1}^{k} |y_{2i-1} + y_{2i}|$ and $\Delta = \sum_{j=1}^{n-2k} |z_j|$. Then $C(T) \geq (\Pi + \Delta)/2$.*

Proof. Let x be a leaf in T. There are two cases.

Case 1. x is some critical leaf y_{2i-1} or y_{2i}. Let r_i be the parent of y_{2i-1} and y_{2i} in T for $1 \leq i \leq k$. Then $|r_i| = |y_{2i-1} + y_{2i}|$.

Case 2. x is some noncritical leaf z_j. Let w_j be the sibling of z_j in T. Let q_j be the parent of z_j and w_j. There are three subcases.

Case 2A. w_j is also a leaf. Since z_j is noncritical, w_j has the same sign as z_j and is also a noncritical leaf. Thus $|q_j| = |z_j| + |w_j|$.

Case 2B. w_j is an internal node with the same sign as z_j. Then $|q_j| \geq |z_j|$.

Case 2C. w_j is an internal node with the opposite sign to z_j. If $|w_j| \geq |z_j|$, then $|q_j| + |w_j| \geq |z_j|$; if $|w_j| < |z_j|$, then $|q_j| + |w_j| = |z_j|$. So, we always have $|q_j| + |w_j| \geq |z_j|$.

Observe that

$$C(T) \geq \sum_{i=1}^{k} |r_i| + \frac{1}{2}\left(\sum_{z_j \text{ in Case 2A}} |q_j|\right) + \sum_{z_j \text{ in Case 2B}} |q_j| + \sum_{z_j \text{ in Case 2C}} |q_j|;$$

$$C(T) \geq \sum_{z_j \text{ in Case 2C}} |w_j|.$$

Simplifying the sum of these two inequalities based on the case analysis, we have $2C(T) \geq \Pi + \Delta$ as desired. $\square$

In view of Lemma 3.1, we desire to minimize $\Pi + \Delta$ over all possible T. Given $x_t, x_{t'} \in X$ with $t \neq t'$, $(x_t, x_{t'})$ is a *critical pair* if x_t and $x_{t'}$ have the opposite signs. A *critical matching* R of X is a set $\{(x_{t_{2i-1}}, x_{t_{2i}}) : i = 1, \ldots, k\}$ of critical pairs where the indices t_j are all distinct. For simplicity, let $y_j = x_{t_j}$. Let $\Pi = \sum_{i=1}^{k} |y_{2i-1} + y_{2i}|$ and $\Delta = \sum_{z \in X - \{y_1, \ldots, y_{2k}\}} |z|$. If $\Pi + \Delta$ is the minimum over all critical matchings of X, then R is called a *minimum critical matching* of X. Such an R can be computed as follows. Assume that X consists of ℓ positive numbers $a_1 \leq \cdots \leq a_\ell$ and m negative numbers $-b_1 \geq \cdots \geq -b_m$.

Algorithm 1.

```
1. If $\ell = m$, let $R = \{(a_i, -b_i) : i = 1, \ldots, \ell\}$.
2. If $\ell < m$, let $R = \{(a_i, -b_{i+m-\ell}) : i = 1, \ldots, \ell\}$.
3. If $\ell > m$, let $R = \{(a_{i+\ell-m}, -b_i) : i = 1, \ldots, m\}$.
```

Lemma 3.2. *If X is sorted, then Algorithm 1 computes a minimum critical matching R of X in $O(n)$ time.*

Proof. By case analysis, if $a_i \leq a_j$ and $b_{i'} \leq b_{j'}$, then $|a_i - b_{i'}| + |a_j - b_{j'}| \leq |a_i - b_{j'}| + |a_j - b_{i'}|$. Thus if $\ell = m$, then pairing a_i with $-b_i$ returns the minimum $\Pi + \Delta$. For the case $\ell < m$, let ϵ be an infinitesimally small positive number. Let X' be X with additional $m - \ell$ copies of ϵ. Then $\sum_{i=1}^{\ell} |a_i - b_{i+m-\ell}| + \sum_{i=1}^{m-\ell} |\epsilon - b_i| = (\ell - m)\epsilon + \Pi + \Delta$ is the minimum over all possible critical matchings of X'. Thus $\Pi + \Delta$ is the minimum over all possible critical matching of X. The case $\ell > m$ is symmetric to the case $\ell < m$. Since X is sorted, the running time of Algorithm 1 is $O(n)$. $\square$

We now present an approximation algorithm to compute the summation over X.

Algorithm 2.

```
1. Use Algorithm 1 to find a minimum critical matching $R$ of $X$. The numbers
   $x_i$ in the pairs of $R$ are the critical leaves in our addition tree over $X$, and
   those not in the critical pairs are the noncritical leaves.
2. Add each critical pair of $R$ separately.
3. Construct a balanced addition tree over the resulting sums of step 2 and the
   noncritical leaves.
```

Theorem 3.3. *Let T be the addition tree over X constructed by Algorithm 2. If X is sorted, then T can be obtained in $O(n)$ time and $E(T) \leq 2(\lceil \log(n-1) \rceil + 1)E(T_{\min})$.*

Proof. Steps 2 and 3 of Algorithm 2 both take $O(n)$ time. By Lemma 3.2, step 1 also takes $O(n)$ time and thus Algorithm 2 takes $O(n)$ time. As for the error analysis, let T' be the addition tree constructed at step 3. Then $C(T) = C(T') + \Pi$. Let h be the number of levels of T'. Since T' is a balanced tree, $C(T') \leq (h-1)(\Pi + \Delta)$ and thus $C(T) \leq h(\Pi + \Delta)$. By assumption, X has at least two numbers with the opposite signs. So there are at most $n - 1$ numbers to be added pairwise at step 3. Thus $h \leq \lceil \log(n-1) \rceil + 1$. Next, by Lemma 3.1, since R is a minimum critical matching of X, we have $C(T_{\min}) \geq (\Pi + \Delta)/2$. In summary, $E(T) \leq 2(\lceil \log(n-1) \rceil + 1)E(T_{\min})$. ∎

3.2. Improved approximation for single-sign X. This section assumes that all x_i are positive; the symmetric case where all x_i are negative can be handled similarly.

Let T be an addition tree over X. Observe that $C(T) = \sum_{i=1}^{n} x_i d_i$, where d_i is the number of edges on the path from the root to the leaf x_i in T. Hence finding an optimal addition tree over X is equivalent to constructing a Huffman tree to encode n characters with frequencies $x_1, \ldots, x_n$ into binary strings [9].

FACT 3.1. *If X is unsorted (respectively, sorted), then a $T_{\min}$ over X can be constructed in $O(n \log n)$ (respectively, $O(n)$) time.*

Proof. If X is unsorted (respectively, sorted), then a Huffman tree over X can be constructed in $O(n \log n)$ [1] (respectively, $O(n)$ [9]) time. ∎

For the case where X is unsorted, many applications require a faster running time than $O(n \log n)$. Previously, the best $O(n)$-time approximation algorithm used a balanced addition tree and thus had a worst-case error at most $\lceil \log n \rceil E_n^*$. Here we provide an $O(n)$-time approximation algorithm to compute the sum over X with a worst-case error at most $\lceil \log \log n \rceil E_n^*$. More generally, given an integer parameter $t > 0$, we wish to find an addition tree T over X such that $C(T) \leq C(T_{\min}) + t \cdot |S_n|$.

ALGORITHM 3.

```
1. Let $m = \lceil n/2^t \rceil$. Partition $X$ into $m$ disjoint sets $Z_1, \ldots, Z_m$ such that each $Z_i$ has exactly $2^t$ numbers, except possibly $Z_m$, which may have less than $2^t$ numbers.
2. For each $Z_i$, let $z_i = \max\{x : x \in Z_i\}$. Let $M = \{z_i : 1 \leq i \leq m\}$.
3. For each $Z_i$, construct a balanced addition tree $T_i$ over $Z_i$.
4. Construct a Huffman tree $H$ over $M$.
5. Construct the final addition tree $T$ over $X$ from $H$ by replacing $z_i$ with $T_i$.
```

THEOREM 3.4. *Assume that $x_1, \ldots, x_n$ are all positive. For any integer $t > 0$, Algorithm 3 computes an addition tree T over X in $O(n + m \log m)$ time with $C(T) \leq C(T_{\min}) + t|S_n|$, where $m = \lceil n/2^t \rceil$. Since $|S_n| \leq C(T_{\min})$, $E(T) \leq (1+t)E(T_{\min})$.*

Proof. For an addition tree L and a node y in L, the *depth* of y in L, denoted by $d_L(y)$, is the number of edges on the path from the root of L to y. Since H is a Huffman tree over $M \subseteq X$ and every $T_{\min}$ is a Huffman tree over X, there exists some $T_{\min}$ such that for each z_j, its depth in $T_{\min}$ is at least its depth in H. Furthermore, in $T_{\min}$, the depth of each $y \in Z_i$ is at least that of z_i. Therefore,

$$\sum_{i=1}^{m} \sum_{x_j \in Z_i} x_j \cdot d_H(z_i) \leq C(T_{\min}).$$

Also note that for $x_j \in Z_i$, $d_T(x_j) - d_H(z_i) \leq \log 2^t = t$. Hence

$$\begin{aligned} C(T) &= \sum_{x_i \in X} x_i \cdot d_T(x_i) \\ &= \sum_{i=1}^{m} \sum_{x_j \in Z_i} x_j \cdot d_H(z_i) + \sum_{i=1}^{m} \sum_{x_j \in Z_i} x_j \cdot (d_T(x_j) - d_H(z_i)) \\ &\leq C(T_{\min}) + t \sum_{x_i \in X} x_i. \end{aligned}$$

In summary, $C(T) \leq C(T_{\min}) + tS_n$. Since step 4 takes $O(m \log m)$ time and the others take $O(n)$ time, the total running time of Algorithm 3 is as stated. ∎

COROLLARY 3.5. *Assume that $n \geq 4$ and all $x_1, \ldots, x_n$ are positive. Then, setting $t = \lfloor \log((\log n) - 1) \rfloor$, Algorithm 3 finds an addition tree T over X in $O(n)$ time with $E(T) \leq \lceil \log \log n \rceil E(T_{\min})$.*

Proof. The proof follows from Theorem 3.4. ∎

Acknowledgments. The authors thank Don Rose, Hai Shao, Xiaobai Sun, and Steve Tate for helpful discussions and thank the anonymous referees for their detailed comments.

REFERENCES

[1] T. H. Cormen, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[2] J. W. Demmel, *Underflow and the reliability of numerical software*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 887–919.

[3] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[4] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surveys, 23 (1990), pp. 5–48.

[5] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.

[6] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, PA, 1996.

[7] D. E. Knuth, *The Art of Computer Programming* II*: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading, MA, 1997.

[8] U. W. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review, 28 (1986), pp. 1–40.

[9] J. V. Leeuwen, *On the construction of Huffman trees*, in Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming, University of Edinburgh, Scotland, Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 382–410.

[10] A. I. Mikov, *Large-scale addition of machine real numbers: Accuracy estimates*, Theoret. Comput. Sci., 162 (1996), pp. 151–170.

[11] T. G. Robertazzi and S. C. Schwartz, *Best "ordering" for floating-point addition*, ACM Trans. Math. Software, 14 (1988), pp. 101–110.

[12] V. J. Torczon, *On the convergence of the multidirectional search algorithm*, SIAM J. Optim., 1 (1991), pp. 123–145.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1577–1595

PRECISION-SENSITIVE EUCLIDEAN SHORTEST PATH IN 3-SPACE*

JÜRGEN SELLEN†, JOONSOO CHOI‡, AND CHEE-KENG YAP§

Abstract. This paper introduces the concept of *precision-sensitive algorithms*, analogous to the well-known output-sensitive algorithms. We exploit this idea in studying the complexity of the 3-dimensional *Euclidean shortest path* problem. Specifically, we analyze an incremental approximation approach and show that this approach yields an asymptotic improvement of running time. By using an optimization technique to improve paths on fixed edge sequences, we modify this algorithm to guarantee a relative error of $O(2^{-r})$ in a time polynomial in r and $1/\delta$, where δ denotes the relative difference in path length between the shortest and the second shortest path.

Our result is the best possible in some sense: if we have a *strongly precision-sensitive* algorithm, then we can show that unambiguous SAT (USAT) is in polynomial time, which is widely conjectured to be unlikely.

Finally, we discuss the practicability of this approach. Experimental results are provided.

Key words. shortest path, exact geometric algorithms, bit complexity, precision-sensitivity

AMS subject classifications. 68Q25, 68U05

PII. S0097539798340205

1. Introduction.

1.1. Precision-sensitivity versus output-sensitivity.

The complexity of geometric algorithms generally falls under one of two distinct computational frameworks. In the *algebraic framework*, the (time) complexity of an algorithm is measured by the number of algebraic operations (such as $+, -, \times, \div, \sqrt{\cdot}$) on real-valued variables, assuming exact computations. In simple cases, the input size has one parameter n corresponding to the number of input values. In the *bit framework*, (time) complexity is measured by the number of bitwise Boolean operations, assuming input values are encoded as binary strings. The input size parameter n above is usually supplemented by an additional parameter L which is an upper bound on the bit-size of any input value; see [6].

Currently, practically every computational geometry algorithm is based on the algebraic model. For instance, we usually say that the planar convex hull problem can be solved in optimal $O(n \log n)$ time. This presumes the algebraic framework. What about the bit framework? One can easily deduce that the bit complexity is $O(n \log n \mu(L))$, where $\mu(L)$ is the bit complexity of multiplying two L-bit integers. However, it is not clear that this is optimal. Thus the possibility for faster planar convex hull algorithms seems wide open in the bit model. Of course, the situation with other problems in computational geometry is similar.

*Received by the editors June 10, 1998; accepted for publication (in revised form) March 7, 1999; published electronically March 15, 2000. An extended abstract of this paper has been published in the Proceedings of the 11th ACM Symposium on Computational Geometry, Vancouver, Canada, 1995, pp. 350–359.

http://www.siam.org/journals/sicomp/29-5/34020.html

†ESG GmbH, Einsteinstraße 174, D-81675 Munich (jsellen@esg-gmbh.de). This author was supported by a postdoctoral fellowship from the DAAD, Germany.

‡Department of Computer Science, Kookmin University, 861-1, Chongnung-Dong, Songbuk-Gu, Seoul 136-702, Korea (jschoi@cs.kookmin.ac.kr). This author was supported by NSF grant CCR-9402464.

§Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012 (yap@jinai.cs.nyu.edu). This author was supported by NSF grant CCR-9402464.

This paper is interested in bit complexity and may be seen as a follow-up to [6]. Besides its inherent interest, there are other reasons for believing that the bit model will become more important for computational geometry in the future. As the field now begins to address implementation issues in earnest, it must focus on low-level operations (what was previously dismissed as "constant time operations"). In low-level operations, it is the bit-size of numbers that is the main determinant of complexity. Second, there are reasons to think that "exact computation" (see [16]) will be an important paradigm for future implementations of geometric algorithms. (The emphasis here is on "implementations" since exact computation is already the de facto standard in theoretical algorithms.) In exact computation, complexity crucially depends on the bit-sizes of input numbers.

The main conceptual contribution in this paper is the idea of *precision-sensitive* algorithms. Today, the concept of *output-sensitive* algorithms has become an important pillar of computational geometry. But output-sensitivity is basically a concept in the algebraic framework. We suggest that precision-sensitivity is the analogous concept in the bit framework. As in output-sensitive algorithms, we may define some implicit parameter $\delta = \delta(I)$ for any input instance I. Instead of measuring the output size, δ now measures the "precision-sensitivity" of I. Intuitively, the parameter δ measures the precision or number of bits needed for output. We seek to design algorithms that can take advantage of this parameter δ. (Our idea is related to recent work in numerical analysis which quantifies the distance from an input instance to the nearest singularity.)

As an example, consider the well-studied 2-dimensional Euclidean shortest path problem. In the algebraic model, the time complexity of this problem was recently shown to be $O(n \log n)$ [8], a significant improvement upon the previous $O(n^2)$ techniques. But little is known about this problem in the bit model. Here, the question reduces to whether we can compare the sums of n square roots of integers in polynomial time. This problem may require exponential time because the difference between two such sums, as far as we know, may be as small as $2^{-2^{Cn}}$ for some $C > 0$. Blömer [3, 4] considers this problem[1] and its extensions. We may let the precision-sensitive parameter δ be the difference in path length between the sought shortest path and the next shortest path. In practical situations, the gap δ is unlikely to be exponentially small. For such inputs, it may be possible to compute the shortest path in time polynomial in n (the number of obstacle vertices), L (the bit length of input numbers) and δ, provided our algorithm is "precision-sensitive."

The introduction of precision-sensitivity paves the way for studying problems that were previously considered hopeless or "solved." Notice that the same situation arises with the introduction of output-sensitivity. To take one example, the hidden surface elimination which is trivially $\Theta(n^2)$ in the usual complexity model (ergo, "uninteresting") becomes very interesting when we consider output-sensitive algorithms. See [1, 2] for some interesting results that exploit output-sensitivity in this problem.

1.2. Precision-sensitive approach to 3ESP. This paper focuses on the 3-dimensional *Euclidean shortest path* (3ESP) problem: given a collection of polyhedral obstacles in $\mathbf{R}^3$, and source and target points $s, t \in \mathbf{R}^3$, construct an obstacle-avoiding polygonal path

$$p_{min} = (s, x_1, \ldots, x_k, t), \tag{1}$$

[1] Interestingly, Blömer and Yap noted that the equality of two sums of square roots can be decided in polynomial time.

$k \geq 0$, from s to t with minimal Euclidean length. Here, the x_i's are called *breakpoints* of the path and are required to lie on edges of the obstacles. This problem is ideal for introducing precision-sensitivity because conventional approaches are doomed to failure due to the problem's NP-hardness, a result of Canny and Reif [5]. It is also useless to introduce output-sensitivity here because the output-size is $O(n)$.

On the other hand, something interesting is going on in the bit model: the algebraic numbers that describe the lengths of the shortest paths may have exponential degrees (see section 2.2). This means that to compare the lengths of two *combinatorially distinct* shortest paths may require exponentially many bits. "Combinatorially distinct" means that the respective paths pass through different sequences of edges, and each is shortest for its edge sequence. In this paper, we use the relative difference between the length d_1 of a shortest path and the length d_2 of the combinatorially distinct next shortest path as our measure of "precision-sensitivity,"

$$\delta = \delta(I) := (d_2 - d_1)/d_1. \tag{2}$$

It should be noted that δ may be 0. One possibility for $\delta = 0$ is when the shortest path passes through a concave corner. Taking δ into account is a crucial step towards a practical 3ESP algorithm.

First we clarify some further aspects of 3ESP. The exponential behavior of 3ESP has two sources: not only is the bit complexity apparently exponential, the number of combinatorially distinct shortest paths can also be exponential. In fact, Canny and Reif's NP-hardness construction exploits the latter property of 3ESP. We can separate the combinatorial aspects from the algebraic aspects as follows. Define the *combinatorial* 3*ESP problem* which, with input as in 3ESP, asks for a *shortest edge sequence*

$$S_{min} = (e_1, \dots, e_k), \tag{3}$$

such that $x_i \in e_i$ for $i = 1, \dots, k$, where the x_i are the breakpoints of some shortest path p_{min} given by (1). Once S_{min} is obtained, there are effective numerical methods to zoom into the actual breakpoints $x_1, \dots, x_k$, as we shall see. Thus the "purely" numerical part of ESP is delegated to a subsequent phase of computation.

How hard is the combinatorial 3ESP problem? Define the implicit parameter $s(I)$ of an input I to 3ESP to be $s = s(I) = |\log(|d_1 - d_2|)|$. We say an algorithm for the combinatorial 3ESP problem is *strongly precision-sensitive* if it is polynomial-time in the parameters n, L, s. By a careful analysis of the Canny–Reif proof, we show the following theorem.

THEOREM 1. *If there exists a strongly precision-sensitive algorithm for the combinatorial* 3*ESP problem, then USAT can be solved in polynomial-time.*

Here *USAT* is the *unambiguous satisfiability problem*, commonly believed not to be in polynomial-time [11, 14]. Note that the parameter $s(I)$ is an absolute measure while our sensitivity parameter $\delta(I)$ is a relative one. But this difference is not crucial. What is more important is the fact that $s(I)$ is roughly logarithmic in $\delta(I)$. In some sense, this theorem justifies our choice of $\delta(I)$.

1.3. Towards a practical algorithm. In hopes of developing a "practical algorithm," Papadimitriou [10] introduces the *approximate* 3*ESP* problem. The input is as in 3ESP plus a new input parameter $\varepsilon > 0$. The problem is to compute an ε-*approximate shortest path*, i.e., one whose length is at most $(1 + \varepsilon)$ times the length of the shortest path. The bit-complexity of this approach is resolved in [6], yielding an algorithm with time

$$T(n, M, W) = O((n^3 M \log M + (nM)^2) \cdot \mu(W)), \tag{4}$$

where $M = O(nL/\epsilon)$, $W = O(\log(n/\epsilon) + L)$ and $\mu(W) = O(W \log W \log\log W)$ is the complexity of multiplying two W-bit numbers. Despite initial hopes, this result is still impractical, even for small examples, because the stated complexity is, roughly speaking, achieved for every input instance. Our goal is to remedy this by introducing precision-sensitivity.

Recall that Papadimitriou's approach is to subdivide each obstacle edge into *segments* in a clever way and, by treating these segments as nodes in a weighted graph, to reduce the problem to finding the shortest path in a graph.

In order to introduce precision-sensitivity, we exploit the alternative scheme introduced in [6] for subdividing edges into segments. The subdivision is parameterized by a choice of $\epsilon > 0$. Our scheme has the property that the $\epsilon/2$-subdivision is a refinement of the ϵ-subdivision and hence we can incrementally reduce the approximation error. The idea is to discard in each refinement step all segments that are provably not used by the shortest path; what remains are called *essential* segments. While it is obvious that such an implementation can drastically decrease running time in practice, we show that, depending on the parameter δ, this improvement is also asymptotical.

Assuming *nondegeneracy* (see section 2.1) of S_{min} in (3), we prove the following theorem.

THEOREM 2. *There is an incremental algorithm to compute an ε-approximate shortest path in time that is polynomial in $1/\delta$ and $1/\varepsilon$. Omitting logarithmic factors, the dependency on $1/\varepsilon$ is only linear rather than quadratic.*

In case the shortest path sequence S_{min} is unique (i.e., $\delta > 0$), we can use techniques from mathematical optimization as soon as we have reached a refinement in which only S_{min} is left. The convergence depends on the spectral bounds μ, ρ corresponding to the minimum and maximum (respectively) eigenvalue of the Hessian H of the path length function $l(\lambda_1, \dots, \lambda_k)$, where $\lambda_1, \dots, \lambda_k \in \mathbf{R}$ parameterize the points $x_1, \dots, x_k$ on S_{min}.

THEOREM 3. *The length of the shortest path can be approximated to relative error ε in time polynomial in $1/\delta, \log(1/\varepsilon), n, L$ and the spectral bounds μ, ρ.*

This theorem and the remark in Theorem 2 about a linear dependency on $1/\varepsilon$ are of practical significance.

It is important to note that the given running times in Theorems 2 and 3 are upper bounds, they are tight only for $\varepsilon \le \delta$. For $\varepsilon > \delta$, and in particular for $\delta = 0$, the running time of both algorithms can be bounded by the running time of the nonincremental approach in [6].

In section 5 we shall provide some experimental results, addressing the practicability of the incremental technique.

2. Preliminaries.

Throughout the paper, we assume that the input is given by a source point s, a target point t, and a set of pairwise disjoint polyhedral obstacles, with a total of less than n edges. For each obstacle edge e, denote its endpoints by $s(e)$, $t(e)$ and write $e = \overline{s(e)t(e)}$. Let $[e]$ denote the infinite line through e. We assume that s, t as well as endpoints of edges are specified by L-bit rational numbers. For any point $q \in \mathbf{R}^3$, $\|q\|$ denotes its Euclidean norm. The scalar product of two k-tuples x, y is denoted $\langle x, y \rangle$.

2.1. Basic properties. We assume the notation in the preceding introduction. In particular, p_{min} is a *global shortest path* from s to t in the free space FS defined by the obstacles. Here, FS is defined as the closure of the complement of the union of the obstacles.

First we fix an *edge sequence* $S = (s, e_1, \dots, e_k, t)$. The sequence S is *degenerate* if $s \in [e_1]$, $t \in [e_k]$, or $[e_j] = [e_{j+1}]$ for some $j \in \{1, \dots, k-1\}$. Note that nondegeneracy of S excludes two edges e_i and e_j from lying in a common line $[e_i] = [e_j]$ only when $|i - j| = 1$, but not if $|i - j| > 1$.

A path

$$p = (s, x_1, \dots, x_k, t)$$

is called an *S-path* if $x_i \in [e_i]$ for all i. An S-path p is *admissible* if $x_i \in e_i$ for all i.

A breakpoint x_i of p that lies on the line between its neighboring vertices, $x_i \in \overline{x_{i-1}x_{i+1}}$, is called *redundant*. Without loss of generality we may assume that p_{min} in (1) contains no redundant vertices.

We will parameterize points $x_i \in [e_i]$ by a scalar λ_i according to the equation

$$x_i = s(e_i) + \lambda_i u(e_i), \text{ with } u(e_i) = \frac{t(e_i) - s(e_i)}{\|t(e_i) - s(e_i)\|}.$$

Let $x_0 = s$ and $x_{k+1} = t$. Then the polygonal path $p = (s, x_1, \dots, x_k, t)$ over S has length

$$l_S(\lambda_1, \dots, \lambda_k) = \sum_{i=0}^{k} \|x_{i+1} - x_i\|.$$

We also write $|p|$ for $l_S(\lambda_1, \dots, \lambda_k)$. Let $p_{min}(S)$ be defined to be the path p over S that minimizes the function $l_S(\lambda_1, \dots, \lambda_k)$, *without consideration of the obstacles and without requiring admissibility.*

A necessary condition for $l_S : \mathbf{R}^k \to \mathbf{R}$ to take its global minimum at $\lambda = (\lambda_1, \dots, \lambda_k)$ is that all partial derivatives vanish at λ. This condition can be interpreted as *Snell's law*, and, as the next lemma will reveal, is also a sufficient condition to specify shortest paths.

LEMMA 1. *The function $l_S : \mathbf{R}^k \to \mathbf{R}$ is convex. If the shortest path over the lines $[e_i]$ has no redundant breakpoints, then l_S has a unique minimizer $\zeta \in \mathbf{R}^k$.*

Proof. (1) Let $l = l_S = \sum_{i=0}^{k} l_i$, where

$$l_i(\lambda_1, \dots, \lambda_k) := ||x_{i+1} - x_i||.$$

We may interpret l_i as a function in two variables λ_i and λ_{i+1} (unless $i = 0$ or k, in which case l_i depends on a single variable λ_1 or λ_k).

To show that l is convex, it suffices to show that each of the l_i is convex (the sum of convex functions is convex). The convexity of l_i is a special case of a general result in convex analysis: for any norm $||.|| : \mathbf{R}^k \to \mathbf{R}$ and any linear function $f : \mathbf{R}^m \to \mathbf{R}^k$, the function $||f|| : \mathbf{R}^m \to \mathbf{R}$ is convex (see, e.g., [12]).

(2) The convexity of l guarantees that every local minimum of l is a global minimum, say, d_S, and that the set of points $\lambda \in \mathbf{R}^k$ satisfying $l(\lambda) = d_S$ (the set of minimizers) is convex.

Assume that there are two distinct minimizers $\zeta_1, \zeta_2 \in \mathbf{R}^k$. Then every $\mu(t) = (\mu_1(t), \ldots, \mu_k(t)) := \zeta_1 + t(\zeta_2 - \zeta_1)$, $t \in [0,1]$, is a minimizer, and hence $l(\mu(t)) \equiv$ const. But

$$l(\mu(t)) = \sum_{i=0}^{k} l_i(\mu(t)),$$

where

$$l_i(\mu(t)) = \sqrt{A_i(t - B_i)^2 + C_i}$$

with $A_i \geq 0$ and $C_i \geq 0$. This can be constant only if each of the functions $l_i(\mu(t))$ is linear in t, i.e., $A_i = 0$ or $C_i = 0$ (this directly follows from $\partial^2 l(\mu(t))/\partial t^2 \equiv 0$).

Now let j be the first index for which $\mu_i(t)$ is not constant, i.e., $\mu_i(t) \equiv$ const $\forall\, i = 1, \ldots, j-1$ and $\mu_j(t) \not\equiv$ const (j may be equal to 1).

The fact that $l_j(\mu(t))$ is linear then implies that $x_{j-1} \in [e_j]$: the point $x_j(t) := s(e_j) + \mu_j(t) u(e_j)$ is moving on the line $[e_j]$ while keeping distance $l_j(\mu(t))$ to the fixed point $x_{j-1} = s(e_{j-1}) + \mu_{j-1}(t) u(e_{j-1})$.

Thus x_{j-1} and $x_j(t)$ lie on the same line $[e_j]$ for all $t \in [0,1]$. From Snell's law it follows that also $x_{j+1}(t)$ must lie on this line, showing that the vertex $x_j(t)$ is redundant. $\square$

Note that, in contrast to this proof, the known proof for the uniqueness of the shortest path [13] (see also [16, appendix]) uses geometrical arguments.

LEMMA 2. *Let $S = (s, e_1, \ldots, e_k, t)$ be nondegenerate.*

(i) *If $[e_i] \cap [e_{i+1}] = \emptyset$ $\forall i$, then the Hessian $H = H(\lambda)$ of $l_S = l_S(\lambda)$ is positive-definite.*

(ii) *If a path $p = (s, x_1, \ldots, x_k, t)$ is such that*

$$x_i \notin [e_{i-1}] \cup [e_{i+1}], \quad i = 1, \ldots, k,$$

then $H(\lambda)$ is locally positive-definite at p.

Proof. The Hessian $H = H(\lambda)$ of $l = l_S$ is a tridiagonal $k \times k$-matrix

$$H = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & \ddots & & \\ & & & & b_{k-1} \\ & & & b_{k-1} & a_k \end{pmatrix}.$$

Let H_i be the Hessian of the function $l_i = ||x_{i+1} - x_i||$, interpreted as function over λ_i and λ_{i+1} if $1 \leq i \leq k-1$, and over λ_1 (resp., λ_k) if $i = 0$ (resp., $i = k$):

$$H_0 = (a_1^-), \quad H_i = \begin{pmatrix} a_i^+ & b_i \\ b_i & a_{i+1}^- \end{pmatrix}, \quad H_k = (a_k^+),$$

with

$$a_i^+ = \frac{\partial^2 l_i}{\partial \lambda_i^2}\ , \ a_{i+1}^- = \frac{\partial^2 l_i}{\partial \lambda_{i+1}^2}\ , \ b_i = \frac{\partial^2 l_i}{\partial \lambda_i \partial \lambda_{i+1}}.$$

Then H and the H_i are related by $a_i = a_i^- + a_i^+$. Abusing the notation, we may write $H = H_0 + \cdots + H_k$.

To prove that H is positive-definite, it remains to show that the determinant of H is not zero. Write

$$H = \left(\begin{array}{c|c} a_1 & b_1 \quad\quad \\ \hline b_1 & \\ & H' \\ & \end{array}\right), \quad H^+ = \left(\begin{array}{c|c} a_1^+ & b_1 \quad\quad \\ \hline b_1 & \\ & H' \\ & \end{array}\right),$$

where H' is the matrix obtained by deleting the first row and first column of H. Then

$$\det(H) = \det(H^+) + a_1^- \det(H').$$

As all H_i are positive-semidefinite, it follows $H^+ = H_1 + \cdots + H_k$ is positive-semidefinite, and thus $\det(H^+) \geq 0$.
This implies

$$\det(H) \geq a_1^- \det(H').$$

Continuing recursively, we finally get

$$\det(H) \geq a_1^- \cdot \cdots \cdot a_k^-.$$

Abbreviating

$$v_i = \frac{x_{i+1} - x_i}{||x_{i+1} - x_i||},$$

we have

$$a_i^- = \frac{1 - \langle v_{i-1}, u(e_i)\rangle^2}{||x_i - x_{i-1}||}.$$

This is strictly positive under conditions (i) or (ii) in the statement of the lemma. Hence $\det(H) > 0$. ◻

2.2. Bit complexity. The goal of this section is to provide some background on the algebraic complexity of 3ESP.

First, we specify shortest paths over edge sequences algebraically. Let $S = (s, e_1, \ldots, e_k, t)$ be a fixed edge sequence.

For given intervals $I_i \in \{ \{0\}, \{|e_i|\}, [0, |e_i|] \}$, we define a Boolean formula (in the free variables $\lambda_1, \ldots, \lambda_k$)

$$B_S(I_1, \ldots, I_k) : \quad \bigwedge_{i=1}^{k} (\mathit{Essential}_i \ \wedge \ \mathit{Optimal}_i)$$

with the predicate

$$\mathit{Essential}_i \Leftrightarrow \left\langle \frac{x_{i+1} - x_i}{\|x_{i+1} - x_i\|}, \frac{x_i - x_{i-1}}{\|x_i - x_{i-1}\|} \right\rangle^2 \neq 1,$$

specifying that x_i is nonredundant, and the predicate

$$\mathit{Optimal}_i \Leftrightarrow \begin{cases} \mathit{Snell}(e_i), & I_i = [0, |e_i|] \\ \lambda_i = 0, & I_i = \{0\} \\ \lambda_i = |e_i|, & I_i = \{|e_i|\}, \end{cases}$$

specifying that — according to I_i — the point x_i is fixed at an endpoint of e_i or obeys Snell's law *Snell*(e_i) at some point in the relative interior of e_i (so I_i only serves as a "flag").

It is obvious that the predicates *Essential*$_i$ and *Optimal*$_i$ can be written as a Boolean combination of a constant number of polynomial inequalities of bounded degree over the variables $\lambda_{i-1}, \lambda_i, \lambda_{i+1}$ and a constant number of "additional variables." (Note that roots $\sqrt{r}$ can be eliminated by introducing a new variable a, substituting $\sqrt{r}$ by a and adding $(a^2 = r \ \wedge \ a \geq 0)$ to the formula. The variable a may be introduced by the quantifier $\exists$.)

As a corollary of Lemma 1, we get the following lemma.

LEMMA 3. *For any fixed edge sequence S and intervals $I_1, \ldots, I_k$, the formula $B_S(I_1, \ldots, I_k)$ is satisfied by at most one algebraic point $\lambda = (\lambda_1, \ldots, \lambda_k)$. If $p_{min}(S)$ is a shortest path over S with nonredundant vertices $x_i = s(e_i) + \lambda_i u(e_i)$, then there exist intervals $I_1, \ldots, I_k$ such that λ satisfies $B_S(I_1, \ldots, I_k)$.*

In particular, there exists a sequence $S = (s, e_1, \ldots, e_k, t)$ and there exist intervals $I_1, \ldots, I_k$ such that the shortest path p_{min} is parameterized by the single solution λ of $B_S(I_1, \ldots, I_k)$.

To derive some upper bounds on the bit complexity of the 3ESP problem, we shall use several results on quantifier elimination and root separation (see, e.g., [9]):

- *Quantifier elimination*: Given a Tarski formula with free variable y
$$\text{(P)} \quad \exists x \in \mathbf{R}^n : \bigvee_i \bigwedge_j p_{ij}(x, y) \Delta_{ij} 0$$
with $\leq m$ polynomials $p_{ij}(x, y)$ each of degree $\leq d$, with integer coefficients of bit-size $\leq L$, and $\Delta_{ij} \in \{>, \geq, =\}$, then there exists an equivalent predicate
$$\text{(P}'\text{)} \quad \bigvee_i \bigwedge_j h_{ij}(y) \Delta'_{ij} 0$$
with $(md)^{O(n)}$ polynomials h_{ij} of degree $(md)^{O(n)}$ and coefficients of bit-size $L(md)^{O(n)}$. ("Equivalent" means that (P) is true for a fixed $y = c$ if and only if (P$'$) is true for $y = c$.) The predicate (P$'$) can be constructed in time polynomial in L and $(md)^{O(n)}$.
- *Cauchy's bound*: Given any univariate polynomial $A(y) = \sum_{i=0}^{d} a_i y^i$ with integer coefficients a_i of bit-size $\leq L$, every root $\alpha \neq 0$ of A satisfies $|\alpha| \geq 2^{-2L}$.
- *Root separation*: If α and β are two distinct roots of $A(y)$, then $|\alpha - \beta| \geq (d2^L)^{-Cd}$ (for some $C > 0$ that does not depend on A). Isolating intervals for all roots of $A(y)$ can be computed in time polynomial in L and d.

Let us consider the formula $B_S(I_1, \ldots, I_k)$ from Lemma 3. With the above results we immediately get the following lemma.

LEMMA 4. *Every coordinate λ_i of a parameter tuple λ satisfying $B_S(I_1, \ldots, I_k)$ has a defining polynomial h_{λ_i} of degree $n^{O(n)}$ with integer coefficients of bit-size $Ln^{O(n)}$. The polynomial h_{λ_i} together with an isolating interval for λ_i can be computed in time polynomial in L and $n^{O(n)}$.*

Proof. Consider the formula
$$\text{(P1)} \quad \exists \lambda_1 \ldots \exists \lambda_{i-1} \exists \lambda_{i+1} \ldots \exists \lambda_k : \ B_S(I_1, \ldots, I_k).$$

The formula (P1) contains $O(n)$ polynomials in $O(n)$ variables of bounded degree with rational (resp., integer) coefficients of size $O(L)$. The stated quantifier elimination result provides $n^{O(n)}$ polynomials h_{ij} of degree $n^{O(n)}$, with integer coefficients of bit-size $Ln^{O(n)}$.

If there is a tuple λ satisfying (P1), then λ_i will be a root of one of the polynomials h_{ij} (recall that λ is unique).

Now consider the product

$$h := \prod_{i,j} h_{ij}.$$

Using root separation, we compute isolating intervals for each of the $n^{O(n)}$ roots of h. For each root, we can check if it satisfies (P1).

Clearly, the whole computation can be done in time polynomial in L and $n^{O(n)}$. ▯

To determine which choice of intervals $I_1, \dots, I_k$ specifies the shortest path over a given sequence S and to determine the shortest path p_{min} we need to compute and compare shortest path lengths.

Lemma 5. *Let λ and λ' be solutions of $B_S(I_1, \dots, I_k)$ and $B_{S'}(I'_1, \dots, I'_{k'})$, respectively, and let p and p' be the corresponding paths. Then $|p| = |p'|$ or*

$$|(|p| - |p'|)| \geq 2^{-Ln^{Cn}}$$

(for a global constant $C > 0$).

Proof. The difference in path length between p and p' is the unique solution y of

$$\text{(P2)} \quad \begin{aligned} &\exists \lambda_1 \dots \lambda_k \exists \lambda'_1 \dots \lambda'_{k'} : \\ &y = \sum_{i=0}^{k} ||x_{i+1} - x_i|| - \sum_{i=0}^{k'} ||x'_{i+1} - x'_i|| \\ &\wedge\ B_S(I_1, \dots, I_k)\ \wedge\ B_{S'}(I'_1, \dots, I'_{k'}) \end{aligned}$$

The formula (P2) can again be written as a Tarski formula. Analogous to Lemma 4, one can use quantifier elimination to obtain a polynomial h with $h(y) = 0$. The coefficients of h are of bit-size $Ln^{O(n)}$. The claim follows immediately from Cauchy's bound. ▯

In order to actually compute the shortest path p_{min}, we have to filter out those shortest paths, or solutions to $B_S(I_1, \dots, I_k)$, which would collide with obstacles. Having calculated the parameter λ satisfying $B_S(I_1, \dots, I_k)$, this amounts to answering the query "$\overline{x_i x_{i+1}} \in FS$?", for $i = 0, \dots, k$. But this query can be expressed as a Tarski sentence in a fixed number of variables and can be decided in time polynomial in L and $n^{O(n)}$. We finally obtain the following theorem.

Theorem 4. *It is possible to compute algebraic representations of all combinatorially distinct shortest paths in time polynomial in L and $n^{O(n)}$.*

In this theorem, we may assume that each shortest path is represented by a sequence $(S, I_1, \dots, I_k, \lambda_1, \dots, \lambda_k)$, where $S = (s, e_1, \dots, e_k, t)$ is an edge sequence, the I_j's are interval flags for S, and the λ_j satisfy the formula $B_S(I_1, \dots, I_k)$. Furthermore, each λ_j is represented by one of its isolated interval representations.

3. Combinatorial 3ESP is as hard as USAT. Recall that the exponential complexity of the 3ESP problem has a combinatorial and an algebraic source. We give evidence that 3ESP remains intractable even after eliminating the algebraic source of complexity.

We briefly review the Canny–Reif construction (see [5, section 2.5]): Given a 3SAT formula f in conjunctive form with m clauses and n variables $b_1, \dots, b_n$, it is possible to construct an environment $E(f)$ such that the following holds for a fixed "reference length" $l = 2^{3n}$ and $\Delta = 2^{-nm-3n-4}$: To each instantiation of $(b_1, \dots, b_n)$ there corresponds an edge sequence $S = S(b_1, \dots, b_n)$ such that the shortest path p over S lies in free space and satisfies

$$|p| \in \begin{cases} [l, l+\Delta] & \text{if } f(b_1, \dots, b_n) = 1, \\ [l+2\Delta, \infty) & \text{if } f(b_1, \dots, b_n) = 0. \end{cases}$$

The number of edges of $E(f)$ as well as the maximal bit-size of coordinates is polynomial in n and m. Deciding the satisfiability of f is reduced to deciding if the shortest path in $E(f)$ has length $\le l + \Delta$.

A careful analysis shows the following property of $E(f)$: if the formula f is uniquely satisfiable, i.e., by exactly one instantiation of $(b_1, \dots, b_n)$, then the shortest path in $E(f)$ is unique and the gap in length between this path and any path that passes over a different edge sequence is single-exponential (i.e., $> c^{-nm}$ for some $c > 1$).

The argument is as follows: The basic construction elements in [5] are parallel, 2-dimensional plates with (for ease of description) 1-dimensional slots. The construction is based on a scene with 2^n shortest paths, with length $l' \le l + \Delta$. In the final step, obstacles are introduced which stretch all paths that correspond to nonsatisfying instances by at least Δ. It remains to verify that there are no further locally shortest paths that use other edge sequences and have length close to l'. The use of parallel plates ensures that these paths would have additional legs between slots. The spacing between plates and between the breakpoints of the shortest paths in the slots gives a lower bound on the additional length and is again roughly Δ. Finally, the gap Δ is single-exponential.

Now assume that we have a strongly precision-sensitive algorithm as defined in section 1.2. Consider the satisfiability problem restricted to 3SAT formulas that are satisfiable by at most one variable instance, known as the *unambiguous satisfiability problem USAT*. Assume we are given such a formula f. By constructing $E(f)$ and running our algorithm, we would be able to decide the satisfiability of f in polynomial time. This proves Theorem 1.

4. Approximation. For simplicity, we shall describe algorithms in this section in the algebraic framework. It is important to note that the hardness result of section 3 is not valid in this model. However, as in [6], the technique extends to the bit framework. In particular, it suffices to compute intermediate numbers to precision $W = O(\log(n/\varepsilon) + L)$.

We review the approximation scheme for 3ESP in [6]: the algorithm mainly consists of three steps. In the first step, the edges are subdivided into segments using a method that depends on some given parameter $\varepsilon' > 0$. This ε'-*subdivision* (as it is called) satisfies the properties given in the following lemma.

LEMMA 6 (see [6]).

(1) *Each edge is divided into $O(L/\varepsilon')$ segments.*

(2) *Each segment σ of the subdivision satisfies $|\sigma| \le \varepsilon' \mathit{dist}(s, \sigma)$.*

(3) *The $\varepsilon'/2$-subdivision is a refinement of the ε'-subdivision.*

In the second step of the algorithm, the *visibility graph* $G_0 = (V_0, E_0)$ of the segments is constructed. The nodes of the graph comprise the subdivision segments including s and t. The edges comprise pairs (σ, σ') of segments that can "see each other," meaning that there exists $x \in \sigma$ and $x' \in \sigma'$ such that $\overline{xx'} \in FS$. In the third step, the visibility graph G_0 is weighted by assigning to each edge (σ, σ') the Euclidean distance between the midpoints of σ and σ'. Finally, the shortest path Σ in G_0 is computed by running Dijkstra's shortest path algorithm. This path is a *segment sequence* $\Sigma = (s, \sigma_1, \dots, \sigma_k, t)$. Its "weight" according to the midpoint distances is further denoted as $|\Sigma|$.

The following lemma relates $|\Sigma|$ to $|p_{min}|$ (and shows the correctness of the approximation scheme).

LEMMA 7. *For $\varepsilon' = \varepsilon/Cn$, C a given constant, Σ satisfies $|\Sigma| \le (1+2\varepsilon/C)|p_{min}|$ and $|p_{min}| \le (1+2\varepsilon/C)|\Sigma|$.*

Proof. Consider the path Σ_{min} in G_0 which corresponds to p_{min} (Σ_{min} is equivalent to a path that connects the midpoints of the segments used by p_{min}). By the triangle inequality the weight of each leg (σ_i, σ_{i+1}) of Σ_{min} can be bounded by the length of the corresponding leg of p_{min}, plus the length of the segments σ_i and σ_{i+1}. Hence, we obtain

$$|\Sigma| \le |\Sigma_{min}| \le |p_{min}| + 2\sum_{j=1}^{k} |\sigma_j|.$$

With $k \le n$, $|\sigma_j| \le \varepsilon' \mathrm{dist}(s, \sigma_j)$, and $\mathrm{dist}(s, \sigma_j) \le |p_{min}|$, we get

$$|\Sigma| \le (1 + 2\varepsilon/C)|p_{min}|.$$

To prove the second inequality, we consider the path p over Σ which connects pairwise visible points $x_i^1 \in \sigma_i$, $x_{i+1}^2 \in \sigma_{i+1}$, and which connects the points x_i^1, x_i^2 on each σ_i by additional legs. By the triangle inequality, we obtain

$$|p_{min}| \le |p| \le |\Sigma| + 2\sum_{j=1}^{k} |\sigma_j|.$$

With $k \le n$, $|\sigma_j| \le \varepsilon' \mathrm{dist}(s, \sigma_j)$, and $\mathrm{dist}(s, \sigma_j) \le |\Sigma|$, we finally get

$$|p_{min}| \le (1 + 2\varepsilon/C)|\Sigma|. \qquad \square$$

4.1. An incremental algorithm. The above algorithm uses a fixed subdivision. In the following, we shall exploit property (3) in Lemma 6 by successively halving the error bound ε and by refining only those segments which the global shortest path could potentially use.

Let $\varepsilon_i = 2^{-i}$ and $\varepsilon_i' = \varepsilon_i/Cn$, for the fixed constant $C = 32$. (This is a significant improvement to the conference version of this paper, where we divide by Cn^2 instead of Cn.) Let $G_i = (V_i, E_i)$ be the weighted visibility graph for any set of segments V_i fulfilling the basic inequality (2) of Lemma 6, and let l_i denote the length of the shortest path from s to t in G_i. By Lemma 7, we get $l_i \le (1 + \varepsilon_i/16)|p_{min}|$ and $|p_{min}| \le (1 + \varepsilon_i/16)l_i$.

LEMMA 8. *If $\Sigma = (s, \sigma_1, \dots, \sigma_k, t)$, $k \le n$, is a path in G_i with $|\Sigma| > (1+\varepsilon_i/4)l_i$, then any path p over Σ satisfies $|p| > |p_{min}|$.*

Proof. Assume $|p| \leq |p_{min}|$. By the triangle inequality, we get

$$|\Sigma| \leq |p| + 2\sum_{j=1}^{k} |\sigma_j|.$$

With $k \leq n$, $|\sigma_j| \leq \varepsilon_i \mathrm{dist}(s, \sigma_j)/32n$, and $\mathrm{dist}(s, \sigma_j) \leq |p| \leq |p_{min}|$, we get

$$|\Sigma| \leq (1 + \varepsilon_i/16)|p_{min}|.$$

With $|p_{min}| \leq (1 + \varepsilon_i/16)l_i$, we finally get

$$|\Sigma| \leq (1 + \varepsilon_i/4)l_i,$$

a contradiction. □

We define the *essential subgraph* $G_i^{ess} = (V_i^{ess}, E_i^{ess})$ of G_i to be the subgraph which is spanned by the union of all (s,t)-paths Σ in G_i with $|\Sigma| \leq (1 + \varepsilon_i/4)l_i$.

LEMMA 9. *If p_{min} leads over a segment sequence $\Sigma = (s, \sigma_1, \ldots, \sigma_k, t)$ in G_i, then Σ is in G_i^{ess}.*

To approximate a shortest path p_{min} by successive refinement, we need thus only to consider the segments in V_i^{ess} in the next step.

We can compute G_i^{ess} as follows: run Dijkstra's single source shortest path algorithm on G_i twice, starting at s and starting at t, and assign to each $\sigma \in V_i$ the distances $d_s(\sigma)$ (resp., $d_t(\sigma)$) to s (resp., t) in G_i. This implies $l_i = d_s(t) = d_t(s)$. Let the weight of edge (σ, σ') in G_i be denoted by $\omega(\sigma, \sigma')$. Then we can choose E_i^{ess} to be the set of all $(\sigma, \sigma') \in E_i$ that satisfy

$$d_s(\sigma) + d_t(\sigma') + \omega(\sigma, \sigma') \leq (1 + \varepsilon_i/4)l_i.$$

In practice, G_i^{ess} should be significantly smaller than G_i. In fact, it approaches the 1-dimensional skeleton formed by all global shortest paths as $\varepsilon_i \to 0$. In the next lemma we show that, depending on the precision-sensitivity parameter δ, the incremental construction of G_i^{ess} will eventually resolve the edge sequence S_{min} of the global shortest path p_{min}.

LEMMA 10. *Let $\varepsilon_i < \delta$ and let (σ, σ') be an arbitrary edge of G_i^{ess} with $\sigma \in e, \sigma' \in e'$ (e and e' are obstacle edges). Then either $e = e'$ or (e, e') is an edge of S_{min}.*

Proof. The graph G_i^{ess} contains a path $\Sigma = \Sigma_1 \cdot (\sigma, \sigma') \cdot \Sigma_2$, where Σ_1 is a shortest path from s to σ and Σ_2 is a shortest path from σ' to t. By construction of G_i^{ess}, Σ satisfies $|\Sigma| \leq (1 + \varepsilon_i/4)l_i$. Let $p = p_1 \cdot (x, x') \cdot p_2$ be an admissible path over Σ, i.e., a path which realizes the visibility relation (p is a zig-zag path which uses additional legs on segments). As Σ_m, $m = 1, 2$, is a shortest path in G_i^{ess}, Σ_m enters and leaves an obstacle edge e at most once (otherwise, there would be a cycle and a shortcut on e). Hence, each Σ_m leads over $k \leq n$ segments σ_j. By the triangle inequality, we get

$$|p| \leq |\Sigma| + 4\sum_{j=1}^{k} |\sigma_j|.$$

With $|\sigma_j| \leq \varepsilon_i \mathrm{dist}(s, \sigma_j)/32n$ and $\mathrm{dist}(s, \sigma_j) \leq |\Sigma|$, we get

$$|p| \leq (1 + \varepsilon_i/4)|\Sigma|.$$

With $|\Sigma| \le (1+\varepsilon_i/4)l_i$ and $l_i \le (1+\varepsilon_i/4)|p_{min}|$, we obtain $|p| < (1+\varepsilon_i)|p_{min}|$. By definition of δ, p must lie on S_{min}. Finally, the edge (σ, σ') used by p must lie on the same obstacle edge e of S_{min} or must correspond to an edge (e, e') of S_{min}. ∎

We are now ready to formulate the *incremental algorithm* to get a relative error of $\varepsilon = 2^{-r}$:

```
(1)  $i := 0;\ \varepsilon'_0 := 1/Cn;$
(2)  Compute the initial $\varepsilon'_0$-subdivision $V_0$;
(3)  Repeat
(4)      Construct the visibility graph $G_i = (V_i, E_i)$;
(5)      Compute $G_i^{ess} = (V_i^{ess}, E_i^{ess})$;
(6)      Compute $V_{i+1}$ by refining $V_i^{ess}$;
(7)      $i := i+1;\ \varepsilon'_i := \varepsilon'_{i-1}/2;$
(8)  Until $i = r+1$.
```

4.2. Spectral analysis. Our first goal in this section is to characterize the behavior of the incremental algorithm for a fixed edge sequence S.

Let $l = l_S$, and let again H be the Hessian of l with spectral bounds μ and ρ. Let $\zeta = (\zeta_1, \ldots, \zeta_k)$ be the parameter tuple specifying $p_{min}(S)$, and z_j the breakpoint of $p_{min}(S)$ on e_j, specified by ζ_j. Our goal is to show that a path p over S whose length differs only slightly from $|p_{min}(S)|$ must also have a parameter λ which is close to ζ. Taylor's theorem shows that for any $\lambda \in \mathbf{R}^k$, there exists $\tau \in \mathbf{R}^k$ such that

$$l(\lambda) - l(\zeta) = \langle \nabla l(\zeta), \lambda - \zeta \rangle + \frac{1}{2}\langle \lambda - \zeta, H(\tau)(\lambda - \zeta)\rangle. \tag{5}$$

The first term is equal to zero as ζ minimizes the function l_S (see section 2.1). The second term can be bounded by the "spectrum" of H:

$$\mu\|\lambda - \zeta\|^2 \ \le\ \langle \lambda - \zeta, H(\tau)(\lambda - \zeta)\rangle \ \le\ \rho\|\lambda - \zeta\|^2. \tag{6}$$

This implies

$$l(\lambda) - l(\zeta) \ge \frac{\mu}{2}\|\lambda - \zeta\|^2.$$

Thus, the parameter λ of any path p over S with $|p| - |p_{min}(S)| \le \tilde{\varepsilon}$ satisfies $\|\lambda - \zeta\|^2 \le 2\tilde{\varepsilon}/\mu$ (for any $\tilde{\varepsilon} > 0$).

In the next lemma, we consider G_i^{ess} after the edge sequence S_{min} of the unique shortest path has been resolved.

LEMMA 11. *Let $\varepsilon_i < \delta$, let $\sigma \in G_i^{ess}$ be a segment with $\sigma \subseteq e$, let $x \in \sigma$ be an arbitrary point, and let $z \in e$ be the breakpoint of the shortest path p_{min} on e. Then*

$$||x - z|| \le \left(2\varepsilon_i \frac{|p_{min}|}{\mu}\right)^{\frac{1}{2}}.$$

Proof. By Lemma 10, we can assume that $x = x_j$ is the jth vertex of a path p over S_{min} with $|p| \le (1+\varepsilon)|p_{min}|$. Accordingly, let $z = z_j$ be the jth vertex of p_{min}. Now, let λ be the parameter of p, and let ζ be the parameter of p_{min}. Setting $\tilde{\varepsilon} = \varepsilon_i|p_{min}|$, we get

$$\begin{aligned} ||x_j - z_j|| &= |\lambda_j - \zeta_j| \le ||\lambda - \zeta|| \\ &\le \left(2\varepsilon_i \frac{|p_{min}|}{\mu}\right)^{\frac{1}{2}}. \end{aligned}$$ ∎

Now assume we run our incremental algorithm for the fixed edge sequence S_{min} and are in step i (i.e., the ith iteration step). Then on each edge e_j those segments whose distance from z_j is more than const $\cdot \sqrt{\varepsilon_i}$ will automatically not be considered. Here, const $= \sqrt{2|p_{min}|/\mu}$ depends on S but not on i.

By construction, each segment σ on e_j has length $|\sigma| \geq a_j\varepsilon_i/32n$ (with a_j the distance from e_j to the source s). Thus we refine at most

$$C(S) \cdot n\varepsilon_i^{-\frac{1}{2}}$$

segments on each e_j in the ith step, with

$$C(S) = \frac{32}{a}\sqrt{2\frac{|p_{min}|}{\mu}}$$

and $a = \min_j\{a_j\}$.

Let $\varepsilon = 2^{-r}$ be the desired relative error. Summing over $i = 1, \ldots, r$, we produce a total of $O(r\sqrt{1/\varepsilon_r})$ segments. This is a significant improvement to the original (not precision-sensitive) scheme, which would produce $O(1/\varepsilon_r)$ segments.

The described effect occurs in the overall algorithm as soon as $\varepsilon_i < \delta$. Lemmas 10 and 11 then imply that the essential subgraph G_i^{ess} contains less than $O(nC(S_{min})2^{i/2})$ segments per edge. On the other hand, if $\varepsilon_i > \delta$, then G_i^{ess} contains $O(nL/\varepsilon_i) = O(nL/\delta)$ segments per edge. (This is the number of segments produced by the non-incremental scheme in [6].) This yields the following lemma.

LEMMA 12. *The essential subgraph G_i^{ess} contains less than*

$$M_i = O\left(n\left(\frac{L}{\delta} + C(S_{min}) \cdot 2^{\frac{i}{2}}\right)\right)$$

segments per edge.

We note that the number of segments per edge which are produced by the algorithm in [6] is also an upper bound for M_i.

The visibility relation between segments can be computed separately for each of the $O(r)$ refinement steps by a sweep algorithm, as described in [6]. The cost of this algorithm dominates the computation of G_i^{ess}. Thus, the running time of the ith step is $T(n, M_i, W)$, with T as in (4). The running time of the total algorithm can be bounded by

$$O(r \cdot T(n, M_r, W)).$$

Thus, we have proven Theorem 2.

4.3. Path optimization. With the incremental approach above, we have a tool to determine S_{min} in (3) in time polynomial in $1/\delta$. As soon as there is only one possible edge sequence (or only a few combinatorially distinct sequences) left, it is however more efficient to use an optimization technique to approximate the actual shortest path. We propose to use a steepest descend method (see, e.g., [7, section C-5]):

Let the spectrum of H be bounded below by $\mu > 0$ and above by ρ (choose μ as the smallest, and ρ as the biggest eigenvalue of H). We can derive explicit values for these bounds (especially for ρ) as described in section 4.4.

Let $I = [\frac{1}{2\rho}, \frac{3}{2\rho}]$. Define the sequence

$$\lambda^{i+1} = \lambda^i - \kappa \nabla l(\lambda^i),$$

with $\kappa \in I$ and λ^0 the known approximation. Then this sequence converges to the unique minimizer ζ of l at the rate of a geometric progression with ratio $q = 1 - \mu/\rho$:

$$||\lambda^{i+1} - \zeta|| \leq q||\lambda^i - \zeta|| \leq q^{i+1}||\lambda^0 - \zeta||.$$

With

$$||l(\lambda^i) - l(\zeta)|| \leq \frac{\rho}{2}||\lambda^i - \zeta||^2$$

and

$$||\lambda^0 - \zeta||^2 \leq \frac{2}{\mu}(l(\lambda^0) - l(\zeta)) \leq \frac{2\delta}{\mu} l(\zeta),$$

we get

$$||l(\lambda^i) - l(\zeta)|| \leq \frac{\rho\delta}{\mu} q^{2i} |p_{min}|.$$

To achieve $||l(\lambda^i) - l(\zeta)|| < 2^{-r}|p_{min}|$, it is sufficient to choose $i > N$ with

$$N = \Theta\left(\frac{\rho}{\mu}\left(r + |\log\delta| + \left|\log\frac{\rho}{\mu}\right|\right)\right).$$

Again, it is important to note that this method — e.g., because of the freedom of choice for κ — easily extends to the bit framework. The running time of the whole algorithm can be resolved as

$$O(\log(1/\delta) \cdot T(n, M_\delta, W) + Nn\mu(W)),$$

where $M_\delta = O(nL/\delta)$. This finally proves Theorem 3.

4.4. Spectral bounds. In this section, we discuss two different methods to get bounds on the spectrum of the Hessian H of the path length function $l = l_{S_{min}}$. We shall use the notations of section 2.

By the theory of Gerschgorin circles, the eigenvalues of H are bounded above by $\rho = \max_i\{\ a_i + |b_i| + |b_{i-1}|\ \}$, with a_i and b_i as in the proof of Lemma 2.

With the help of ρ, we can directly give a bound on μ. As the determinant of H is equal to the product of all k eigenvalues of H, we get

$$\mu \geq \frac{\det(H)}{\rho^{k-1}},$$

where the determinant of H satisfies the inequality

$$\det(H) \geq \prod_{i=1}^{k} a_i^-.$$

A crucial deficiency of the above bound is that it is exponential in k, the number of intermediate vertices of p_{min}.

A bound on μ not depending on k can be obtained by a method based on the theorem of Courant–Fischer (see [15, pp. 101–102]):

Let A, B and C be positive-semidefinite symmetric matrices with $C = A + B$, and α (resp., β) the smallest eigenvalue of A (resp., B). Then each eigenvalue γ of C satisfies $\gamma \geq \alpha + \beta$. Assuming k to be even (the case of odd k can be similarly treated), we split $H = A + B$ according to

$$A = \sum_{i=0}^{k/2} H_{2i} \ , \quad B = \sum_{i=1}^{k/2} H_{2i-1}.$$

The matrices A and B are block matrices, and the eigenvalues are the eigenvalues of the H_i. It follows that

$$\mu \geq \min\{ \ \mu_i \ ; \ i = 0, \ldots, k \ \},$$

where μ_i is the smallest eigenvalue of H_i. This bound has the nice property that it depends only on pairs of edges of S_{min}.

5. Experimental results. The preceding algorithms for the approximate 3ESP problem have a high polynomial dependency on the number of edges n or the desired error bound ε. But these theoretical bounds need not reflect the "average behavior" or practical situations. This suggests some empirical studies.

To verify the practicability of our incremental approach, we implemented a simplified version of the proposed algorithm. The simplification is based on the observation that for certain special cases, the visibility relation between segments can be replaced by the visibility of segment midpoints:

Let the obstacles be 2-dimensional facets arranged in h parallel planes separating the start and target points. Let $\varepsilon' = \varepsilon/h$, and consider the subdivision defined in section 4. Then the following lemma holds.

LEMMA 13. *There exists a free polygonal path p from s to t, which connects segment midpoints and satisfies $|p| \leq (1+\varepsilon)|p_{min}|$.*

Proof. The shortest path p_{min} from s to t is strictly monotone in the direction of the normal vector of the planes containing the obstacle facets. Now pick an arbitrary vertex v of p_{min} and move this vertex on the incident edge in either direction while keeping the other path vertices fixed. We continue this deformation until the (deformed) path hits another obstacle facet or until v hits a segment midpoint. In the first case, we consider the intersection point as a further path vertex, and in the latter case we continue the process by picking another path vertex until all vertices coincide with segment midpoints. It is easy to see that this process terminates after at most h deformation steps, introducing an absolute error of at most $h\varepsilon'|p_{min}|$. $\square$

Further, we used a uniform subdivision for each edge by starting with the edges as segments and successively halving the segments.

Tables 1 and 2 show the result of the incremental algorithm for the situation in Figures 1 and 2. The figures visualize the situation after 10 iteration steps: the essential segments are the solid black parts of the edges, and the shortest paths from start to goal determined so far are drawn dashed. In the first example there are two shortest paths, which are resolved after the 8th iteration step. In the second example,

TABLE 1
Performance statistics for the example "Horizontal obstacles 1."

Steps	0	2	4	6	8	10
Error	1.6771	0.4193	0.1048	0.0262	0.0066	0.0016
Length	2.4852	2.3746	2.3290	2.3252	2.3250	2.3250
Segments	19	76	224	186	366	728

TABLE 2
Performance statistics for the example "Horizontal obstacles 2."

Steps	0	2	4	6	8	10
Error	2.2500	0.5625	0.1406	0.0352	0.0088	0.0022
Length	2.4116	2.3263	2.3085	2.3070	2.3069	2.3069
Segments	28	112	290	334	368	430

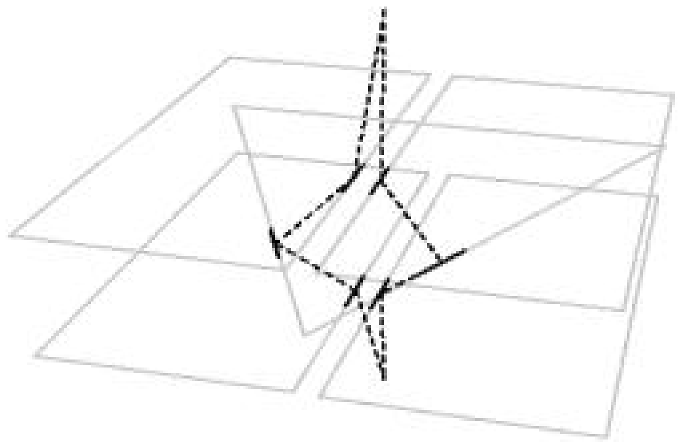


FIG. 1. *Horizontal obstacles 1.*

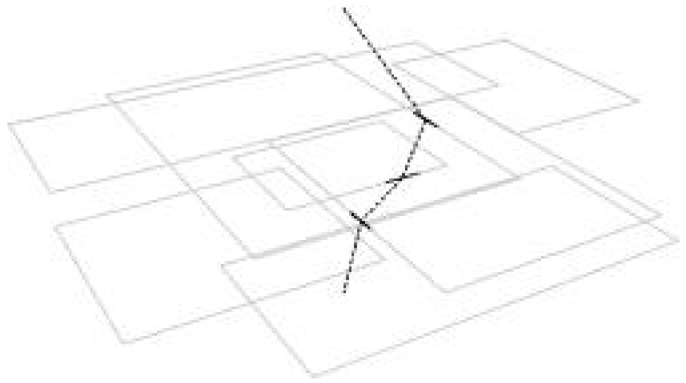


FIG. 2. *Horizontal obstacles 2.*

the unique shortest path is resolved after the 10th step.

The tables show the guaranteed relative error in path length, the length of the shortest path in the current visibility graph, and the number of the essential segments. The running time of the algorithm is mostly quadratic in this number, and was, for these examples, in the range of seconds on a state-of-the-art workstation.

The following behavior has been typical for the examples we tried: until the error bound ε_i is small enough to discard, the number of essential segments is doubled per step. Then comes a phase where the segment number does not change significantly. Once the shortest paths are resolved, the number of essential segments is doubled every two iteration steps, as predicted by the theoretical results.

6. Final remarks. We have developed the first precision-sensitive algorithms for 3ESP. Beyond its intrinsic interest, it demonstrates a critical exploitation of precision-sensitivity. We conjecture that other previously intractable problems may likewise yield to this approach.

If the sensitivity parameter δ is zero, we can modify our approach to take advantage of the "next sensitivity" parameter, namely, the gap between the second and the third shortest path, etc. A general treatment of this may be interesting.

We note that attention has to be paid to degenerate situations in this problem. But it seems unavoidable to take this into account because degeneracy seems to be one cause of intrinsic complexity in 3ESP. Note that there has been some recent literature on degeneracy in geometric problems.

The merits of the incremental 3ESP seem evident: in our examples, there would have been no chance to detect the shortest path by the exhaustive approach [6]. Our algorithm is a useful tool when a researcher needs to determine the real shortest path in a particular small environment.

REFERENCES

[1] M. de Berg, *Efficient algorithms for ray shooting and hidden surface removal*, Ph.D. thesis, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, 1992.

[2] M. Bern, *Hidden surface removal for rectangles*, J. Comput. System Sci., 40 (1990), pp. 49–59.

[3] J. Blömer, *Computing sums of radicals in polynomial time*, in Proceedings of the IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1991, pp. 670–677.

[4] J. Blömer, *Simplifying Expressions Involving Radicals*, Ph.D. thesis, Department of Mathematics, Free University, Berlin, Germany, 1992.

[5] J.F. Canny and J. Reif, *New lower bound techniques for robot motion planning problems*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1987, pp. 49–60.

[6] J. Choi, J. Sellen, and C.-K. Yap, *Approximate Euclidean shortest path in 3-Space*, in Proceedings of the 10th ACM Symposium on Computational Geometry, ACM, New York, 1994, pp. 41–48.

[7] A.A. Goldstein, *Constructive Real Analysis*, Harper and Row, New York, 1967.

[8] J. Hershberger, S. Suri, *Efficient computation of Euclidean shortest paths in the plane*, in Proceedings of the 34th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1993, pp. 508–517.

[9] J. Renegar, *On the Computational Complexity and Geometry of the First-Order Theory of the Reals*, Technical report, School of Operations Research Report, Cornell University, Ithaca, NY, 1989.

[10] C.H. Papadimitriou, *An algorithm for shortest-path motion in three dimensions*, Inform. Process. Lett., 20 (1985), pp. 259–263.

[11] C.H. Papadimitriou, *Computational Complexity*, Addison–Wesley, Reading, MA, 1994.

[12] R.T. Rockafellar, *Convex Analysis*, Princeton University Press, Princeton, NJ, 1970.

[13] M. Sharir and A. Schorr, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193–215.
[14] L.G. Valiant and V.V. Vazirani, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.
[15] J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, New York, 1965.
[16] C.-K. Yap, *Towards exact geometric computation*, Comput. Geom., 7 (1997), pp. 3–23.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1596–1620

OPTIMAL INFORMATION GATHERING ON THE INTERNET WITH TIME AND COST CONSTRAINTS*

OREN ETZIONI†, STEVE HANKS†, TAO JIANG‡, AND OMID MADANI†

Abstract. The World Wide Web provides access to vast amounts of information, but content providers are considering charging for the information and services they supply. Thus the consumer may face the problem of balancing the benefit of asking for information against the cost (in terms of both money and time) of acquiring it. We study information-gathering strategies that maximize the expected value to the consumer. In our model there is a single information request, which has a known *benefit* to the consumer. To satisfy the request, *queries* can be sent simultaneously or in sequence to any of a finite set of independent *information sources.* For each source we know the monetary cost of making the query, the amount of time it will take, and the probability that the source will be able to provide the requested information. A *policy* specifies which sources to contact at which times, and the *expected value* of the policy can be defined as some function of the likelihood that the policy will yield an answer, the expected benefit, and the monetary cost and time delay associated with executing the policy. The problem is to find an expected-value-maximizing policy.

We explore four variants of the objective function V: (i) V consists only of the benefit term subject to threshold constraints on both total cost and total elapsed time, (ii) V is linear in the expected total cost of the policy subject to the constraint that the total elapsed time never exceeds some deadline, (iii) V is linear in the expected total elapsed time subject to the constraint that the total cost never exceeds some threshold, and (iv) V is linear in the expected total monetary cost and the expected time delay of the policy. The problems of devising an optimal querying policy for all four variants and approximating an optimal querying policy for variants (iii) and (iv) are shown to be NP-hard. For (i), and with a mild simplifying assumption for (iii), we give a fully polynomial time approximation scheme. For (ii), we consider *batched* querying policies, and design an $O(n^2)$ time approximation algorithm with ratio $\frac{1}{2}$ and a polynomial time approximation scheme for optimal single-batch policies, and an $O(kn^2)$ time approximation algorithm with ratio $\frac{1}{5}$ for optimal k-batch policies.

Key words. approximation algorithm, batched policy, computational complexity, information gathering, information retrieval, Internet, scheduling, time and cost trade off, World Wide Web

AMS subject classifications. 68Q20, 68Q25, 90B35

PII. S0097539797314465

1. Introduction.

The Internet is rapidly becoming the foundation of an information economy. Valuable information sources include online travel agents, nationwide yellow pages, job listing services, online malls, and many more. Currently, most of this information is available free of charge, and as a result parallel search tools such as MetaCrawler [16] and BargainFinder [9] respond to requests by querying numerous information sources simultaneously to maximize the information provided and minimize delay. However, information providers may start charging for their services [7, 11]. Billing protocols to support an "information marketplace" have been announced by large players such as Visa and Microsoft [15] and by researchers [18].

* Received by the editors October 24, 1996; accepted for publication (in revised form) May 12, 1999; published electronically March 15, 2000. This research was supported in part by Office of Naval Research grant 92-J-1946, ARPA / Rome Labs grant F30602-95-1-0024, a gift from Rockwell International Palo Alto Research, National Science Foundation grant IRI-9357772, Natural Science and Engineering Research Council of Canada research grant OGP0046613, and Canadian Genome Analysis and Technology grant GO-12278.

http://www.siam.org/journals/sicomp/29-5/31446.html

† Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350 (etzioni@cs.washington.edu, hanks@cs.washington.edu, madani@cs.washington.edu).

‡ Department of Computer Science, University of California, Riverside, CA 92521 (jiang@cs.ucr.edu).

Once billing mechanisms are in place, consumers of information may face the problem of balancing the benefit of obtaining information against the cost (both monetary and temporal) of obtaining it. Information providers will differ in the quality of the information they provide as well as the amount they charge and the speed at which they deliver information. The consumer thus faces the problem of developing a schedule of queries to the providers that maximizes expected value, which can be expressed in terms of (1) the benefit associated with a successful query, (2) the likelihood that a particular query will yield successful results, (3) the cost of making a query, and (4) the amount of time it takes.

This paper analyzes the "query scheduling" problem for a number of variants of the objective function. We begin by stating the problem precisely and then summarize our main results.

1.1. The model. The basic problem is to find a *policy* for obtaining the answer to a (single) query. The policy will dictate which information source will be queried and when. To define a policy we begin with a (finite) set of information sources, $s_1, \ldots, s_n$. For each source s_i we introduce a cost parameter c_i and a duration parameter d_i. The former is the monetary cost assessed when the source is activated and the latter is the amount of time it takes the source to process the query. The cost and duration are known with certainty and are charged whether or not the source returns an answer to the query.[1] Finally we have p_i, the probability that s_i will return an answer to the query. Success probabilities are independent for distinct sources, and whether or not s_i will answer a query is uncertain but consistent: if s_i successfully answers a query it will always do so subsequently, and if it fails to answer a query it will always fail to do so subsequently.[2] We assume that accurate estimates of these parameters are obtainable from the history of the interactions with the information sources.

A policy can be represented as a sequence of pairs $\mathcal{P} = (s_{i_1}, t_1), \ldots, (s_{i_m}, t_m)$, where $t_1 \leq t_2 \leq \cdots \leq t_m$. This specifies that source s_{i_1} will be initiated at t_1, s_{i_2} will be initiated at t_2, and so on. An execution of the policy is terminated either when some source returns a correct answer or when the policy has been exhausted. Since each source in a policy succeeds probabilistically, a policy generates a probability distribution over *outcomes*, where each outcome is one possible way that the policy might be played out. We use $S(\mathcal{O})$, $C(\mathcal{O})$, and $T(\mathcal{O})$ to denote the outcome's success (1 or 0), total cost, and duration, respectively. The *value* of an outcome $\mathcal{O}$ is a function of $S(\mathcal{O})$, $C(\mathcal{O})$, and $T(\mathcal{O})$. The first component of the value function of an outcome $\mathcal{O}$ is always a constant reward R if the query was answered and 0 otherwise. The function additionally contains two additive components, one a function of $C(\mathcal{O})$ and one a function of $T(\mathcal{O})$. The *expected value* of a policy $\mathcal{P}$, denoted $V(\mathcal{P})$, is simply the expectation of the values of all its outcomes.

Our objective is to find a policy to maximize the expected value. We will consider four versions of the objective function: linear and threshold versions of the cost and time components. With suitable scaling of the monetary and time costs, the four objective functions assume the forms given in Table 1.1. We will hereafter refer to the four problems by their acronyms: TT for threshold in cost and time, LT for linear in cost and threshold in time, TL for threshold in cost and linear in time, and LL for linear in cost and time. Note that in the threshold cases we try to find a policy with

[1] All our results can be extended to the case when the cost is charged only if the query is successful. Cost *expectations* in place of costs should be used in a few of the models.

[2] As a result it is never profitable to query a source more than once.

TABLE 1.1
The four objective functions. Here, $\mathcal{O}$ denotes a possible outcome of the policy $\mathcal{P}$ to be found.

Objective	Time threshold	Linear in time
Cost threshold	TT: max $\mathbf{E}[R \cdot S(\mathcal{O})]$ s.t. $\forall \mathcal{O}\ \ C(\mathcal{O}) \leq \varsigma$ and $T(\mathcal{O}) \leq \tau$	TL: max $\mathbf{E}[R \cdot S(\mathcal{O}) - T(\mathcal{O})]$ s.t. $\forall \mathcal{O}\ \ C(\mathcal{O}) \leq \varsigma$
Linear in cost	LT: max $\mathbf{E}[R \cdot S(\mathcal{O}) - C(\mathcal{O})]$ s.t. $\forall \mathcal{O}\ \ T(\mathcal{O}) \leq \tau$	LL: max $\mathbf{E}[R \cdot S(\mathcal{O}) - C(\mathcal{O}) - T(\mathcal{O})]$

the maximum expected value subject to the constraint that the policy never violates the threshold. In the remainder of the paper, assume without loss of generality that $R = 1$, unless otherwise stated.

1.2. "Batched" policies. The results in this paper concerning the model LT will reflect one more simplifying assumption: that the duration parameters d_i are the same for each source. This assumption is powerful because it allows us to consider scheduling sources in simultaneous "batches": all sources will be scheduled at $t = 0, d, 2d, \ldots$, where d is the common duration.

Although not fully general, this is a reasonable model of the current and probable future state of information access on the Internet. The current common mode of providing information is to supply small amounts of information quickly and cheaply (rather than process large-scale lengthy requests) [16]. As a result the duration for processing a single query relative to the user's time threshold is typically small. Furthermore, the number of providers continues to grow dramatically. In the case in which there are many information providers but each takes a short amount of time, the assumption of equal process duration may be an excellent approximation: the error introduced by assuming equal times will tend to be small relative to the amount of time the user is willing to wait for his information (and thus will not affect the quality of the schedule significantly), yet the sheer number of potential providers will still require an algorithm to choose carefully among its sources since a simple policy of completely serial or parallel queries is liable to be a very bad one.

1.3. Summary of our results. We show that finding an optimal policy is NP-hard for models TT and LT. Reductions from the problems in LT and TT show that even approximating an optimal policy for models LL and TL is NP-hard. A fully polynomial time approximation scheme (FPTAS) is obtained for the model TT, using an extension of the well-known rounding technique for Knapsack [6]. The FPTAS also works for the model TL under a weak assumption: every source is "profitable" individually according to the TL objective function, i.e., for every source s_i, $Rp_i - d_i \geq 0$. The approximation algorithms for the case LT, where the objective function is linear in total cost subject to a time threshold, are perhaps the most interesting technically. We assume that all sources have the same time duration and consider batched policies with a bounded number of batches. We will first present an $O(n^2)$ time approximation algorithm for optimal single-batch policies with ratio $\frac{1}{2}$, and then extend it to a polynomial time approximation scheme (PTAS). For any constant $r > 1$, the PTAS runs in time $O(n^{r+1})$ to achieve an approximation ratio $\frac{r-1}{r+1}$. The algorithms are simple and are similar to the ones in [14] for Knapsack, but the analyses are more sophisticated. We then design an approximation algorithm with ratio $\frac{1}{5}$ for optimal k-batch policies, running in time $O(kn^2)$. The algorithm is based on the ratio $\frac{1}{2}$ algorithm for single-batch policies, but it also involves some new ideas.

1.4. Related work. Scheduling problems have been studied in many contexts including job-shop scheduling, processor allocation, etc. However, our Internet-inspired query scheduling problem has a unique flavor due to the need to balance the competing time and cost constraints on policies with unbounded parallelism. We here consider a number of alternative models that have appeared in the literature, underscoring the difference from our own. If we constrain the policies to be serialized, then an optimal solution can be found in polynomial time (see section 4 for the LT case). Similar problems have been addressed in [4, 8, 12, 17] and elsewhere. The difference in this paper is the ability to query any number of sources in parallel. Papers [3, 5] study scheduling tasks with unlimited parallelism, but their models are different because all tasks have to be executed successfully, whereas in our model a successful answer from any single source suffices. Furthermore, the positive results in [3, 5] are restricted to an exponential time dynamic programming algorithm and some heuristics. Another model of optimal information gathering has recently been studied in [2]. There, the objective is to find a query policy that minimizes the expected value of a linear combination of the total dollar cost and total time cost. A constant ratio approximation algorithm is obtained. Note that their model omits the positive reward associated with the successful completion of a query, which changes the nature of the problem as far as the design of approximation algorithms is concerned.

This paper provides complete proofs and adds new results to the work appearing in [1]. The paper is organized as follows. The hardness results for all four models are given in the next section. Sections 3 and 4 present the approximation algorithms with their analyses for optimal single-batch policies and optimal k-batch policies in the LT model. The FPTASs for the two models involving a cost threshold are given in section 5. The proofs of some technical claims are provided in the appendix.

2. The complexity of computing optimal querying policies.

We first prove that computing an optimal policy in models TT and LT is NP-hard. The proofs are reductions from the Partition problem: Given a finite multiset S of positive integers $w_i \in S$, is there a subset $I \subset S$ such that $\sum_{w_i \in I} w_i = \frac{1}{2}\sum_{w_i \in S} w_i$? The only subtlety is that we have to use exponential numbers in the constructions.

THEOREM 2.1. *Finding an optimal policy in model* TT *is NP-hard.*

Proof. It is clear that any optimal policy in this model is in fact a single-batch policy. Hence we need only to consider single-batch policies. We show that Partition reduces to this problem. Assume that the duration parameters of all the sources are less than the deadline. Then, the expected value of any policy $\mathcal{P}$ in this model is

$$V(\mathcal{P}) = 1 - \prod_{s_i \in \mathcal{P}} (1 - p_i).$$

Thus, maximizing $V(\mathcal{P})$ subject to a cost threshold is equivalent to minimizing $\prod_{s_i \in \mathcal{P}}(1 - p_i)$ under the same constraint, which in turn means maximizing $\sum_{s_i \in \mathcal{P}} - \ln(1 - p_i)$ under the same constraint.

Consider an instance of Partition consisting of a set $S = \{w_1, \dots, w_n\}$ of integers, and let $C = \sum_{i=1}^{n} w_i$. For each source s_i, let its cost $c_i = w_i$, success probability $p_i = 1 - (1 + 1/C)^{-w_i}$, and time duration $d_i = 0$. Take the cost threshold to be $C/2$, and the time threshold to be some positive number. The expression $(1 + 1/C)^{-w_i}$ can be evaluated using the standard repeated squaring technique or a binomial series expansion approximation. It will become clear that we have only to keep at most $3 \log C + \log n$ precision bits during the process.

Clearly, $V(\mathcal{P}) \leq 1-(1+1/C)^{-\frac{C}{2}}$ for any feasible policy $\mathcal{P}$. Now, let $x = \sum_{w_i \in \mathcal{P}} w_i$

and treat x as a continuous variable. Consider function $h(x) = 1-(1+1/C)^{-x}$, which is clearly increasing in x. Since

$$(1+1/C)^{-\frac{C}{2}+1} - (1+1/C)^{-\frac{C}{2}} = (1+1/C)^{-\frac{C}{2}}(1/C) > 1/(\sqrt{e}C),$$

where e is the natural constant, there is a separation of at least $1/(\sqrt{e}C)$ in the value of the function $h(x)$ between point $\frac{C}{2}$ and any point less than or equal to $\frac{C}{2}-1$. If we keep $3\log C + \log n$ bits during calculation of each $1-(1+1/C)^{-w_i}$, then each p_i would have $2\log C + \log n$ precision bits. Thus, the precision of our evaluation of $V(\mathcal{P}) = 1 - \prod_{s_i\in\mathcal{P}}(1-p_i)$ is at least $2\log C$ bits, which is sufficient to allow us to distinguish between the case $\sum_{s_i\in\mathcal{P}} c_i = \frac{C}{2}$ and the case $\sum_{s_i\in\mathcal{P}} c_i < \frac{C}{2}$, due to the above $1/(\sqrt{e}C)$ separation. □

For the problem instance LT we prove a stronger result by showing that a special case of the problem is NP-hard.

THEOREM 2.2. *Finding an optimal single-batch policy for the* LT *objective function is NP-hard.*

Proof. Note that the objective in the single-batch case is to find a set $\mathcal{P}$ of sources to query in parallel such that the function

$$V(\mathcal{P}) = R\left(1 - \prod_{s_i\in\mathcal{P}}(1-p_i)\right) - \sum_{s_i\in\mathcal{P}} c_i \tag{2.1}$$

is maximized. This is equivalent to minimizing the quantity $R\prod_{s_i\in\mathcal{P}}(1-p_i)+\sum_{s_i\in\mathcal{P}} c_i$ over all possible sets of sources.

Consider an instance of Partition consisting of a set $S=\{w_1,\ldots,w_n\}$ of integers, and let $C=\sum_{i=1}^n w_i$. Define the parameters for the optimal single-batch policy problem as follows:

$$\begin{aligned} &c_i = w_i,\\ &p_i = 1-(1+1/C)^{-w_i},\\ &R = [\ln(1+1/C)]^{-1}(1+1/C)^{\frac{C}{2}}. \end{aligned}$$

Again, it will become clear that only $4\log C + \log n$ bits must be kept in the calculation of p_i's and R. For any subset $S_1 \subseteq S$ we have

$$\begin{aligned} \sum_{w_i\in S_1} c_i + R\prod_{w_i\in S_1}(1-p_i) &= \sum_{w_i\in S_1} w_i + [\ln(1+1/C)]^{-1}(1+1/C)^{\frac{C}{2}}\prod_{w_i\in S_1}(1+1/C)^{-w_i}\\ &= \sum_{w_i\in S_1} w_i + [\ln(1+1/C)]^{-1}(1+1/C)^{\frac{C}{2}-\sum_{w_i\in S_1} w_i}. \end{aligned}$$

Again, let $x=\sum_{w_i\in S_1} w_i$ and treat x as a continuous variable. We want to locate the minimum of the following function:

$$h(x) = x + [\ln(1+1/C)]^{-1}(1+1/C)^{\frac{C}{2}-x}.$$

Setting the derivative to zero,

$$\begin{aligned} &h'(x) = 1 + [\ln(1+1/C)]^{-1}[-\ln(1+1/C)](1+1/C)^{\frac{C}{2}-x} = 0\\ \Rightarrow\quad & 1 = [\ln(1+1/C)]^{-1}\ln(1+1/C)(1+1/C)^{\frac{C}{2}-x}\\ \Rightarrow\quad & 1 = (1+1/C)^{\frac{C}{2}-x}\\ \Rightarrow\quad & x = \frac{C}{2}. \end{aligned}$$

We also note that the second derivative of $h(x)$ is always positive, which shows the convexity of the function:

$$h''(x) = [\ln(1+1/C)]^{-1}[\ln(1+1/C)]^2(1+1/C)^{\frac{C}{2}-x} > 0.$$

The following shows that there is a separation of $\Omega(1/C^2)$ in the value of the function $h(x)$ between the points $\frac{C}{2}$ and $\frac{C}{2} \pm 1$(assume that $C > 2$):

$$\begin{aligned}
h\left(\frac{C}{2}-1\right) - h\left(\frac{C}{2}\right) &= C^{-1}[\ln(1+1/C)]^{-1} - 1\\
&> C^{-1}\left(\frac{6C^3}{6C^2-3C+2}\right) - 1\\
&= \frac{3C-2}{6C^2-3C+2}\\
&> 1/(6C^2),\\
h\left(\frac{C}{2}+1\right) - h\left(\frac{C}{2}\right) &= 1 - \left(\frac{1}{C+1}\right)(\ln(1+1/C))^{-1}\\
&> 1 - \left(\frac{1}{C+1}\right)\left(\frac{2C^2}{2C-1}\right)\\
&> 1/(6C^2).
\end{aligned}$$

Therefore, we have only to keep $4\log C + \log n$ precision bits in the calculation of R and p_i's. Hence the reduction can be done in polynomial time. ∎

Hence the problem of deciding whether the expected value of some policy for the single-batch case exceeds a certain threshold is NP-hard. This problem readily reduces to a problem in the LL model where the duration parameters of the sources are all set to the threshold in question. In this case, it is not hard to see that there is a policy with a positive value for the LL model problem if and only if there is a policy for the single-batch model problem with expected value greater than the threshold. Below, by a *positive approximation* we mean constructing a policy which has positive expected value if and only if the value of an optimal policy is positive. Positive approximation is a very relaxed approximation criterion, and we just argued that even positively approximating the optimal in model LL is hard. A similar reduction from model TT shows that positively approximating problems in model TL is NP-hard as well.

THEOREM 2.3. *Positively approximating an optimal policy for the objective functions in the models* TL *or* LL *is NP-hard.*

3. Approximating optimal single-batch policies.

In this and the following sections our focus is on the LT model. In this section we consider policies that send out all their queries in a single batch, i.e., all queries are sent in parallel at time $t = 0$. We present an algorithm that approximates the optimal single-batch policy with ratio 1/2 and then develop a PTAS. Although the PTAS is a straightforward extension of the ratio 1/2 algorithm, its analysis is very different.

Recall again that a single-batch policy is just a set of sources, and our goal is to maximize the objective function in equality (2.1).

The following simple facts and definitions will be useful in this and the next sections. The first lemma shows the subadditivity of the objective function for batched policies.

LEMMA 3.1. *Let* OPT_0 *be an optimal* k*-batch policy. For any partition of* OPT_0 *into two subpolicies* OPT_1 *and* OPT_2*, where the sources in* OPT_1 *and* OPT_2 *are scheduled in the same batches as they are in* OPT_0*,* $V(\mathrm{OPT}_0) \le V(\mathrm{OPT}_1) + V(\mathrm{OPT}_2)$*.*

Proof. We prove it for $k = 2$; the extension to the general k is straightforward. For each $i = 0, 1, 2$ and $j = 1, 2$, let $P_{i,j}$ and $C_{i,j}$ be the collective success probability and cost of the sources in batch j of OPT_i, respectively. Then

$$\begin{aligned}V(\mathrm{OPT}_0) &= P_{0,1} - C_{0,1} + (1 - P_{0,1})(P_{0,2} - C_{0,2})\\ &= P_{1,1} - C_{1,1} + P_{0,1} - P_{1,1} - C_{2,1} + (1 - P_{0,1})(P_{0,2} - C_{0,2}).\end{aligned}$$

Since $P_{0,1} = P_{1,1} + P_{2,1} - P_{1,1}P_{2,1}$,

$$P_{0,1} - P_{1,1} = P_{2,1}(1 - P_{1,1}) \le P_{2,1}.$$

Hence

$$P_{1,1} - C_{1,1} + P_{0,1} - P_{1,1} - C_{2,1} \le P_{1,1} - C_{1,1} + P_{2,1} - C_{2,1}.$$

Similarly, we have

$$P_{0,2} - C_{0,2} \le P_{1,2} - C_{1,2} + P_{2,2} - C_{2,2}.$$

Because OPT_0 is optimal, $P_{1,2} - C_{1,2} \ge 0$ and $P_{2,2} - C_{2,2} \ge 0$. Therefore,

$$\begin{aligned}V(\mathrm{OPT}_0) &\le (P_{1,1} - C_{1,1} + P_{2,1} - C_{2,1}) + (1 - P_{0,1})(P_{1,2} - C_{1,2} + P_{2,2} - C_{2,2})\\ &= (P_{1,1} - C_{1,1} + (1 - P_{0,1})(P_{1,2} - C_{1,2})) + P_{2,1} - C_{2,1}\\ &\quad + (1 - P_{0,1})(P_{2,2} - C_{2,2})\\ &\le (P_{1,1} - C_{1,1} + (1 - P_{1,1})(P_{1,2} - C_{1,2})) + P_{2,1} - C_{2,1}\\ &\quad + (1 - P_{2,1})(P_{2,2} - C_{2,2}),\end{aligned}$$

and thus

$$V(\mathrm{OPT}_0) \le V(\mathrm{OPT}_1) + V(\mathrm{OPT}_2). \qquad \square$$

LEMMA 3.2. *Suppose that* $\mathcal{P}$ *is any* k*-batch policy,* i *is an index between* 1 *and* k*, and* s_j *is a source not appearing in* $\mathcal{P}$*. Let* $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ *denote the subpolicies consisting of the first* $i - 1$ *batches, the* i*th batch, and the last* $k - i$ *batches of* $\mathcal{P}$*, respectively. Also denote the expected cost and collective success probability of the sources in policy* $\mathcal{P}_l$ *as* C_l *and* P_l*,* $l = 1, 2, 3$*. Then adding* s_j *to the* i*th batch of policy* $\mathcal{P}$ *increases its expected value by*

$$\begin{aligned}V(\mathcal{P} \cup \{s_j\}) - V(\mathcal{P}) &= (1 - P_1)(p_j(1 - P_2)(1 - P_3 + C_3) - c_j)\\ &= (1 - P_1)p_j((1 - P_2)(1 - P_3 + C_3) - c_j/p_j).\end{aligned} \tag{3.1}$$

In particular, if $k = i = 1$*, the net increase is*

$$V(\mathcal{P} \cup \{s_j\}) - V(\mathcal{P}) = p_j(1 - P_2) - c_j = p_j(1 - P_2 - c_j/p_j). \tag{3.2}$$

Proof. The expected values of the policies $\mathcal{P}$ and $\mathcal{P} \cup \{s_j\}$ can be written as

$$V(\mathcal{P}) = (P_1 - C_1) + (1 - P_1)((P_2 - C_2) + (1 - P_2)(P_3 - C_3)),$$

```
1.   Sort the sources so that $c_1/p_1 \leq \cdots \leq c_n/p_n$.
2.   APPR = $\emptyset$.   (* APPR is the best policy found so far. *)
3.   For $i := 1$ to $n$
4.     $S := \{s_i\}$.   (* $S$ is the current policy constructed. *)
5.     $Q := 1 - p_i$.   (* $Q$ is the collective failure probability of $S$. *)
6.     For $j := 1$ to $n$, where $j \neq i$
7.       If $Q \geq c_j/p_j$ then (* profitability check *)
8.         $S := S \cup \{s_j\}$.
9.         $Q := Q(1 - p_j)$.
10.      else exit to step 11.
11.    If $V(\text{APPR}) < V(S)$ then APPR := $S$.
12.  Output policy APPR.
```

FIG. 3.1. *The algorithm Pick-a-Star.*

$$V(\mathcal{P}\cup\{s_j\}) = (P_1-C_1)+(1-P_1)((P_2+p_j-P_2p_j-C_2-c_j)+(1-P_2)(1-p_j)(P_3-C_3)).$$

Taking the difference gives us the lemma. □

Thus, in the case of $k = 1$, adding s_j to the policy $\mathcal{P}$ results in an increased expected value if and only if the collective failure probability $1 - P_2$ of the sources in $\mathcal{P}$ is strictly greater than the cost-to-success-probability ratio c_j/p_j of source s_j. Observe that the increased value $p_j(1 - P_2 - c_j/p_j)$ is proportional to the success probability p_j as long as the ratio c_j/p_j is kept constant. Also observe that, for general k, $1 - P_3 + C_3 = 1 - (P_3 - C_3) = 1 - V(\mathcal{P}_3)$. It follows from Lemma 3.2 that a source s_i with $c_i > p_i$ is not useful if $V(\mathcal{P}_3) \geq 0$. Hence we can assume from now on that $p_i \geq c_i$ for all i.

We say that a source s is *profitable in a policy* $\mathcal{P}$ if s is queried in $\mathcal{P}$ and dropping it from $\mathcal{P}$ would not increase the expected value of $\mathcal{P}$. The above lemma states that source s_j in batch i of the k-batch policy $\mathcal{P}$ is profitable in $\mathcal{P}$ if and only if $c_j/p_j \leq (1 - P_{2,j})(1 - V(\mathcal{P}_3))$, where $P_{2,j}$ is the collective success probability of sources in batch i *excluding* source s_j. A policy $\mathcal{P}$ is *irreducible* if every source in $\mathcal{P}$ is profitable in $\mathcal{P}$. Clearly, every optimal k-batch policy is irreducible.

3.1. A ratio $\frac{1}{2}$ approximation algorithm. Our algorithm, as shown in Figure 3.1, is somewhat similar to the greedy approximation algorithm for Knapsack given in [14], though the analysis of its performance is more complex.

The algorithm Pick-a-Star sorts the sources in ascending order of the ratio c_i/p_i. It then goes over each source s_i, picks it, and then picks the rest from the sorted list (with s_i removed) until either the list is exhausted or it reaches a source s_j which cannot be profitable, meaning that the profitability criterion

$$\prod_{k=i \text{ or } k<j} (1 - p_k) \geq c_j/p_j$$

is not satisfied. Equality (3.2) in Lemma 3.2 and the comments following the lemma explain the choice of the criterion. Pick-a-Star keeps track of the policy with the highest expected value over the iterations. Clearly the running time is $O(n^2)$.

Now we analyze the performance of Pick-a-Star and show that it results in an expected value that is at least half of the optimum. Let APPR be the policy obtained by Pick-a-Star and OPT be an optimal single-batch policy. Since Pick-a-Star picks the first source optimally (i.e., through exhaustive search), $V(\text{APPR}) \geq V(\{s_i\}) = p_i - c_i$

for all $i \leq n$. Thus, without loss of generality, we may assume $|\text{APPR}| > 1$. Moreover, we will assume henceforth that the first source picked by Pick-a-Star is the "most profitable" source in OPT, i.e., some source s_i with the maximum $V(\{s_i\})$ over all sources in OPT. Let s_{last} be the last source picked by Pick-a-Star. We can assume that the collective failure probability of APPR is at least the ratio c_{last}/p_{last}, because otherwise we could modify APPR by decreasing p_{last} while keeping c_{last}/p_{last} constant until the collective failure probability of APPR becomes equal to c_{last}/p_{last}. This is possible since the collective failure probability of $\text{APPR} - \{s_{last}\}$ is greater than c_{last}/p_{last}. By Lemma 3.2 such modification could only worsen the expected value of APPR. Note that if s_{last} appears also in OPT, then it is treated as a different copy and kept intact. Hence we do not change the expected value of OPT in this case (or in any other case). Note also that this potential modification does not affect the first source picked by Pick-a-Star since $|\text{APPR}| > 1$.

Define $S_0 = \text{APPR} \cap \text{OPT}$, $S_1 = \text{APPR} - S_0$, and $S_2 = \text{OPT} - S_0$. For each $i = 0, 1, 2$, let C_i and P_i be the collective cost and success probability of the sources in S_i. Note that if the success probability of source s_{last} is modified in APPR as mentioned above and s_{last} also appears in OPT, then the two copies of s_{last} in APPR and OPT are viewed as distinct sources and thus s_{last} will not be included in S_0. On the other hand, if s_{last} is not modified in APPR and s_{last} appears in OPT, then the two copies of s_{last} in APPR and OPT are viewed as the same source and thus s_{last} will be included in S_0. Observe that

$$\forall s_i \in S_1 \forall s_j \in S_2, \frac{c_i}{p_i} \leq \frac{c_j}{p_j}. \tag{3.3}$$

Let us first consider the (easier) case in which $S_2 = \emptyset$. Observe that $S_1 \subseteq \{s_1, \ldots, s_{last}\}$. Since the collective failure probability of $\text{APPR} - \{s_{last}\}$ is greater than $c_{last}/p_{last} \geq \cdots \geq c_1/p_1$, every element of S_1 is profitable in the set $\text{APPR} - \{s_{last}\}$. By Lemma 3.2, $V(\text{APPR} - \{s_{last}\}) \geq V(\text{OPT} - \{s_{last}\})$. We also know that $V(\text{APPR}) \geq V(\text{APPR} - \{s_{last}\})$ by Lemma 3.2. Since $V(\text{APPR}) \geq V(\{s_{last}\})$ and $V(\text{OPT}) \leq V(\text{OPT} - \{s_{last}\}) + V(\{s_{last}\})$ by Lemma 3.1,

$$2V(\text{APPR}) \geq V(\text{OPT} - \{s_{last}\}) + V(\{s_{last}\}) \geq V(\text{OPT}).$$

Now suppose that $S_2 \neq \emptyset$. Since OPT is irreducible and Pick-a-Star picks sources until no remaining source can be profitable, $S_1 \neq \emptyset$. Let $m = |S_2|$ and $l = |S_1|$. Let

$$\alpha_1 = \max_{s_i \in S_1} \frac{c_i}{p_i(1-P_0)} \leq \frac{c_{last}}{p_{last}(1-P_0)},$$
$$\alpha_2 = \min_{s_i \in S_2} \frac{c_i}{p_i(1-P_0)}.$$

By relation 3.3, clearly $\alpha_1 \leq \alpha_2$. The next lemma relating α_1, α_2 to P_1, P_2 is a key to our analysis.

LEMMA 3.3. (i) $\alpha_1 \leq 1 - P_1 \leq \alpha_2$ *and* (ii) $1 - P_2 \geq \alpha_2^{\frac{m}{m-1}}$.

Proof. Recall that we have assumed that $(1 - P_0)(1 - P_1) \geq c_{last}/p_{last}$. Thus $1 - P_1 \geq \alpha_1$. Since Pick-a-Star stopped before picking anything from S_2, $(1 - P_0)(1 - P_1) \leq c_i/p_i$ for any $s_i \in S_2$. These prove (i). To prove (ii), let $p_{min} = \min_{s_i \in S_2} p_i$. Since OPT is irreducible, $(1 - P_0)(1 - P_2)/(1 - p_{min}) \geq c_{min}/p_{min} \geq (1 - P_0)\alpha_2$. So,

$$(1 - P_2)^{\frac{m-1}{m}} \geq (1 - P_2)/(1 - p_{min}) \geq \alpha_2,$$

i.e., $1 - P_2 \geq \alpha_2^{\frac{m}{m-1}}$. ◻

Now we want to find a lower bound for the ratio

$$\frac{V(\text{APPR})}{V(\text{OPT})} = \frac{P_0 - C_0 + (1-P_0)P_1 - C_1}{P_0 - C_0 + (1-P_0)P_2 - C_2}. \tag{3.4}$$

Since $V(S_0) \geq V(S_2)/m$ by the choice of the first source picked by Pick-a-Star and the fact that S_0 is irreducible,

$$V(\text{OPT}) \leq V(S_0) + V(S_2) \leq (m+1)V(S_0).$$

This implies

$$\frac{(1-P_0)P_2 - C_2}{P_0 - C_0 + (1-P_0)P_2 - C_2} \leq \frac{m}{m+1}.$$

Define

$$r = \frac{(1-P_0)P_1 - C_1}{(1-P_0)P_2 - C_2}. \tag{3.5}$$

To obtain a lower bound of 1/2 for the ratio in equality (3.4), we need

$$\frac{1}{m+1} + r\frac{m}{m+1} \geq \frac{1}{2}, \quad \text{i.e.,} \quad r \geq \frac{m-1}{2m}, \tag{3.6}$$

which we show below. The following lemma gives a clean lower bound for ratio r.

LEMMA 3.4.

$$r \geq \min_{\alpha_1 \leq 1-P_1 \leq \alpha_2} \frac{P_1 - l(1-(1-P_1)^{1/l})\alpha_1}{(1-\alpha_2^{m/(m-1)})(1-\alpha_2)}.$$

Proof. Observe that

$$\begin{aligned} r &= \frac{(1-P_0)P_1 - (\sum_{s_i \in S_1} p_i \frac{c_i}{p_i})}{(1-P_0)P_2 - C_2} \\ &\geq \frac{(1-P_0)P_1 - (\sum_{s_i \in S_1} p_i)(1-P_0)\alpha_1}{(1-P_0)P_2 - C_2} \\ &\geq \frac{P_1 - (\sum_{s_i \in S_1} p_i)\alpha_1}{P_2 - C_2/(1-P_0)} \\ &\geq \frac{P_1 - (\sum_{s_i \in S_1} p_i)\alpha_1}{P_2 - P_2\alpha_2} \\ &\geq \frac{P_1 - (\sum_{s_i \in S_1} p_i)\alpha_1}{(1-\alpha_2^{m/(m-1)})(1-\alpha_2)}. \end{aligned}$$

The last step follows from (ii) of Lemma 3.3. Since

$$\sum_{s_i \in S_1} (1-p_i) \geq l \left[\prod_{s_i \in S_1} (1-p_i) \right]^{1/l} = l(1-P_1)^{1/l},$$

$\sum_{s_i \in S_1} p_i \leq l - l(1-P_1)^{1/l}$. Hence the lemma follows from (i) of Lemma 3.3. ◻

Now we try to simplify the lower bound function.[3]

CLAIM 3.5. *The ratio*

$$\frac{P_1 - l(1-(1-P_1)^{1/l})\alpha_1}{(1-\alpha_2^{m/(m-1)})(1-\alpha_2)}$$

is increasing in P_1 when $1 - P_1 \geq \alpha_1$.

By Lemma 3.3, the smallest P_1 can be is $1 - \alpha_2$. The above ratio is clearly decreasing in $\alpha_1 \leq \alpha_2$. For convenience, let $x = \alpha_2$. We set $P_1 = 1 - x$ and $\alpha_1 = x$, and we get

$$r \geq \frac{1}{1-x^{m/(m-1)}}\left[\frac{1-x-l(1-x^{1/l})x}{1-x}\right]. \tag{3.7}$$

CLAIM 3.6. *The right-hand side of inequality* (3.7) *is nonincreasing in l.*

Taking the limit

$$\lim_{l\to\infty} l(1-x^{1/l}) = \lim_{l\to\infty}\frac{1-x^{1/l}}{1/l} = \lim_{l\to\infty}\frac{(-x^{1/l})(\ln x)(-1/l^2)}{-1/l^2} = -\ln x,$$

we get

$$r \geq \frac{1}{1-x^{m/(m-1)}}\left[\frac{1-x+x\ln x}{1-x}\right]. \tag{3.8}$$

CLAIM 3.7. *The right-hand side of inequality* (3.8) *is decreasing in $x \in (0,1)$.*

Since the right-hand side expression is undefined at $x = 1$, we take the limit

$$\begin{aligned}
r &\geq \lim_{x\to1}\left[\frac{1}{1-x^{m/(m-1)}}\left(\frac{1-x+x\ln x}{1-x}\right)\right]\\
&= \lim_{x\to1}\frac{1-x+x\ln x}{1-x-x^{\frac{m}{m-1}}+x^{\frac{2m-1}{m-1}}}\\
&= \lim_{x\to1}\frac{-1+\ln x+x/x}{-1-\frac{m}{m-1}x^{\frac{1}{m-1}}+\frac{2m-1}{m-1}x^{\frac{m}{m-1}}}\\
&= \lim_{x\to1}\frac{1/x}{-\frac{m}{m-1}\frac{1}{m-1}x^{\frac{2-m}{m-1}}+\frac{2m-1}{m-1}\frac{m}{m-1}x^{\frac{1}{m-1}}}\\
&= \frac{1}{-\frac{m}{(m-1)^2}+\frac{m(2m-1)}{(m-1)^2}} = \frac{1}{\frac{m}{m-1}(\frac{-1+2m-1}{m-1})}\\
&= \frac{m-1}{2m}.
\end{aligned}$$

This verifies inequality (3.6) and completes the proof that $V(\text{APPR})/V(\text{OPT}) \geq 1/2$.

THEOREM 3.8. *Pick-a-Star produces a single-batch policy with an expected value that is at least half of the optimum.*

[3]The proofs of the claims appear in the appendix.

3.2. Extending Pick-a-Star to a PTAS. The extension of the algorithm is straightforward. Let $r \geq 1$ be any fixed constant. The new algorithm iterates over all possible choices of at most r sources and schedules the rest of the sources based on the cost-to-success probability ratio using the same stopping criterion. It then outputs the best policy found in all iterations. Call the new algorithm Pick-r-Stars. Clearly, it runs in $O(n^{r+1})$ time. We show that Pick-r-Stars achieves an approximation ratio of $\frac{r-1}{r+1}$. The analysis is different from the previous subsection in that we will make use of the r sources in the optimal policy with the highest success probability instead of the the most profitable ones. We would like to remark here that this new strategy does not work for Pick-a-Star, nor does our analysis of Pick-a-Star work for general r because the best lower bound that the analysis yields for the ratio defined in equality (3.5) is $\frac{m-1}{2m}$.

Let APPR be the policy found by Pick-r-Stars and OPT an optimal policy. As in the previous subsection, we assume without loss of generality that (i) $|\text{APPR}| > r$ and $|\text{OPT}| > r$, (ii) APPR contains the r sources in OPT with the highest success probability, and (iii) the collective failure probability of APPR is at least the ratio c_{last}/p_{last}, where s_{last} is the last source picked by Pick-r-Stars. Since OPT is irreducible, we can also assume that APPR $\not\subseteq$ OPT, because otherwise APPR = OPT.

Again, let $S_0 = \text{APPR} \cap \text{OPT}$, $S_1 = \text{APPR} - S_0$, and $S_2 = \text{OPT} - S_0$ and the corresponding collective costs and success probabilities C_i and P_i, for each $i = 0, 1, 2$. We also have $c_i/p_i \leq c_j/p_j$ for all $s_i \in S_1, s_j \in S_2$. Define $l = |S_1|$, $m = |S_2|$, and

$$\begin{aligned}
\alpha_0 &= \max_{s_i \in S_0} \frac{c_i}{p_i}, \\
\alpha_1 &= \max_{s_i \in S_1} \frac{c_i}{p_i} \leq \frac{c_{last}}{p_{last}(1-P_0)}, \\
\alpha_2 &= \min\left\{1,\ \min_{s_i \in S_2} \frac{c_i}{p_i}\right\}.
\end{aligned}$$

Then we again have

$$\alpha_1 \leq (1-P_0)(1-P_1) \leq \alpha_2. \tag{3.9}$$

To obtain a clean lower bound for the approximation ratio $V(\text{APPR})/V(\text{OPT})$, we go through a sequence of simplifying steps. In the process we will guarantee that the ratio $V(\text{APPR})/V(\text{OPT})$ never improves and inequality (3.9) always holds.

First, we will assume that $|S_0| = r$, i.e., APPR and OPT share exactly r common sources, by the following argument. Let $s_i \in S_0$ be any source. Then

$$\begin{aligned}
\frac{V(\text{APPR})}{V(\text{OPT})} &= \frac{1-(1-P_0)-C_0+(1-P_0)P_1-C_1}{1-(1-P_0)-C_0+(1-P_0)P_2-C_2} \\
&= \frac{\frac{1}{1-p_i}-\frac{1-P_0}{1-p_i}-\frac{C_0}{1-p_i}+\frac{(1-P_0)P_1}{1-p_i}-\frac{C_1}{1-p_i}}{\frac{1}{1-p_i}-\frac{1-P_0}{1-p_i}-\frac{C_0}{1-p_i}+\frac{(1-P_0)P_2}{1-p_i}-\frac{C_2}{1-p_i}} \\
&= \frac{\frac{1-c_i}{1-p_i}-\frac{1-P_0}{1-p_i}-\frac{C_0-c_i}{1-p_i}+\frac{(1-P_0)P_1}{1-p_i}-\frac{C_1}{1-p_i}}{\frac{1-c_i}{1-p_i}-\frac{1-P_0}{1-p_i}-\frac{C_0-c_i}{1-p_i}+\frac{(1-P_0)P_2}{1-p_i}-\frac{C_2}{1-p_i}} \\
&> \frac{1-\frac{1-P_0}{1-p_i}-\frac{C_0-c_i}{1-p_i}+\frac{(1-P_0)P_1}{1-p_i}-\frac{C_1}{1-p_i}}{1-\frac{1-P_0}{1-p_i}-\frac{C_0-c_i}{1-p_i}+\frac{(1-P_0)P_2}{1-p_i}-\frac{C_2}{1-p_i}}.
\end{aligned}$$

The last step holds because $p_i - c_i = V(\{s_i\}) > 0$ and thus $1 - c_i > 1 - p_i$. Hence we can define a new pair of APPR and OPT by removing the source s_i and dividing the cost of every remaining source in S_0 by $1 - p_i$. Clearly inequality (3.9) still holds for the new pair.

Second, we can assume that $\alpha_1 = \alpha_2 = (1 - P_0)(1 - P_1)$. This can be achieved by increasing the cost of each source in S_1 and decreasing the cost of each source in S_2. So now, $c_i = \alpha_1 p_i$ for each $s_i \in S_1 \cup S_2$.

Third, we assume without loss of generality that the r sources in S_0 have the same success probability $p = 1 - (1 - P_0)^{1/r}$. Since these sources are assumed to have the largest success probability in OPT, $p_i \le p$ for each $s_i \in S_2$. Moreover, we can assume that $p_i = p$ for each $s_i \in S_2$ by the following argument. If $p_i < p$, we increase p_i to p and c_i to $\alpha_1 p$. By Lemma 3.2, this improves $V(\text{OPT})$ because the set OPT is irreducible. If this results in a set that is not irreducible, we can make it irreducible by dropping some sources in S_2, again improving the expected value of OPT.

Fourth, we can assume that $c_i/p_i = \alpha_0$ for all $s_i \in S_0$. The condition can be achieved by increasing the costs of the sources in S_0. This would decrease the expected value of both APPR and OPT by the same amount and thus decrease the ratio $V(\text{APPR})/V(\text{OPT})$.

Finally, we can worsen APPR by assuming that

$$p_i = 1 - (1 - P_1)^{1/l} = 1 - (\alpha_1/(1 - P_0))^{1/l} = 1 - (\alpha_1/(1 - p)^r)^{1/l}$$

for each $p_i \in S_1$, as mentioned in the previous subsection.

Now we have clean formulas for the expected values:

$$\begin{aligned} V(\text{APPR}) &= 1 - (1 - p)^r \frac{\alpha_1}{(1 - p)^r} - \alpha_0 r p - l\alpha_1 \left[1 - \left(\frac{\alpha_1}{(1 - p)^r}\right)^{1/l}\right], \\ V(\text{OPT}) &= 1 - (1 - p)^{r+m} - \alpha_0 r p - \alpha_1 m p. \end{aligned}$$

We further simplify the formulas by getting rid of l and m.

LEMMA 3.9.

$$V(\text{APPR}) \ge 1 - \alpha_1 - \alpha_0 r p + \alpha_1 \ln\left(\frac{\alpha_1}{(1 - p)^r}\right).$$

Proof. Since $\alpha_1/(1 - p)^r < 1$ by inequality 3.9, the function $-l\alpha_1[1 - (\frac{\alpha_1}{(1-p)^r})^{1/l}]$ is nonincreasing in l by Claim 3.6 in the last subsection. Taking the limit, we get $V(\text{APPR}) \ge \lim_{l\to\infty} 1 - \alpha_1 - \alpha_0 r p - l\alpha_1(1 - (\frac{\alpha_1}{(1-p)^r})^{1/l})$ or

$$V(\text{APPR}) \ge 1 - \alpha_1 - \alpha_0 r p + \alpha_1 \ln\left(\frac{\alpha_1}{(1 - p)^r}\right). \qquad \square$$

LEMMA 3.10.

$$V(\text{OPT}) \le 1 + \frac{\alpha_1 p}{\ln(1 - p)} - \alpha_0 r p - \frac{\alpha_1 p}{\ln(1 - p)} \ln\left[\frac{-\alpha_1 p}{(1 - p)^r \ln(1 - p)}\right].$$

Proof. We can treat m as a continuous variable and maximize $1 - (1 - p)^{r+m} - \alpha_0 r p - \alpha_1 m p$ over all real values of m. Taking the derivative with respect to m and setting it to 0, we get

$$m = \frac{1}{\ln(1 - p)} \ln\left(\frac{-\alpha_1 p}{(1 - p)^r \ln(1 - p)}\right).$$

Hence

$$\max_m 1-(1-p)^{r+m}-\alpha_0 rp-\alpha_1 mp = 1+\frac{\alpha_1 p}{\ln(1-p)}$$
$$-\alpha_0 rp-\frac{\alpha_1 p}{\ln(1-p)}\ln\frac{-\alpha_1 p}{(1-p)^r\ln(1-p)}. \qquad \square$$

Now we are ready to show that $V(\text{APPR})/V(\text{OPT}) \geq (r-1)/(r+1)$. It suffices to prove that

$$\frac{V(\text{OPT})-V(\text{APPR})}{V(\text{APPR})} \leq \frac{2}{r-1}. \tag{3.10}$$

We first give an overestimate of the difference $V(\text{OPT})-V(\text{APPR})$. The following simple mathematical facts for $0<p<1$ will be useful:

$$\begin{aligned}
p \ < \ -\ln(1-p) &= p+\frac{p^2}{2}+\frac{p^3}{3}+\cdots < \ \frac{p}{1-p},\\
0 \ < \ 1+\frac{p}{\ln(1-p)} &= 1-\frac{1}{1+p/2+p^2/3+\cdots}\\
&< 1-\frac{1}{1+p+p^2+\cdots}\\
&= p\\
-\ln\left(\frac{-p}{\ln(1-p)}\right) &= -\ln\left(1-\left(1+\frac{p}{\ln(1-p)}\right)\right)\\
&> 1+\frac{p}{\ln(1-p)}.
\end{aligned}$$

Let $f_1(\alpha_1)=\alpha_1 p^2-\alpha_1 p\ln(\frac{\alpha_1}{(1-p)^r})$.

LEMMA 3.11. $V(\text{OPT})-V(\text{APPR})<f_1(\alpha_1)$.

Proof. From Lemmas 3.9 and 3.10, we know

$$\begin{aligned}
V(\text{OPT})-V(\text{APPR}) \leq{}& \alpha_1\left(1+\frac{p}{\ln(1-p)}\right)-\alpha_1\left(1+\frac{p}{\ln(1-p)}\right)\ln\left(\frac{\alpha_1}{(1-p)^r}\right)\\
&-\frac{\alpha_1 p}{\ln(1-p)}\ln\left(\frac{-p}{\ln(1-p)}\right).
\end{aligned}$$

Therefore,

$$\begin{aligned}
V(\text{OPT})-V(\text{APPR}) &< \alpha_1\left(1+\frac{p}{\ln(1-p)}\right)-\alpha_1 p\ln\left(\frac{\alpha_1}{(1-p)^r}\right)\\
&\quad+\frac{\alpha_1 p}{\ln(1-p)}\left(1+\frac{p}{\ln(1-p)}\right)\\
&= \alpha_1\left(1+\frac{p}{\ln(1-p)}\right)^2-\alpha_1 p\ln\left(\frac{\alpha_1}{(1-p)^r}\right)
\end{aligned}$$

or

$$V(\text{OPT})-V(\text{APPR}) \leq \alpha_1 p^2-\alpha_1 p\ln\left(\frac{\alpha_1}{(1-p)^r}\right). \qquad \square$$

Next we find an underestimate of $V(\text{APPR})$. Observe that the following conditions follow from inequality (3.9), the simplifying assumptions on the sources in S_i (e.g., $\forall\, s_i \in S_0, c_i/p_i = \alpha_0$), and the arguments for the second claim of Lemma 3.3:

$$(1-p)^r > \alpha_1,$$

$$(1-p)^{r-1} = ((1-p)^r)^{\frac{r-1}{r}} > \alpha_0.$$

Let $f_2(\alpha_1) = \frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1 \ln(\frac{\alpha_1}{(1-p)^r}) + (1-p)^r - \alpha_1$.

Lemma 3.12.

$$V(\text{APPR}) > f_2(\alpha_1) = \frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1.$$

Proof.

$$\begin{aligned} V(\text{APPR}) &\geq 1 - \alpha_1 - \alpha_0 rp + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) \\ &= 1 - (1-p)^r - \alpha_0 rp + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1 \\ &> 1 - (1-p)^r - (1-p)^{r-1} rp + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1 \\ &= p^2(1 + 2(1-p) + \cdots + (r-1)(1-p)^{r-2}) + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) \\ &+ (1-p)^r - \alpha_1. \end{aligned}$$

Thus

$$V(\text{APPR}) > \frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1. \qquad \square$$

Hence we have the following clean lower bound on the ratio $\frac{V(\text{OPT})-V(\text{APPR})}{V(\text{APPR})}$.

Lemma 3.13.

$$\frac{V(\text{OPT}) - V(\text{APPR})}{V(\text{APPR})} > \min_{\alpha_1 < (1-p)^r} \frac{f_1(\alpha_1)}{f_2(\alpha_1)}.$$

Lemma 3.14. *For all* $\alpha_1 < (1-p)^r$, $f_1(\alpha_1)/f_2(\alpha_1) > 1/(r-1)$.

Proof. First consider the case $\alpha_1 \geq (1-p)^{2r-1}$. Let $\alpha_1 = (1-p)^x$. Hence $r < x \leq 2r-1$. Observe that because the function $y \ln y + 1 - y$ is positive for all $y > 0$,

$$\alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1 = (1-p)^r \left(\frac{\alpha_1}{(1-p)^r} \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + 1 - \frac{\alpha_1}{(1-p)^r}\right) > 0.$$

This means $f_2(\alpha_1) > \frac{r(r-1)}{2}(1-p)^{r-2}p^2$. Thus

$$\frac{f_1(\alpha_1)}{f_2(\alpha_2)} < \frac{(1-p)^x p^2 - (x-r)(1-p)^x p \ln(1-p)}{\frac{r(r-1)}{2}(1-p)^{r-2}p^2}$$

$$
\begin{aligned}
&< \frac{(1-p)^x p^2 + (x-r)(1-p)^{x-1}p^2}{\frac{r(r-1)}{2}(1-p)^{r-2}p^2} \\
&< \frac{1+x-r}{r(r-1)/2} \\
&\le \frac{2r}{r(r-1)} \\
&= \frac{2}{r-1}.
\end{aligned}
$$

When $\alpha_1 \le (1-p)^{2r-1}$, we claim that the function $f_2(\alpha_1) - \frac{r-1}{2} f_1(\alpha_1)$ is decreasing in α_1.

CLAIM 3.15. *The function*

$$
\begin{aligned}
f_2(\alpha_1) - \frac{r-1}{2} f_1(\alpha_1) = {}& \frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) \\
&+ (1-p)^r - \alpha_1 - \frac{r-1}{2}\left(\alpha_1 p^2 - \alpha_1 p \ln\left(\frac{\alpha_1}{(1-p)^r}\right)\right)
\end{aligned}
$$

is decreasing in α_1 *for* $\alpha_1 \le (1-p)^{2r-1}$.

Hence, for any $\alpha_1 \le (1-p)^{2r-1}$,

$$
f_2(\alpha_1) - \frac{r-1}{2} f_1(\alpha_1) \ge f_2((1-p)^{2r-1}) - \frac{r-1}{2} f_1((1-p)^{2r-1}) > 0. \qquad \square
$$

This concludes the analysis for Pick-r-Stars.

THEOREM 3.16. *Pick-r-Stars produces a single-batch policy with an expected value that is at least* $(r-1)/(r+1)$ *of the optimum.*

4. Approximating optimal k-batch policies.

Here we present an algorithm for the linear cost and time threshold (LT) model that approximates optimal k-batch policies with a constant ratio 1/5. Recall that our simplifying assumption here is that time durations are equal; consequently, optimal batched policies exist. The k-batch approximation algorithm, called Reverse-Greedy, is illustrated in Figure 4.1. The algorithm works by constructing an irreducible policy. It greedily constructs the policy batch by batch, starting from the last batch (rightmost) and going in reverse time. For each batch it invokes the single-batch algorithm Pick-a-Star, but with a modified profitability criterion that follows from equality (3.1) of Lemma 3.2 (see the comments following the lemma). Even though each source is profitable at the time it is added to a partially constructed batch, the completed batch may not be irreducible, i.e., some of the sources picked before the last source may become nonprofitable after the addition of the last source. Thus after each call to Pick-a-Star, the algorithm scans back over the newly created batch and drops any source that is nonprofitable. In this way, the final policy is surely irreducible. Clearly Reverse-Greedy can be implemented to run in time $O(kn^2)$.

The analysis of Reverse-Greedy makes use of Theorem 3.8. The difficulty here is that because the sources can be scheduled in different batches, some batches of an optimal k-batch policy could be arbitrarily better individually than their counterparts in APPR. To get around this, we relate an irreducible k-batch policy to its optimally serialized version. For any policy $\mathcal{P}$, let $\overline{\mathcal{P}}$ denote the optimal serial policy for the sources in $\mathcal{P}$. It is not hard to see that $V(\overline{\mathcal{P}}) \ge V(\mathcal{P})$. Note that a serial policy $\overline{\mathcal{P}}$ may violate the time threshold, however we use $V(\overline{\mathcal{P}})$ in the analysis as a means

```
1.  Sort the sources so that $c_1/p_1 \leq \cdots \leq c_n/p_n$.
2.  APPR $= \emptyset$.    (* APPR denotes the $k$-batch policy. *)
3.  For $i := k$ downto 1
4.    $S = \emptyset$.    (* $S$ is the best $i$th batch found so far. *)
5.    $Q = 1$.    (* $Q$ is the collective failure probability of $S$. *)
6.    For $j := 1$ to $n$, where $s_j \notin$ APPR
7.      $S_1 := \{s_j\}$.
8.      $Q_1 := 1 - p_j$.    (* $Q_1$ is the collective failure probability of $S_1$. *)
9.      For $l := 1$ to $n$, where $l \neq j$ and $s_l \notin$ APPR
10.       If $Q_1(1 - V(APPR)) \geq c_l/p_l$ then    (* profitability check *)
11.         $S_1 := S_1 \cup \{s_l\}$.
12.         $Q_1 := Q_1(1 - p_l)$.
13.       else exit to step 14.
14.     If $V(S) < V(S_1)$ then $S := S_1; Q = Q_1$.
15.   For each $s_j$ in $S$    (* Make $S$ irreducible. *)
16.     If $c_j/p_j > \frac{Q}{1-p_j}(1 - V(APPR))$
17.       $S = S - \{s_j\}$; $Q = Q/(1 - p_j)$;
18.   Add $S$ to APPR as the $i$th batch.
19. Output policy APPR.
```

FIG. 4.1. *The algorithm Reverse-Greedy.*

of bounding the value of an optimal scheduling of the sources in $\mathcal{P}$. First, let us characterize an optimal serial policy.

LEMMA 4.1. *For any set of sources, an optimal serial policy (including all sources in the set) sorts the sources in the nondecreasing order of their cost to success probability ratios.*

Proof. Consider any serial policy $\mathcal{P} = s_{i_1}, \ldots, s_{i_m}$. Then

$$V(\mathcal{P}) = p_{i_1} - c_{i_1} + (1 - p_{i_1})(p_{i_2} - c_{i_2}) + \cdots + \prod_{j=1}^{m-1}(1 - p_{i_j})(p_{i_m} - c_{i_m}).$$

Hence swapping source s_{i_j} and source $s_{i_{j+1}}$ would result in a net value of

$$\begin{aligned}&\prod_{l=1}^{j-1}(1 - p_{i_l})(p_{i_{j+1}} - c_{i_{j+1}} + (1 - p_{i_{j+1}})(p_{i_j} - c_{i_j}))\\ &- \prod_{l=1}^{j-1}(1 - p_{i_l})(p_{i_j} - c_{i_j} + (1 - p_{i_j})(p_{i_{j+1}} - c_{i_{j+1}}))\\ &\qquad = \prod_{l=1}^{j-1}(1 - p_{i_l})(p_{i_{j+1}}c_{i_j} - p_{i_j}c_{i_{j+1}}),\end{aligned}$$

which is positive if $c_{i_{j+1}}/p_{i_{j+1}} < c_{i_j}/p_{i_j}$. ∎

The following property of optimal serial policies is a side product of the above lemma and will be useful in the analysis.

COROLLARY 4.2. *Let S_1 and S_2 be two sets of sources and $S_1 \subseteq S_2$. An optimal serial policy for S_2 gives an expected value at least that of an optimal serial policy for S_1.*

Proof. Recall our basic assumption from section 3 that $p_i \geq c_i$ for all sources s_i. As a consequence an optimal serial policy for the sources in S_2 exists that includes all the sources in S_2. Starting from such an optimal serial policy for S_2, we can gradually swap the sources that are not in S_1 towards the end of the sequence and eventually remove them. By Lemma 4.1, no such swap or removal can increase the expected value. ▯

The following lemma, which is somewhat surprising, is a key to our analysis. It states that serializing an irreducible batched policy can at most triple the expected value.

LEMMA 4.3. *Let P denote the success probability of an irreducible batched policy $\mathcal{P}$. Then $V(\overline{\mathcal{P}}) \leq (2+P)V(\mathcal{P})$.*

Proof. The proof is by induction on the number of sources of the policy. Assume that the statement holds for any irreducible policy consisting of $n-1 \geq 1$ sources, and consider an irreducible policy $\mathcal{P}$ consisting of n sources. Let S denote the set of sources in the first (leftmost) batch and s_n denote a source in S. In order to maximize the difference between $V(\overline{\mathcal{P}})$ and $V(\mathcal{P})$ we assume that the cost-to-success-probability ratios of all the sources in S are at the maximum possible (without violating the irreducibility condition). We perturb the initial policy $\mathcal{P}$ and transform it into another irreducible policy $\mathcal{P}'$, by moving s_n to the left in the time-line so that it finishes before all the other sources. We increase the cost of source s_n and other sources in S by amounts to be described below. Note that policy $\mathcal{P}'$ will not be different from policy $\mathcal{P}$ if $S - \{s_n\} = \emptyset$. Due to the potential increase in costs in the transformation, we will see that $V(\mathcal{P}') \leq V(\mathcal{P})$ and $V(\overline{\mathcal{P}'}) \leq V(\overline{\mathcal{P}})$. We first show that

$$2\,(V(\mathcal{P}) - V(\mathcal{P}')) \geq V(\overline{\mathcal{P}}) - V(\overline{\mathcal{P}'}). \tag{4.1}$$

We complete the proof by showing, using the inductive hypothesis, that the statement of the lemma holds for policy $\mathcal{P}'$.

Let Q denote the collective failure probability of the sources in S and V_a denote the expected value of policy $\mathcal{P}$ without batch S. Let $S' = S - \{s_n\}$. Since the cost-to-success-probability ratios of sources in S are as high as possible, we have $c_i/p_i = Q/(1-p_i)(1-V_a)$ for $s_i \in S$ according to the definition of irreducibility. Let c'_n and c'_i be the costs of sources s_n and $s_i \in S'$ in policy $\mathcal{P}'$. Again, we set these costs to be as high as they can be: $c'_i = p_i(Q/(1-p_i)(1-p_n))(1-V_a) = c_i/(1-p_n)$ and $c'_n = p_n(Q/(1-p_n) + \sum_{s_i\in S'} c'_i - Q/(1-p_n)V_a)$. Therefore we have that $\forall s_i \in S', c'_i - c_i = c'_i - (1-p_n)c'_i = c'_i p_n$ and $c'_n - c_n = \sum_{s_i\in S'} p_n c'_i$. Thus the change in the cost of s_n is equal to the total change in the costs of the sources in S'.

We have $V(\mathcal{P}') = 1 - Q - c'_n - (1-p_n)\sum_{s_i\in S'} c'_i + QV_a$, and $V(\mathcal{P}) = 1 - Q - \sum_{s_i\in S} c_i + QV_a$. Thus $V(\mathcal{P}) - V(\mathcal{P}') = c'_n - c_n = p_n \sum_{s_i\in S'} c'_i$. In other words, in going from policy $\mathcal{P}'$ to policy $\mathcal{P}$, only the reduction in the cost of source s_n is felt in the increase of the expected value. Observe that this reduction is exactly half of the total change in all costs. Now it is easy to see that $2\,(V(\mathcal{P}) - V(\mathcal{P}')) \geq V(\overline{\mathcal{P}}) - V(\overline{\mathcal{P}'})$: In the transition from the optimal serial policy $\overline{\mathcal{P}'}$ to the optimal serial policy $\overline{\mathcal{P}}$, the reduction in the costs of both source s_n and the sources in S', which is twice the reduction in the cost of source s_n, is reflected in the increase of the expected value. However, $V(\overline{\mathcal{P}}) - V(\overline{\mathcal{P}'}) \leq 2p_n \sum_{s_i\in S'} c'_i$, since in a serial policy the effect of a cost reduction on each source is reduced due to the success probability of the sources queried earlier.

Next we show

$$V(\overline{\mathcal{P}'}) \leq (2+P)V(\mathcal{P}'). \tag{4.2}$$

Note that source s_n is the only scheduled source in the leftmost batch of policy $\mathcal{P}'$, and since its cost-to-success-probability ratio c_n/p_n is maximized, $c_n/p_n = 1 - V$, where V denotes the expected value of the policy $\mathcal{P}'$ excluding s_n. Again, since c_n/p_n is maximized, $V = V(\mathcal{P}')$, i.e., the presence of s_n in $\mathcal{P}'$ does not increase $V(\mathcal{P}')$ (see Lemma 3.2). Hence $c_n/p_n = 1 - V(\mathcal{P}')$. Consider the subpolicy A consisting of the sources in $\mathcal{P}'$ with cost-to-success-probability ratio less than $1 - V(\mathcal{P}')$, and let P_A denote its success probability. Let B denote the subpolicy consisting of the rest of the sources in $\mathcal{P}'$ with success probability P_B. The sources in B have higher cost-to-success-probability ratios than the sources in A; hence they are queried later than the sources in A in policy $\overline{\mathcal{P}'}$. Therefore we have $V(\overline{\mathcal{P}'}) = V(\overline{A}) + (1 - P_A)V(\overline{B})$. Since $\mathcal{P}'$ is irreducible, $V(A) \le V(\mathcal{P}')$, and because subpolicy A is irreducible and does not include s_n, we have $V(\overline{A}) \le (2 + P_A)V(A) \le (2 + P_A)V(\mathcal{P}')$ by induction. It is not hard to see that $V(\overline{B}) \le P_B V(\mathcal{P}')$: Let the sources in B be $s_1, \ldots, s_m$, and let α_i denote the cost-to-success-probability ratio of s_i, where $\alpha_1 \le \cdots \le \alpha_m$. Then

$$\begin{aligned} V(\overline{B}) &= p_1(1-\alpha_1) + (1-p_1)p_2(1-\alpha_2) + \cdots + \left(\prod_{i=1}^{m-1}(1-p_i)\right) p_m(1-\alpha_m) \\ &\le \left(p_1 + (1-p_1)p_2 + \cdots + \left(\prod_{i=1}^{m-1}(1-p_i)\right) p_m\right)(1-\alpha_1) \\ &\le P_B V(\mathcal{P}'). \end{aligned}$$

Therefore

$$\begin{aligned} V(\overline{\mathcal{P}'}) &= V(\overline{A}) + (1-P_A)V(\overline{B}) \\ &\le 2V(\mathcal{P}') + P_A V(\mathcal{P}') + (1-P_A)P_B V(\mathcal{P}') \\ &= 2V(\mathcal{P}') + (P_A + (1-P_A)P_B)V(\mathcal{P}') \\ &= (2+P)V(\mathcal{P}'). \end{aligned}$$

We complete the proof using inequalities (4.1) and (4.2):

$$V(\overline{\mathcal{P}}) \le 2(V(\mathcal{P}) - V(\mathcal{P}')) + V(\overline{\mathcal{P}'})) \le 2V(\mathcal{P}) - 2V(\mathcal{P}') + (2+P)V(\mathcal{P}')$$

or

$$V(\overline{\mathcal{P}}) \le 2V(\mathcal{P}) + PV(\mathcal{P}) = (2+P)V(\mathcal{P}). \qquad \square$$

A similar proof shows that Lemma 4.3 also holds for the general case of an irreducible schedule with sources that can have unequal durations. However the irreducibility criterion is slightly more complicated than what is derived from (3.2) for batched schedules, and unfortunately, unlike for the case for batched schedules, we don't know how to use that fact to derive an approximation algorithm for the general case. The interested reader is referred to [10] for a proof for the general case. Interestingly, serializing a single irreducible batch of sources can at most double the value (this result was used in [1]).

Now we analyze the performance of algorithm Reverse-Greedy. Just as in the case for single-batch policies, we will also use set operations on k-batch policies when there is no ambiguity. Denote the optimal policy as OPT, and partition OPT as

$$\begin{aligned} \mathrm{OPT}_1 &= \mathrm{APPR} \cap \mathrm{OPT}, \\ \mathrm{OPT}_2 &= \mathrm{OPT} - \mathrm{OPT}_1, \end{aligned}$$

where the sources in OPT_1 and OPT_2 are scheduled in the same batches as they are in OPT. By Lemma 3.1,

$$V(\mathrm{OPT}) \le V(\mathrm{OPT}_1) + V(\mathrm{OPT}_2).$$

We compare the performances of OPT_1 and OPT_2 with that of APPR separately.

LEMMA 4.4. *$V(\mathrm{OPT}_1) \le 3V(\mathrm{APPR})$.*

Proof. This follows immediately from Corollary 4.2 and Lemma 4.3. ∎

LEMMA 4.5. *$V(\mathrm{OPT}_2) \le 2V(\mathrm{APPR})$.*

Proof. For each $i = 1, \dots, k$, let $\mathrm{OPT}_2(i)$ and $\mathrm{APPR}(i)$ denote the subpolicies of OPT_2 and APPR consisting of the last $k-i+1$ batches. We prove inductively, starting from the last batch, that $V(\mathrm{OPT}_2(i)) \le 2V(\mathrm{APPR}(i))$. Without loss of generality, we may assume that $V(\mathrm{APPR}(i)) \le V(\mathrm{OPT}_2(i))$ for all i because otherwise we could always replace the last $k-i+1$ batches of OPT_2 with those of APPR and continue with the induction. This could only improve $V(\mathrm{OPT}_2)$.

The base of the induction is clearly true by Theorem 3.8. Note that making each batch irreducible can only increase the expected value of the batch. Suppose that the claim holds for $i+1 \le k$, and consider batch i. Let P_o, C_o be the collective success probability and cost of the sources in the ith batch of OPT_2 and P_a, C_a the corresponding quantities for APPR. By Lemma 3.2,

$$V(\mathrm{APPR}(i)) - V(\mathrm{APPR}(i+1)) = P_a(1 - V(\mathrm{APPR}(i+1))) - C_a,$$

$$V(\mathrm{OPT}_2(i)) - V(\mathrm{OPT}_2(i+1)) = P_o(1 - V(\mathrm{OPT}_2(i+1))) - C_o.$$

Taking the ratio and noting that $V(\mathrm{APPR}(i+1)) \le V(\mathrm{OPT}_2(i+1))$,

$$\begin{aligned}\frac{P_a(1 - V(\mathrm{APPR}(i+1))) - C_a}{P_o(1 - V(\mathrm{OPT}_2(i+1))) - C_o} &= \frac{1 - V(\mathrm{APPR}(i+1))}{1 - V(\mathrm{OPT}_2(i+1))}\left[\frac{P_a - \frac{C_a}{1-V(\mathrm{APPR}(i+1))}}{P_o - \frac{C_o}{1-V(\mathrm{OPT}_2(i+1))}}\right]\\ &\ge \frac{P_a - C_a/(1 - V(\mathrm{APPR}(i+1)))}{P_o - C_o/(1 - V(\mathrm{OPT}_2(i+1)))}\\ &\ge \frac{P_a - C_a/(1 - V(\mathrm{APPR}(i+1)))}{P_o - C_o/(1 - V(\mathrm{APPR}(i+1)))}.\end{aligned}$$

Let set S consist of all sources that do not appear in $\mathrm{APPR}(i+1)$. Clearly, S includes all sources in OPT_2 and thus all sources in the ith batch of OPT_2. Divide the cost of each source by $1 - V(\mathrm{APPR}(i+1))$. Then on input S, Pick-a-Star would return exactly the same set as the ith batch of APPR with expected value being $P_a - C_a/(1 - V(\mathrm{APPR}(i+1)))$. By Theorem 3.8, the value is at least half of the optimal expected value for set S which is in turn at least $P_o - C_o/(1-V(\mathrm{APPR}(i+1)))$. This means

$$2(V(\mathrm{APPR}(i)) - V(\mathrm{APPR}(i+1))) \ge V(\mathrm{OPT}_2(i)) - V(\mathrm{OPT}_2(i+1)),$$

and hence $2V(\mathrm{APPR}(i)) \ge V(\mathrm{OPT}_2(i))$. ∎

Lemmas 4.4 and 4.5 together give the following theorem.

THEOREM 4.6. *Algorithm Reverse-Greedy returns a k-batch policy with an expected value at least $1/5$ of the optimum.*

5. Approximation algorithms for the cost threshold models. We first present an FPTAS for model TL under a weak assumption: $p_i - d_i \geq 0$ for every source s_i (d_i is the time duration of s_i), i.e., every source considered is profitable by itself. The extension to model TT (with no restriction) is straightforward. Note that in model TT our goal is simply to maximize the overall probability of getting the information under the time and cost constraints.

The main idea is the rounding technique introduced in [6] for Knapsack. As mentioned before, in the cost threshold model TL, an optimal policy should be in fact a single-batch policy. Let $\mathcal{P} = \{s_{i_1}, \ldots, s_{i_m}\}$ be a single-batch policy, where $d_{i_1} \leq \cdots \leq d_{i_m}$. Then

$$V(\mathcal{P}) = 1 - \prod_{j=1}^{m}(1 - p_{i_j}) - \sum_{j=1}^{m}\prod_{l=1}^{j-1}(1 - p_{i_l})p_{i_j}d_{i_j} - \prod_{j=1}^{m}(1 - p_{i_j})d_{i_m}.$$

We cannot apply the rounding technique to the above objective function directly because it involves subtractions. We rewrite the expression as

$$\begin{aligned} V(\mathcal{P}) &= \sum_{j=1}^{m}\prod_{l=1}^{j-1}(1 - p_{i_l})p_{i_j}(1 - d_{i_j}) - \prod_{j=1}^{m}(1 - p_{i_j})d_{i_m} \\ &= \sum_{j=1}^{m-1}\prod_{l=1}^{j-1}(1 - p_{i_l})p_{i_j}(1 - d_{i_j}) + \prod_{j=1}^{m-1}(1 - p_{i_j})(p_{i_m} - d_{i_m}). \end{aligned} \tag{5.1}$$

Since $p_i - d_i \geq 0$ by our assumption, every term is nonnegative in (5.1), and we can apply rounding as follows.

Let $\epsilon > 0$ be any desired relative error. Sort the sources in the ascending order of their time durations. We will exhaustively consider every possible choice of s_{i_m}. For each $i \leq n$, consider only policies that includes the source s_i and possibly some others from $\{s_1, \ldots, s_{i-1}\}$, subject to the same cost threshold. Let $\mathrm{OPT}(i)$ denote an optimal such policy. For simplicity, assume that $|\mathrm{OPT}(i)| > 1$. We find a trivial lower bound for $V(\mathrm{OPT}(i))$:

$$V(\mathrm{OPT}(i)) \geq L_i = \max\left\{p_i - d_i, \quad \max_{\substack{j<i \\ c_j + c_i \leq \varsigma}} p_j(1 - d_j)\right\}.$$

Similar to [6], we formulate a new instance by rounding $p_i - d_i$ and each $p_j(1 - d_j)$ *down* to the nearest multiple of $\epsilon L_i/(2i)$. But here we also need round each $1 - p_j$ to the nearest power of $(1 - \epsilon/(2i))^{1/(i-1)}$. In other words, we round each $\log(1 - p_j)$ to the nearest multiple of $(\log(1 - \epsilon/(2i)))/(i - 1)$. We solve the new instance optimally. Let OPT_i denote an optimal policy for the new instance. It is sufficient to bound the difference between $V(\mathrm{OPT}(i))$ and $V(\mathrm{OPT}_i)$ and we do this by obtaining an upper bound between each term of $OPT(i)$ and the corresponding rounded value. Each term difference is upper bounded when the $p_i - d_i$ or $p_i(1 - d_i)$ part of a term is at its maximum L_i and the probability factors in the unrounded term are all 1 (otherwise they are factored out and reduce the difference). Thus we obtain a maximum difference of $L_i - L_i(1 - \frac{\epsilon}{2i})[(1 - \frac{\epsilon}{2i})^{\frac{1}{i-1}}]^{i-1}$ for every term and there can be at most i many:

$$V(\mathrm{OPT}(i)) - V(\mathrm{OPT}_i) \leq i\left(L_i - L_i\left(1 - \frac{\epsilon}{2i}\right)\left[\left(1 - \frac{\epsilon}{2i}\right)^{\frac{1}{i-1}}\right]^{i-1}\right)$$

$$\begin{aligned}&= iL_i\left(1-\left(1-\frac{\epsilon}{2i}\right)^2\right)\\&\le iL_i\left(1-\left(1-\frac{\epsilon}{i}\right)\right)\\&=\epsilon L_i\\&\le \epsilon V(\mathrm{OPT}(i)).\end{aligned}$$

Hence the new instance approximates the original problem with the desired ratio.

We can compute OPT_i for the new instance by dynamic programming in the space $\mathcal{S}_i$ of all possible values of $V(\mathrm{OPT}_i)$. Denote $Q_i = \prod_{j<i}(1-p_j)$. By the above rounding, the cardinality of $\mathcal{S}_i$ is upper bounded by

$$i\left(\frac{2i}{\epsilon}\right)\left(\frac{(i-1)\log Q_i}{\log(1-\epsilon/(2i))}\right) < i\left(\frac{2i}{\epsilon}\right)\left(\frac{2i^2\log Q_i}{-\epsilon}\right) = \frac{-4i^4\log Q_i}{\epsilon^2},$$

which is polynomial in the input size and $1/\epsilon$. In the above inequality, the factor $\frac{2i}{\epsilon}$ represents the number of different values that $p_i - d_i$ and $p_j(1-d_j)$ can have after being rounded to the nearest multiple of $\epsilon L_i/(2i)$, and the factor $\frac{(i-1)\log Q_i}{\log(1-\epsilon/(2i))}$ represents the number of different exponents that a product of the form $\prod_{l=1}^{j-1}(1-p_{i_l})$ can have after being rounded to the nearest power of $(1-\epsilon/(2i))^{1/(i-1)}$.

Rewrite the expression in equality (5.1) as a nested form:

$$p_{i_1}(1-d_{i_1}) + (1-p_{i_1})[p_{i_2}(1-d_{i_2}) + (1-p_{i_2})[\cdots + (1-p_{i_{m-1}})(p_{i_m} - d_{i_m})]].$$

The form easily suggests a backward inductive algorithm. The algorithm will cycle through the list $s_{i-1},\ldots,s_1$. For each $j = i,\ldots,1$ and each possible value $x \in \mathcal{S}_i$, it computes and records a policy of expected value x for the subset of sources $\{s_j,\ldots,s_i\}$ that contains the source s_i and costs the least. The above nested form allows the algorithm to find the cheapest policy of a specific expected value for subset $\{s_j,\ldots,s_i\}$ by expanding the cheapest policies of the same or lower expected values recorded before for subset $\{s_{j+1},\ldots,s_i\}$ to potentially include the source s_j. The running time is at most $O(i^2|\mathcal{S}_i|)$, which is polynomial in the input size and $1/\epsilon$.

THEOREM 5.1. *Assume that $p_i - d_i \ge 0$ for every source s_i. There is an FPTAS for the problem of computing optimal policies in model* TL.

COROLLARY 5.2. *There is an FPTAS for the problem of computing optimal policies in model* TT.

Proof. Recall that the objective function in this case is

$$V(\mathcal{P}) = 1 - \prod_{j=1}^{m}(1-p_{i_j}),$$

which involves only the success probabilities of the sources queried. Hence the above FPTAS works if we simply throw out all sources whose time duration exceeds the deadline. ◻

6. Concluding remarks. As charging for information on the Internet becomes more common, information-acquisition algorithms will have to weigh the benefits of acquiring information against the cost of doing so. Characteristics of these problems are (1) the fact that the information provided by a source cannot be fully predicted in all cases, so the benefit of asking for information can be uncertain, (2) the fact that there can be monetary and time costs associated with information requests, and (3) the fact that information providers can be accessed both serially and in parallel.

We have developed a model that takes into account these aspects of information scheduling and have established worst-case complexity and approximation results for a variety of objective functions: those in which the value is linear in the cost or time attributes and the consumer supplies a cost and/or time threshold for acquiring information. All of these models have plausible applications for information access on the World Wide Web.

Appendix A. Verifying the claims. We prove the claims made on the behavior of the various functions that came up in the main proofs.

Proof of Claim 3.5. We show that the function $P_1 - l(1-(1-P_1)^{1/l})\alpha_1$ is decreasing in P_1, where $1 - P_1 \geq \alpha_1$.

We take the derivative with respect to P_1 and get

$$1 - \alpha_1(1-P_1)^{\frac{1}{l}-1} = 1 - \frac{\alpha_1}{(1-P_1)^{\frac{l-1}{l}}}.$$

The ratio is increasing in P_1 as long as $(1-P_1)^{\frac{l-1}{l}} > \alpha_1$.

Proof of Claim 3.6. We show that the function $h(l, p_1) = l(1-(1-p_1)^{1/l})$ is nondecreasing in l.

Taking the derivative, we have

$$\frac{\partial h(l,p_1)}{\partial l} = 1 - (1-p_1)^{1/l} + \frac{\ln(1-p_1)(1-p_1)^{1/l}}{l}.$$

We observe that $\frac{\partial h(l,0)}{\partial l} = 1 - 1 + 0 = 0$ and

$$\begin{aligned}\frac{\partial h(l,p_1)}{\partial p_1 \partial l} &= 1/l(1-p_1)^{1/l-1} - \frac{(1-p_1)^{1/l}}{l(1-p_1)} - \frac{(1-p_1)^{1/l-1}\ln(1-p_1)}{l^2} \\ &= -\frac{(1-p_1)^{1/l-1}\ln(1-p_1)}{l^2} \geq 0.\end{aligned}$$

Therefore $\frac{\partial h(l,p_1)}{\partial l}$ is nonnegative for all $p_1 \in [0, 1.0]$. Whence $h(l,p_1)$ is nondecreasing in l.

Proof of Claim 3.7. We need show that the ratio $(\frac{1}{1-x^{m/(m-1)}})(\frac{1-x+x\ln x}{1-x})$ is decreasing in $x \in (0,1)$.

We take the derivative

$$\begin{aligned}&\left(\frac{\frac{m}{m-1}x^{\frac{1}{m-1}}}{(1-x^{\frac{m}{m-1}})^2}\right)\left(\frac{1-x+x\ln x}{1-x}\right) + \left(\frac{1}{1-x^{m/(m-1)}}\right)\left(\frac{(1-x)\ln x + 1 - x + x\ln x}{(1-x)^2}\right) \\ &\quad = \frac{1}{(1-x^{m/(m-1)})(1-x)}\left[\left(\frac{\frac{m}{m-1}x^{\frac{1}{m-1}}}{1-x^{\frac{m}{m-1}}}\right)(1-x+x\ln x) + \left(\frac{1-x+\ln x}{1-x}\right)\right].\end{aligned}$$

The factor $\frac{1}{(1-x^{m/(m-1)})(1-x)}$ is positive in the interval; hence it suffices to show that the term in the brackets is negative. Noting that $1 - x + x\ln x$ in the left summand is nonnegative (the derivative $\ln x$ is negative, and the function is zero at 1), and that in the right summand $1 - x + \ln x \leq 0$, we conclude that the left summand is nonnegative, while the right one is nonpositive. We will verify that

$$\frac{1}{1-x} \geq \frac{\frac{m}{m-1}x^{\frac{1}{m-1}}}{1-x^{\frac{m}{m-1}}}. \tag{A.1}$$

This eliminates the extra factors and makes it sufficient to show that $h_1(x) = (1 - x + x\ln x) + 1 - x + \ln x < 0$. We note that $h_1(x)$ is negative at x arbitrarily close to zero, and zero at $x = 1$, and its derivative $-1 + \ln x + 1/x = \frac{1-x+x\ln x}{x}$ is nonnegative in the interval.

Rearranging (A.1), we need to verify that

$$\frac{\frac{m}{m-1}x^{\frac{1}{m-1}}(1-x)}{1-x^{\frac{m}{m-1}}} \le 1.$$

The fraction is zero at $x = 0$ and, taking the limit as x approaches 1, we have

$$\lim_{x\to 1} \frac{\frac{m}{m-1}x^{\frac{1}{m-1}}(1-x)}{1-x^{\frac{m}{m-1}}} = \frac{\frac{m}{m-1}(\frac{1}{m-1} - \frac{m}{m-1})}{-\frac{m}{m-1}} = 1.$$

Finally, we can verify that the derivative is nonnegative in the region. We may ignore the factor $\frac{m}{m-1}$, and to simplify a little, we make the substitution $u = x^{\frac{1}{m-1}}$. Hence we want to show that the derivative of $h_2(u) = \frac{u(1-u^{m-1})}{1-u^m}$ is positive. Taking the derivative

$$h_2'(u) = \frac{(1 - mu^{m-1})(1-u^m) + mu^m(1-u^{m-1})}{(1-u^m)^2},$$

we will show that the numerator

$$1 - mu^{m-1} - u^m + mu^{2m-1} + mu^m - mu^{2m-1} = u^{m-1}((m-1)u - m) + 1$$

is positive. The function $h_3(u) = u^{m-1}((m-1)u - m) + 1$ is nonnegative on the interval since $h_3(0) = 1$ and $h_3(1) = 0$ and $h_3'(u) < 0$ on the interval:

$$\begin{aligned} h_3'(u) &= (m-1)u^{m-2}((m-1)u - m) + (m-1)u^{m-1} \\ &= (m-1)u^{m-2}((m-1)u - m + u) = (m-1)u^{m-2}(m(u-1)) < 0. \end{aligned}$$

Hence $h_2'(u) \ge 0$.

Proof of Claim 3.15. We need to show that the function

$$\begin{aligned} f_2(\alpha_1) - \frac{r-1}{2}f_1(\alpha_1) = {}& \frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1 \ln\left(\frac{\alpha_1}{(1-p)^r}\right) \\ &+ (1-p)^r - \alpha_1 - \frac{r-1}{2}\left(\alpha_1 p^2 - \alpha_1 p \ln\left(\frac{\alpha_1}{(1-p)^r}\right)\right) \end{aligned}$$

is decreasing in α_1 for $\alpha_1 \le (1-p)^{2r-1}$. Taking the partial derivative on $f_2(\alpha_1) - \frac{r-1}{2}f_1(\alpha_1)$ with respect to α_1,

$$\begin{aligned} &\frac{\partial\left[\frac{r(r-1)}{2}(1-p)^{r-2}p^2 + \alpha_1\ln\left(\frac{\alpha_1}{(1-p)^r}\right) + (1-p)^r - \alpha_1 - \frac{r-1}{2}\left(\alpha_1 p^2 - \alpha_1 p \ln\left(\frac{\alpha_1}{(1-p)^r}\right)\right)\right]}{\partial \alpha_1} \\ &= \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + 1 - 1 - \frac{r-1}{2}p^2 + \frac{(r-1)p}{2}\ln\left(\frac{\alpha_1}{(1-p)^r}\right) + \frac{(r-1)p}{2} \\ &< \ln\left(\frac{\alpha_1}{(1-p)^r}\right) + \frac{(r-1)p}{2} \end{aligned}$$

$$\begin{aligned}
&\leq (r-1)\ln(1-p) + \frac{(r-1)p}{2}\\
&< -(r-1)p + \frac{(r-1)p}{2}\\
&\leq 0.
\end{aligned}$$

Acknowledgments. Many thanks to Richard Anderson and Richard Karp for the discussions and their very valuable suggestions, and to the two anonymous referees for their careful reading and constructive criticism of the paper.

REFERENCES

[1] O. Etzioni, S. Hanks, T. Jiang, R. M. Karp, O. Madani, and O. Waarts, *Efficient information gathering on the Internet*, in Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 234–243.

[2] O. Etzioni, R. M. Karp, and O. Waarts, *Efficient Access to Information Sources on the Internet*, manuscript, 1996.

[3] P. Feigin and G. Harel, *Minimizing costs of personnel testing programs*, Naval Research Logistics Quarterly, 29 (1982), pp. 87–95.

[4] M. Garey, *Optimal task scheduling with precedence constraints*, Discrete Math., 4 (1973), pp. 37–56.

[5] M. Henig and D. Simchi-Levy, *Scheduling tasks with failure probabilities to minimize expected cost*, Naval Research Logistics, 37 (1990), pp. 99–109.

[6] O. Ibarra and C. Kim, *Fast approximation algorithms for the knapsack and sum of subsets problems*, J. ACM, 22 (1975), pp. 463–468.

[7] The LEXIS-NEXIS Corporation, 1997; available online at http://www.lexis-nexis.com/lncc/.

[8] J. Kadane, *Quiz show problems*, Math. Anal. Appl., 27 (1969), pp. 609–623.

[9] B. Krulwich, *The BargainFinder agent: Comparison price shopping on the Internet*, in Bots and Other Internet Beasties, SAMS.NET, Indianapolis, IN, 1996.

[10] O. Madani, *Serializing an Irreducible Schedule at most Triples the Value*, manuscript, 1996.

[11] The Knowledge Finder Corporation, providing access to MEDLINE, 1997; available online at http://www.kfinder.com.

[12] L. Mitten, *An analytic solution to the least cost testing sequence problem*, J. Indust. Eng., 11 (1960), p. 17.

[13] M. Porat, *The Information Economy*, U.S. Office of Telecommunications, Dept. of Commerce, Washington, D.C., 1977.

[14] S. Sahni, *Approximation algorithms for the* 0/1-*knapsack problem*, J. ACM, 22 (1975), pp. 115–124.

[15] *Secure Transaction Technology*, available online at http://www.visa.com/cgi-bin/vee/sf/set/intro.html.

[16] E. Selberg and O. Etzioni, *Multi-service search and comparison using the MetaCrawler*, in Proceedings of the 4th World Wide Web Conference, Boston, MA, Elsevier, New York, 1995, pp. 195–208.

[17] H. Simon and J. Kadane, *Optimal problem-solving search: All-or-none solutions*, Artificial Intelligence, 6 (1975), pp. 235–247.

[18] M. Sirbu and J. D. Tygar, *NetBill: An internet commerce system optimized for network delivered services*, IEEE Personal Communications, 2 (1995), pp. 34–39.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1621–1629

VIRTUAL PATH LAYOUTS IN ATM NETWORKS*

LADISLAV STACHO† AND IMRICH VRŤO†

Abstract. We study virtual path layouts in a very popular type of fast interconnection networks, namely asynchronous transfer mode (ATM) networks. One of the main problems in such networks is to construct path layouts that minimize the hop-number (i.e., the number of virtual paths between any two nodes) as a function of the edge congestion c (i.e., the number of virtual paths going through a link). In this paper we construct for any n vertex network H and any c a virtual path layout with hop-number $O(\frac{diam(H)\log\Delta}{\log c})$, where $diam(H)$ is the diameter of the network H and Δ is its maximum degree. Involving a general lower bound from [E. Kranakis, D. Krizanc, and A. Pelc, *Seventh IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 1995, pp. 662–668], we see that these hop-numbers are optimal for bounded degree networks with the diameter $O(\log n)$ for any congestion c. In the case of unbounded degree networks (with the diameter $O(\log n)$) these hop-numbers are optimal for any $c \geq \Delta$. For instance, this gives optimal hop-numbers for hypercube related networks. Moreover, we improve known results for paths and meshes and prove optimal hop-numbers for hypercubes.

Key words. ATM network, congestion, hop-number, virtual paths layout

AMS subject classifications. 68M10, 90B12

PII. S0097539796308151

1. Introduction.

Broadband integrated services digital network (B-ISDN) is a new paradigm in digital communication which integrates previous distinct networks (telephone, cable television, computer) into a single digital network. The basic network transmission medium is a fiber-optic cable capable of transferring data at very high rates. The bottleneck caused by slow software-based switches is resolved by special purpose fast hardware. To utilize this, a new multiplexing and switching technology called ATM (asynchronous transfer mode) was proposed (see, e.g., [9]). Packet routing in ATM networks is based on relatively small fixed-size packets. Messages may be transmitted through arbitrarily long virtual paths. Packets are routed along these paths by maintaining a routing field whose subfields determine intermediate destinations of the packet, i.e., end-points of virtual paths on their way to the final destination. One of the main problems in such networks is to construct path layout that minimizes the hop-number (i.e., the number of virtual paths between any two nodes) as a function of the edge congestion c (i.e., the number of virtual paths going through a link).

In this paper we construct for any n vertex network H and any c a virtual path layout with the hop-number $O(\frac{diam(H)\log\Delta}{\log c})$, where $diam(H)$ is the diameter of the network H and Δ is its maximum degree. Involving a general lower bound from [6], we see that these hop-numbers are optimal for bounded degree networks with the diameter $O(\log n)$ for any congestion c. In the case of unbounded degree networks (with the diameter $O(\log n)$) these hop-numbers are optimal for any $c \geq \Delta$. For instance, this gives optimal hop-numbers for hypercube related networks. Moreover, we improve known results for paths and meshes of Kranakis, Krizanc, and Pelc [6].

*Received by the editors August 14, 1996; accepted for publication (in revised form) August 10, 1999; published electronically March 17, 2000. The research of both authors was supported by the Slovak Grant Agency under grant 2/7007/20.

http://www.siam.org/journals/sicomp/29-5/30815.html

†Institute of Mathematics, Slovak Academy of Sciences, Department of Informatics, P.O. Box 56, 840 00 Bratislava 4, Slovakia (stacho@savba.sk, vrto@savba.sk).

Finally, we prove optimal hop-numbers for hypercubes.

1.1. Model, notation, and results. The basic model of ATM networks was introduced by Gerstel et al. in [1, 2, 4, 5] and further developed in [3, 6]. By a network we understand any graph $G = (V_G, E_G)$. By $diam(G)$ we denote the *diameter* of G. Similarly, by $d(x, y)$ we denote the distance of the vertices x and y in G. Let Δ denote the maximum degree of a graph. Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs such that $|V_G| \leq |V_H|$. An *embedding* of G in H is a pair of mappings (ϕ, ψ) satisfying

$$\phi : V_G \to V_H \quad \text{is an injection}, \quad \psi : E_G \to \{\text{ set of all paths in } H\ \},$$

such that if $uv \in E_G$, then $\psi(uv)$ is a path between $\phi(u)$ and $\phi(v)$. Define the *congestion of an edge* $e \in E_H$ as $cg_{(\phi,\psi)}(e) = |\{f \in E_G \ : \ e \in \psi(f)\}|$ and the *congestion* of G in H as $cg(G, H) = \min_{(\phi,\psi)} \max_{e\in E_H} \{cg_{(\phi,\psi)}(e)\}$. Note that if there is no confusion we will omit the subscript (ϕ, ψ) in $cg_{(\phi,\psi)}(e)$.

The paths $\{\psi(E_G)\}$ are called *virtual paths*. The virtual path problem can be mathematically defined as follows. Given a graph H and a positive number c, among all graphs G satisfying $|V_G| = |V_H|$ and $cg(G, H) \leq c$, find a graph G_0 of minimum diameter. The minimum diameter is called the *hop-number* and is denoted by $Hop_H(c)$.

Gerstel and Zaks [5] studied virtual path layouts for paths, cycles, and meshes. They assumed an additional requirement that virtual channels are the shortest paths in the networks. Kranakis, Krizanc, and Pelc [6] dropped the requirement and proved several optimal results on the hop-number of paths and meshes for small c. Namely, if P_n denotes an n-vertex path, then $Hop_{P_n}(2) = \sqrt{2n} + o(1)$, $Hop_{P_n}(c) = \Theta(n^{1/c})$, when c is a constant and $Hop_{P_n}(\log^2 n/\log\log n) = \Theta(\log n/\log\log n)$, and if M_n denotes an $n \times n$ mesh, then $Hop_{M_n}(c) = \Theta(\log n)$ for constant c.

We assume the same model as Kranakis, Krizanc, and Pelc [6]. Our main result says that for any $c \geq 1$ and any n vertex graph H of bounded degree and diameter $O(\log n)$, $Hop_H(c) = \Theta(\log n/\log c)$. Several standard networks belong to this class of graphs, e.g., mesh of trees, butterfly, cube-connected-cycles, binary de Bruijn, or shuffle-exchange graph. If $c \geq \Delta$, then we construct optimal virtual path layouts for some unbounded degree networks, e.g., star, pancake, and k-ary de Bruijn and Kautz graphs.

Further, we improve the results of Kranakis, Krizanc, and Pelc [6] in the following way:

(i) $Hop_{P_n}(c) = \Theta(\frac{\log n}{\log c})$, if $c \geq \log^{1+\varepsilon} n$, for any fixed $\varepsilon > 0$.

(ii) If H denotes two-dimensional $n\times n$ mesh, then $Hop_H(c) = \Theta(\frac{\log n}{\log c})$for $c \geq 2$.

Finally, we show that for the n-dimensional hypercube Q_n,

$$Hop_{Q_n}(c) = \begin{cases} \Theta\left(\frac{n}{\log n}\right) & \text{if } 2 \leq c \leq n, \\ \Theta\left(\frac{n}{\log c}\right) & \text{otherwise.} \end{cases}$$

2. General bounds. In this section we describe a general method for constructing virtual path layouts and then apply it to some standard networks. Since we deal only with asymptotic results, we can restrict ourselves to looking for the graph G_0 among trees.

PROPOSITION 2.1. *Let a graph G_0 minimize $diam(G)$ with respect to $cg(G, H) \leq c$. Then there exists a tree T with $diam(T) \leq 2diam(G_0)$ and $cg(T, H) \leq c$.*

Indeed, we can take T as a breadth first search spanning tree of H_0. We will often apply the following useful lemma from [10].

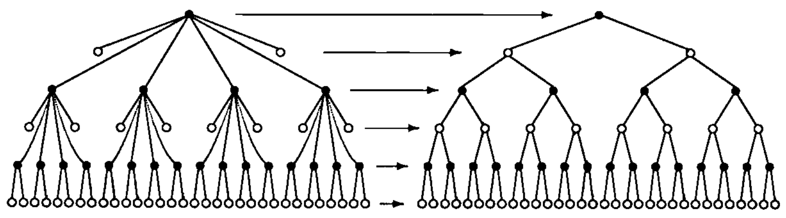


FIG. 1. *The embedding of $F(4;2,6)$ in $R(2,6)$.*

LEMMA 2.2. *Let G_1, G_2, and G_3 be graphs with $|V_{G_1}| \leq |V_{G_2}| \leq |V_{G_3}|$. Consider an embedding of G_1 into G_2 and an embedding of G_2 into G_3. Then $cg(G_1, G_3) \leq cg(G_1, G_2)cg(G_2, G_3)$.*

It is worth noting that an embedding of G_1 into G_2 followed by the embedding of G_2 into G_3 induces an embedding of G_1 into G_3. An image of an edge under this composed embedding is not necessarily a path but a walk in general. It is easy to transform the walk into a path by omitting multiple edges and vertices.

Throughout, we will use a general lower bound of Kranakis, Krizanc, and Pelc [6].

LEMMA 2.3. *For any n vertex graph H of maximum degree Δ and for any $c \geq 1$,*

$$Hop_H(c) \geq \frac{\log n}{\log(c\Delta)} - 1.$$

By $R(k, n)$ we denote the complete k-ary n-level tree. Note that by a k-ary 1-level and 0-ary k-level tree we mean a single vertex. We sometimes abbreviate this notation in the case of $R(2, n)$ to B_n (the complete binary tree) and $R(l, 2)$ to S_l (the star on $l + 1$ vertices).

Let T be a rooted tree. Let G be a rooted tree with $l \geq k$ leaves, where l and k are positive integers. By $G * kT$ we denote the graph constructed as follows: take k copies of T and identify the roots of the copies of T with k distinct leaves of G.

For $d, i, n \geq 1$, and $k = d^i$, define the tree $F(k; d, n)$ recursively as follows. If $i = 1$, set $F(k; d, n) = R(d, n)$. If $i \geq n$, set $F(k; d, n) = R(\frac{d^n-1}{d-1} - 1, 2)$. If $1 < i < n$, set $F(k; d, n) = R(\frac{d^{i+1}-1}{d-1} - 1, 2) * kF(k; d, n - i)$. Note that the number of vertices of $F(k; d, n)$ is $\frac{d^n-1}{d-1}$. A picture of $F(4; 2, 6)$ is depicted in Figure 1 (the left tree).

LEMMA 2.4. *For $d, i, n \geq 1$, and $k = d^i$, $diam(F(k; d, n)) = 2(a + \lceil\frac{r}{i}\rceil) = 2\lceil\frac{n-1}{i}\rceil$, where $n - 1 = ai + r$, $0 \leq r < i$.*

Proof. For $n = 1, 2$, the lemma is easy to observe. We assume that the lemma is true for all $l < n$ and we prove it for $l = n$. For $i = 1$, $diam(R(d, n)) = 2(n - 1)$, the lemma is true. Similarly, for $i \geq n$, $diam(R(\frac{d^n-1}{d-1} - 1, 2)) = 2$. Now, let $1 < i < n$. Hence $F(k; d, n) = R(\frac{d^{i+1}-1}{d-1} - 1, 2) * kF(k; d, n-i)$. Let $n-1 = ai+r$, $0 \leq r < i$. Since $n-i \geq 1$ and since $n-i = (a-1)i+r$, by induction, $diam(F(k; d, n-i)) = 2(a-1+\lceil\frac{r}{i}\rceil)$, and $diam(R(\frac{d^{i+1}-1}{d-1} - 1, 2)) = 2$. It follows from the construction of $F(k; d, n)$ that $diam(F(k; d, n)) = diam(F(k; d, n - i)) + 2 = 2(a + \lceil\frac{r}{i}\rceil) = 2\lceil\frac{n-1}{i}\rceil$. ◻

LEMMA 2.5. *For $d, i, n \geq 1$, and $k = d^i$, $cg(F(k; d, n), R(d, n)) \leq \frac{k-1}{d-1}$.*

Proof. We proceed by induction on n. For $n = 1, 2$, the result is easy to see. Assuming the result is true for $l \leq n - 1$, we prove it for $l = n$. For $i = 1$, the statement of the lemma is true (we take as the mapping (ϕ, ψ) the natural isomorphism $\phi : F(k; d, n) \to R(d, n)$, and ψ the induced mapping by ϕ). If $i \geq n$, let ϕ map the root of $R(\frac{d^n-1}{d-1} - 1, 2)$ on the root of $R(d, n)$ and other vertices, injectively in an arbitrary way, on the remaining vertices of $R(d,n)$. Further, let ψ map each edge e of $R(\frac{d^n-1}{d-1}-1,2)$ on the unique paths in $R(d,n)$ connecting the images of the end-vertices of e. It is a matter of routine to observe that the congestion of each edge of $R(d,n)$ induced by (ϕ,ψ) is at most $\frac{k-1}{d-1}$ (the maximum congestion is achieved on edges incident with the root of $R(d,n)$), thus $cg(R(\frac{d^n-1}{d-1}-1,2),R(d,n)) \le \frac{k-1}{d-1}$. Now, let $1<i<n$. By definition $F(k;d,n)=R(\frac{d^{i+1}-1}{d-1}-1,2)*kF(k;d,n-i)$. It holds that $R(d,n)=R(d,i+1)*kR(d,n-i)$. Using induction we can embed $kF(k;d,n-i)$ into $kR(d,n-i)$ (each $F(k;d,n-i)$ into some $R(d,n-i)$) with congestion $\le \frac{k-1}{d-1}$ and thus $cg(kF(k;d,n-i),kR(d,n-i)) \le \frac{k-1}{d-1}$. Similarly, we obtain $cg(R(\frac{d^{i+1}-1}{d-1}-1,2),R(d,i+1)) \le \frac{k-1}{d-1}$. Obviously, these two embeddings together induce an embedding (ϕ,ψ) of $F(k;d,n)$ into $R(d,n)$ with congestion at most $\frac{k-1}{d-1}$. ∎

Remark 2.1. The embedding (ϕ,ψ), constructed in the previous proof, has the property that for each vertex $v \in V_{F(k;d,n)}$ there is at most one neighbor $u \in V_{F(k;d,n)}$ such that the unique $\phi(v)-\phi(u)$ path in $R(d,n)$ contains a vertex x with $d(\phi(x),r) < d(\phi(v),r)$, where r is the root of $R(d,n)$. This can be observed in Figure 1, where the embedding of $F(4;2,6)$ in $R(2,6)$ is depicted.

LEMMA 2.6. *Let $\Delta \ge 3$, $i \ge 1$, and let T be a tree with maximum degree at most Δ. Then there exists a tree T' with $|V_T|=|V_{T'}|$, $diam(T') \le 2\lceil\frac{diam(T)}{i}\rceil$, and $cg(T',T) \le \frac{(\Delta-1)^i-1}{\Delta-2}$.*

Proof. Let v be a vertex of degree $\le \Delta-1$ in T (since T is a tree, such a vertex always exists). For each $x \in V_T$, $d(x,v) \le diam(T)$. It is easy to observe that T can be extended to the complete $(\Delta-1)$-ary tree $L=R(\Delta-1,diam(T)+1)$ by successively adding vertices of degree one. Let S be the set of all added vertices to T. Let us consider the graph $G=F((\Delta-1)^i;\Delta-1,diam(T)+1)$. By Lemma 2.4, $diam(G) \le 2\lceil\frac{diam(T)}{i}\rceil$. Moreover, by Lemma 2.5, there is an embedding (ϕ,ψ) of G in L with $cg_{(\phi,\psi)}(G,L) \le \frac{(\Delta-1)^i-1}{\Delta-2}$. Let T' be defined as $T'=G-\{x : \phi(x)\in S\}$. In what follows we show that T' is indeed the required tree.

Let ϕ' be the restricted mapping ϕ to the graph T'. Obviously, ϕ' is an injection. Since ϕ' is a surjection as well, $|V_{T'}|=|V_T|$. Since ψ maps each edge xy to the unique $\phi(x)-\phi(y)$ path in L and since L and T are trees, the restricted mapping ψ to the graph T', say ψ', is well defined.

Hence (ϕ',ψ') is an embedding of T' in T with $cg_{(\phi',\psi')}(T',T) \le cg_{(\phi,\psi)}(G,L) \le \frac{(\Delta-1)^i-1}{\Delta-2}$. Since G is a tree, to finish the proof it is sufficient to prove that T' is connected, since then $diam(T') \le diam(G) \le 2\lceil\frac{diam(T)}{i}\rceil$. Let x be the last vertex added to T in the construction of L. Since x is a leaf in L and by Remark 2.1, it follows that $\phi^{-1}(x)$ is a leaf in G as well. Thus after removing x and $\phi^{-1}(x)$, the resulting graphs will be connected. We can continue removing vertices from L and its preimages in G (in the opposite way they were added to T) to obtain T and T', respectively. Since in each step the deleted vertex is a leaf of a subgraph of L, by Remark 2.1, its preimage is also a leaf of a subgraph of G, and thus T' is connected. ∎

THEOREM 2.7. *For any graph H of maximum degree $\Delta \ge 3$ and any given $c \ge 1$,*

$$Hop_H(c) = O\left(\frac{diam(H)\log\Delta}{\log c}\right).$$

Proof. Let T be a breadth first search spanning tree of G. Then T has the maximum degree at most $\Delta \ge 3$ and $diam(T) \le 2diam(H)$. Find i such that $\frac{(\Delta-1)^i-1}{\Delta-2} \le$

$c < \frac{(\Delta-1)^{i+1}-1}{\Delta-2}$. Now, by Lemma 2.6, there is a tree T' with $cg(T',T) \leq \frac{(\Delta-1)^{i}-1}{\Delta-2}$. Clearly $cg(T',H) \leq \frac{(\Delta-1)^{i}-1}{\Delta-2} \leq c$. Similarly, by Lemma 2.6,

$$\begin{aligned} Hop_H(c) \leq diam(T') \leq 2\left\lceil \frac{diam(T)}{i} \right\rceil \leq 2\left\lceil \frac{2diam(H)\log(\Delta-1)}{\log((\Delta-2)c+1)-\log(\Delta-1)} \right\rceil \\ = O\left(\frac{diam(H)\log\Delta}{\log c}\right). \quad \square \end{aligned}$$

Theorem 2.7 together with Lemma 2.3 has two important consequences.

Corollary 2.8. *Let H be a graph of order n with $\Delta = O(1)$ and $diam(H) \leq O(\log n)$. Then $Hop_H(c) = \Theta(\frac{\log n}{\log c})$ for any c.*

Note that several standard networks belong to this class of graphs, e.g., mesh of trees, butterfly, cube-connected-cycles, and binary de Bruijn, or shuffle-exchange graph.

Corollary 2.9. *Let H be a graph of order n with maximum degree $\Delta \geq 3$ satisfying $diam(H)\log\Delta = O(\log n)$. Then $Hop_H(c) = \Theta(\frac{\log n}{\log c})$ for $c \geq \Delta$.*

If $c \geq \Delta$, then the above corollary gives optimal virtual path layouts for, e.g., star, pancake, and k-ary de Bruijn and Kautz graphs.

3. Paths and meshes.

Let P_n denote an n vertex path and $M_n = P_n \times P_n$ denote an $n \times n$ mesh. The next theorem improves virtual path layouts for paths and meshes of Kranakis, Krizanc, and Pelc [6].

Theorem 3.1. *Let $c \geq 1$. Then*

(i)

$$Hop_{P_n}(c) = \Theta\left(\frac{\log n}{\log c}\right) \text{ for } c \geq \log^{1+\varepsilon} n \quad \text{for any fixed } \varepsilon > 0,$$

(ii)

$$Hop_{M_n}(c) = \Theta\left(\frac{\log n}{\log c}\right) \quad \text{for } c \geq 2.$$

Proof. Both lower bounds easily follow by applying Lemma 2.3. Consider the upper bounds. To prove (i), first find m and $k = 2^i$ such that

$$2^m - 1 \leq n < 2^{m+1} - 1 \text{ and } (k-1)(m+2)/2 + 1 \leq c < (2k-1)(m+2)/2 + 1.$$

Setting $G_1 = F(k;2,m), G_2 = B_m$, and $G_3 = P_{2^m-1}$ in Lemma 2.2 and using a result of Lengauer (based on the orthogonal projection of B_m on a horizontal line) [8], $cg(B_m, P_{2^m-1}) \leq (m+2)/2$, and by Lemma 2.5 we get an embedding of $F(k;2,m)$ into P_{2^m-1} with

$$cg(F(k;2,m), P_{2^m-1}) \leq (k-1)(m+2)/2.$$

Now choose $n - 2^m + 1$ new vertices and join them to leaves of $F(k;2,m)$ (at most two per each leaf) in such a way that the resulting graph, say T, has $m+1$ levels. Note that this is always possible. Clearly, T has n vertices. It is easy to extend the above embedding of $F(k;2,m)$ into P_{2^m-1} to an embedding of T into P_n with

$$cg(T, P_n) \leq cg(F(k;2,m), P_{2^m-1}) + 1 \leq c.$$

Moreover, by Lemma 2.4,

$$Hop_{P_n}(c) \le diam(T) \le diam(F(k;2,m)) + 2 = 2\left\lceil \frac{m-1}{i} \right\rceil + 2 \le \frac{2m}{\log k} + 4$$
$$\le \frac{2\log(n+1)}{\log \frac{c}{2\log n}} + 4 \le \frac{2\log(n+1)}{\frac{\varepsilon}{1+\varepsilon}\log c - 1} + 4 = O\left(\frac{\log n}{\log c}\right)$$

for $c \ge \log^{1+\varepsilon} n$.

For (ii) assume first the case $n = 2^m, m \ge 1$. Find $k = 2^i$ such that

$$2(k-1) + 1 \le \frac{c}{4} < 2(2k-1) + 1.$$

Set $G_1 = F(k;2,2m), G_2 = B_{2m}$, and $G_3 = M_n$ in Lemma 2.2. Zienicke [11] showed that $cg(B_{2m}, M_n) = 2$. Now we add one new vertex and edge to $F(k;2,2m)$ resulting in a graph F on n vertices embeddable into M_n with $cg(F, M_n) \le 2(k-1)+1 \le c/4$. The diameter of F is increased by at most 1.

Second, consider an arbitrary n. Find m such that $2^m - 1 \le n < 2^{m+1} - 1$. Embed the graph F into the left lower submesh of M_n of size $2^m \times 2^m$. Similarly, embed the graph F into the other three "corner" submeshes of M_n. We get a virtual path layout for M_n with congestion at most $4cg(F, M_n) \le c$ and the hop-number

$$Hop_{M_n}(c) \le 2diam(F) \le 2(diam(F(k;2,2m)) + 1) \le 2\left\lceil \frac{2m-1}{i} \right\rceil + 2$$
$$\le 2\frac{2\log(n+1)}{\log(\frac{c+4}{16})} + 4 = O\left(\frac{\log n}{\log c}\right). \qquad \square$$

Note that $Hop_{P_n}(c)$ for nonconstant $c \le \log n$ remains an open problem.

4. Hypercubes. The main result of this section is an optimal virtual path layout for hypercubes. The n-dimensional hypercube, denoted by Q_n, is defined, by means of the Cartesian product of graphs, as $Q_0 = v$ (a single vertex) and $Q_n = Q_{n-1} \times P_2$. The edges of Q_n are divided into n groups in a natural way according to the dimensions they belong to.

Let us define the tree T_1 to be a single vertex. Now, for $n \ge 2$ let the tree $T_n = S_{2k-2} * kT_{n-i}$, where $k = 2^i \le n < 2^{i+1}$. Note that the number of vertices of T_n is $2^n - 1$.

LEMMA 4.1. *The diameter of T_n satisfies*

$$diam(T_n) < \frac{8n}{\lfloor \log n \rfloor} + 2\lfloor \log n \rfloor.$$

Proof. Let $depth(T_n)$ denote the depth of T_n. Then

$$depth(T_n) = depth(T_{n-i}) + 1,$$

where $i = \lfloor \log n \rfloor$. Solving this recurrence, we get

$$depth(T_n) \le \left\lceil \frac{n - 2^i + 1}{i} \right\rceil + \left\lceil \frac{2^i - 2^{i-1}}{i-1} \right\rceil + \cdots + \left\lceil \frac{2^2 - 2^1}{1} \right\rceil$$
$$< \sum_{j=1}^{i} \left\lceil \frac{2^j}{j} \right\rceil \le \sum_{j=1}^{i} \frac{2^j}{j} + i \le \frac{2^{i+2}}{i} + i \le \frac{4n}{\lfloor \log n \rfloor} + \lfloor \log n \rfloor,$$

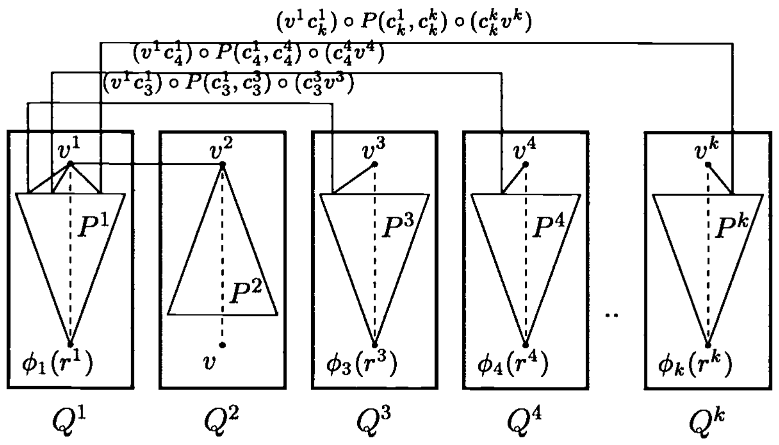

$(v^1c^1_k) \circ P(c^1_k, c^k_k) \circ (c^k_k v^k)$
$(v^1c^1_4) \circ P(c^1_4, c^4_4) \circ (c^4_4 v^4)$
$(v^1c^1_3) \circ P(c^1_3, c^3_3) \circ (c^3_3 v^3)$
v^1
v^2
v^3
v^4
v^k
P^1
P^2
P^3
P^4
P^k
$\phi_1(r^1)$
v
$\phi_3(r^3)$
$\phi_4(r^4)$
$\phi_k(r^k)$
Q^1
Q^2
Q^3
Q^4
Q^k

FIG. 2. *The k subhypercubes Q_{n-i}.*

where the estimation of the sum was done by a straightforward induction. Noting that $diam(T_n) \leq 2depth(T_n)$, we have the result. $\square$

Let G be a graph and let S (U) be a subset of E_G (V_G). We define the *graph induced by U under S* as the graph with the vertex set U and all edges $xy \in (U \times U) \cap S$.

Let S be the set of all edges in some $i \leq n$ dimensions of Q_n. Then the graph $Q_n - S$ consists of 2^i copies of Q_{n-i}, say $Q^1, Q^2, \ldots, Q^{2^i}$. Let v^1 be a vertex of Q^1. Let $v^l \in Q^l$, $2 \leq l \leq 2^i$, be the corresponding vertices to v^1, i.e., $v^l, 2 \leq l \leq i$, is the copy of v^1 chosen in the natural way. Then the graph induced by $\{v^1, v^2, \ldots, v^{2^i}\}$ under S is an i-dimensional hypercube, say $Q_i(v^1)$. Observe that the hypercubes $Q_i(x)$ for all $x \in Q^1$ are vertex disjoint. Moreover, the mapping $\xi : Q^1 \to Q^l$, $l = 2, \ldots, 2^i$, given by $\xi(x) = y$, where $y \in Q^l \cap Q_i(x)$, is the isomorphism induced by $Q_i(v^1)$.

LEMMA 4.2. *For all $n \geq 1$, $cg(T_n, Q_n) \leq 2$.*

Proof. In fact we prove, by induction on n, the following stronger statement. There exists an embedding (ϕ, ψ) of T_n into Q_n with $cg(e) \leq 2$, for all edges $e \in Q_n$, satisfying the following two additional conditions. Let r be the root of T_n and $v \in V_{Q_n} \setminus \phi[V_{T_n}]$.

(i) There exists a $\phi(r) - v$ path P such that each edge of P has congestion at most 1, and just one neighbor of v is on P.

(ii) For all neighbors x of v in Q_n, $cg(vx) = 0$.

Let us call the vertex v a *free vertex*, the path P a *free path*, and the embedding a *good embedding*. For $n = 1, 2, 3$ the statement is easy to observe. Thus we assume we have proved the statement for all integers up to $n - 1$ and we prove it for $n \geq 4$. By definition, $T_n = S_{2k-2} * kT_{n-i}$, where $k = 2^i \leq n < 2^{i+1}$. Let us delete all edges in some i dimensions of Q_n obtaining thereby k copies of Q_{n-i}, say $Q^1, \ldots, Q^k$ (see Figure 2).

For notational convenience let us denote some important vertices of T_n. By r we denote the root of T_n (this is also the root of S_{2k-2}). Further, by r^l we denote the leaf of S_{2k-2}, which is the root of $F^l = T_{n-i}$ $(l = 1, 2, \ldots, k)$. Note that r^l are the vertices at which the subtrees T_{n-i} are amalgamated with S_{2k-2} in T_n. Finally, by x^l $(l = 3, 4, \ldots, k)$ we denote the $k - 2$ remaining leaves of S_{2k-2}.

Let v^1 be any vertex of Q^1. In what follows we describe for $l = 1, \ldots, k$ a good embedding of F^l into Q^l. By the induction hypothesis there is a good embedding

(ϕ_1, ψ_1) of F^1 into Q^1. Since Q^1 is vertex transitive, we may assume that v^1 is the free vertex in Q^1. Moreover, there is a free $\phi_1(r^1) - v^1$ path, say P^1. By v^l we denote the vertex corresponding to v^1 in Q^l. It follows from the symmetry of Q_{n-i} that there is a good embedding (ϕ_2, ψ_2) of F^2 into Q^2 (obtained from the embedding of F^1 into Q^1) for which $\phi_2(r^2) = v^2$ and the free vertex in Q^2, say v, is the vertex corresponding to $\phi_1(r^1)$. Furthermore, there is a $v^2 - v$ free path P^2 in Q^2. Finally, for $l = 3, \ldots, k$, by the existence of the isomorphism induced by $Q_i(v^1)$, there is a good embedding (ϕ_l, ψ_l) of F^l into Q^l for which $\phi_l(r^l)$ is the vertex corresponding to $\phi_1(r^1)$ and v^l is the free vertex in Q^l. By the same argument, there is a $v^l - \phi_l(r^l)$ free path P^l in Q^l, $l = 3, \ldots, k$.

Since $Q^1 \cup Q^2 \cup \cdots \cup Q^k$ is a factor of Q_n and since $F^1 \cup F^2 \cup \cdots \cup F^k \cong kT_{n-i}$, the embeddings (ϕ_l, ψ_l), $l = 1, \ldots, k$, together induce an embedding (ϕ, ψ) of kT_{n-i} into $Q_n \setminus \{v, v^1, v^3, v^4, \ldots, v^k\}$. To obtain the required embedding we put $\phi(r) = v^1$ and $\phi(x^l) = v^l$ for $l = 3, 4, \ldots, k$. Finally, we have to embed the edges of S_{2k-2}. Since all the vertices of S_{2k-2} are already embedded, it is enough to describe the paths $v^1 - \phi(r^l)$, $l = 1, 2, \ldots, k$, and the paths $v^1 - v^l$, $l = 3, 4, \ldots, k$.

The path $v^1 - \phi(r^1) = P^1$ and the path $v^1 - \phi(r^2) = v^1v^2$. We construct the remaining paths using neighbors of the vertex v^1 in Q^1. Since the degree of Q_n is $n \geq k$ and since P^1 is a free path, there are $k-2$ neighbors of v^1 which lie neither on P^1 nor on v^1v^2, say $c_3^1, c_4^1, \ldots, c_k^1$. For each vertex c_j^1, $j = 3, 4, \ldots, k$, there is the hypercube $Q_i(c_j^1)$. Let c_j^l be the corresponding vertex for c_j^1 in Q^l. Since the mapping on corresponding vertices is an isomorphism, the vertex c_j^l is adjacent to the vertex v^l. Since $Q_i(c_j^1)$ $(j = 3, 4, \ldots, k)$ is connected, there is a $c_j^1 - c_j^l$ path, say $P(c_j^1, c_j^l)$, in $Q_i(c_j^1)$. Note that all $P(c_j^1, c_j^l)$ for $j = 3, 4, \ldots, k$ are vertex disjoint. Now, we are able to describe all remaining paths. For $j = 3, 4, \ldots, k$, the path $v^1 - v^j = (v^1c_j^1) \circ P(c_j^1, c_j^j) \circ (c_j^jv^j)$, and the path $v^1 - \phi(r^j) = (v^1c_j^1) \circ P(c_j^1, c_j^j) \circ (c_j^jv^j) \circ P^j$, where $\circ$ is the concatenation operation.

The embedding is now completely defined. It is an easy but time-consuming exercise to observe that $cg(e) \leq 2$ for each edge $e \in Q_n$. If we define $P = (v^1v^2) \circ P^2$, then since P^2 is a free path, P is a $\phi(r) - v$ free path as well. Moreover, no edge incident with v is used in the embedding, thus $cg(vx) = 0$ for all neighbors x of v and the embedding (ϕ, ψ) is good. □

Theorem 4.3. *For the n-dimensional hypercube we have*

$$Hop_{Q_n}(c) = \begin{cases} \Theta\left(\frac{n}{\log n}\right) & \text{if } 2 \leq c \leq n, \\ \Theta\left(\frac{n}{\log c}\right) & \text{otherwise.} \end{cases}$$

Proof. The upper bound of the first case follows from Lemmas 4.1 and 4.2. The lower bound is implied by Lemma 2.3. The second case is proved in a similar way as the result for the mesh using the fact from [7] that the complete binary tree on $2^n - 1$ vertices can be embedded in the n-dimensional hypercube with congestion 1. □

In particular, the above result says that if the congestion c is smaller than the degree of the hypercube, then the hop-number does not depend on c. It is an interesting open question whether this holds for other important networks of unbounded degree like k-ary de Bruijn and star graph. Another open problem is to find an optimal virtual paths layout for the n-vertex path if c is nonconstant and less than or equal to $\log^{1+\varepsilon} n$ for $\varepsilon > 0$.

REFERENCES

[1] I. Cidon, O. Gerstel, and S. Zaks, *A scalable approach to routing in ATM networks*, in Eighth International Workshop on Distributed Algorithms, Lecture Notes in Comput. Sci. 857, Springer-Verlag, Berlin, 1994, pp. 209–222.

[2] O. Gerstel, A. Wool, and S. Zaks, *Optimal layouts on a chain ATM network*, in 3rd Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 979, Springer-Verlag, Berlin, 1995, pp. 509–522.

[3] L. Gasieniec, E. Kranakis, D. Krizanc, and A. Pelc, *Minimizing congestion of layouts for ATM networks with faulty links*, in 21st International Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Comput. Sci. 1113, Springer-Verlag, Berlin, 1996, pp. 372–381.

[4] O. Gerstel and S. Zaks, *The virtual path layout problem in fast networks*, in 13th ACM Symposium on Principles of Distributed Computing, ACM, Baltimore, 1994, pp. 235–243.

[5] O. Gerstel and S. Zaks, *The virtual path layout problem in ATM rings and mesh networks*, in 1st Conference on Structure, Information and Communication Complexity, Carleton University Press, Ottawa, 1994, pp. 16–18.

[6] E. Kranakis, D. Krizanc, and A. Pelc, *Hop-congestion trade-offs for high-speed networks*, in 7th IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society, Los Alamitos, CA, 1995, pp. 662–668.

[7] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Mateo, CA, 1992.

[8] T. Lengauer, *Upper and lower bounds on the complexity of the min-cut linear arrangement problem on trees*, SIAM J. Alg. Disc. Meth., 3 (1982), pp. 99–113.

[9] D. E. McDysan and D. L. Spohn, *ATM: Theory and Applications*, Series on Computer Communication, McGraw–Hill, New York, 1995.

[10] J. Rolim, O. Sýkora, and I. Vrťo, *Optimal cutwidths of meshes*, in 21st International Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Comput. Sci. 1017, Springer–Verlag, Berlin, 1995, pp. 252–264.

[11] P. Zienicke, *Embeddings of treelike graphs into 2-dimensional meshes*, in 16th International Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Comput. Sci. 484, Springer–Verlag, Berlin, 1991, pp. 182–192.

SIAM J. COMPUT.
Vol. 29, No. 5, pp. 1630–1651

SELF-TESTING WITHOUT THE GENERATOR BOTTLENECK*

FUNDA ERGÜN†, S. RAVI KUMAR‡, AND D. SIVAKUMAR§

Abstract. Suppose P is a program designed to compute a function f defined on a group G. The task of self-testing P, that is, testing if P computes f correctly on most inputs, usually involves testing explicitly if P computes f correctly on *every* generator of G. In the case of multivariate functions, the number of generators, and hence the number of such tests, becomes prohibitively large. We refer to this problem as the *generator bottleneck*. We develop a technique that can be used to overcome the generator bottleneck for functions that have a certain nice structure, specifically if the relationship between the values of the function on the set of generators is easily checkable. Using our technique, we build the first efficient self-testers for many linear, multilinear, and some nonlinear functions. This includes the FFT, and various polynomial functions. All of the self-testers we present make only $O(1)$ calls to the program that is being tested. As a consequence of our techniques, we also obtain efficient program result-checkers for all these problems.

Key words. program correctness, self-testing, generator bottleneck

AMS subject classifications. 68Q25, 68Q40, 68Q60

PII. S0097539796311168

1. Introduction.

The notions of program result-checking, self-testing, and self-correcting as introduced in [4, 17, 5] are powerful tools for attacking the problem of program correctness. These methods offer both realistic and efficient tools for software verification. Various useful mathematical functions have been shown to have self-testers and self-correctors; some examples can be found in [5, 3, 17, 9, 14, 19, 1, 20, 22, 6]. The theoretical developments in this area are at the heart of the recent breakthrough results on probabilistically checkable proofs and the subsequent results that show nonapproximability of hard combinatorial problems.

Suppose we are given a program P designed to compute a function f. Informally, a *self-tester* for f distinguishes the case where P computes f correctly always from the case where P errs frequently. A *result-checker* for a function f takes as input a program P and an input q to P and outputs PASS when P correctly computes f always and outputs FAIL if $P(q) \neq f(q)$. Given a program P that computes f correctly on most inputs, a *self-corrector* for f is a program $P_{\rm sc}$ that uses P as an oracle and computes f correctly on every input with high probability.

*Received by the editors October 28, 1996; accepted for publication (in revised form) December 12, 1997; published electronically March 17, 2000. This paper unifies the preliminary versions which appeared in *Proc. 27th Annual Symposium on the Theory of Computing* and in *Proc. 15th Annual Foundations of Software Technology and Theoretical Computer Science.*

http://www.siam.org/journals/sicomp/29-5/31116.html

†Department of Computer Science, Cornell University, Ithaca, NY 14853-7501 (ergun@cs.cornell.edu). The research of this author was partially supported by ONR Young Investigator Award N00014-93-1-0590, the Alfred P. Sloan Research Award, and NSF grant DMI-91157199.

‡Department of Computer Science, Cornell University, Ithaca, NY 14853-7501 (ravi@cs.cornell.edu). The research of this author was partially supported by ONR Young Investigator Award N00014-93-1-0590, the Alfred P. Sloan Research Award, and NSF grant DMI-91157199.

§Department of Computer Science, University of Houston, Houston, TX 77204 (siva@cs.uh.edu). Most of the research of this author was performed while at SUNY/Buffalo and supported in part by K. Regan's NSF grant CCR-9409104.

1.1. Definitions and basics. Before we discuss our results, we present the basic definitions of testers, checkers, etc., and state some desirable properties of these programs. Let f be a function on a domain $\mathcal{D}$ and let P be a program that purports to compute f. The testers, correctors, and checkers we define are probabilistic programs that take P as an oracle, and in addition, take one or both of the following parameters as input: an *accuracy parameter* ϵ that specifies the conditions that P is expected to meet and a *confidence parameter* ρ that is an upper bound on the probability that the tester/corrector/checker fails to do its job. The following definitions formalize the notions of self-tester [5], self-corrector [5, 17], and result-checker [4].

Definition 1.1 (self-tester). *An* ϵ-self-tester *for f is a probabilistic oracle program T that, given $\rho > 0$, satisfies the following conditions*:

- $\Pr_{x\in\mathcal{D}}[P(x) = f(x)] = 1 \Rightarrow \Pr[T^P$ *outputs* PASS $] = 1$, *and*
- $\Pr_{x\in\mathcal{D}}[P(x) \neq f(x)] \geq \epsilon \Rightarrow \Pr[T^P$ *outputs* FAIL $] \geq 1-\rho$.

Definition 1.2 (self-corrector). *An* ϵ-self-corrector *for f is a probabilistic oracle program $P_{\rm sc}$ that, given any input y, and $\rho > 0$, satisfies the following condition*:

- $\Pr_{x\in\mathcal{D}}[P(x) = f(x)] \geq 1-\epsilon \Rightarrow \Pr[P_{\rm sc}(y) = f(y)] \geq 1-\rho$.

Definition 1.3 (result-checker). *A* checker *(or* result-checker*) for f is a probabilistic oracle program C that, given an input y and $\rho > 0$, satisfies the following conditions*:

- $\Pr_{x\in\mathcal{D}}[P(x) = f(x)] = 1 \Rightarrow C^P(y)$ *outputs* PASS, *and*
- $P(y) \neq f(y) \Rightarrow \Pr[C^P(y)$ *outputs* FAIL $] \geq 1-\rho$.

We now list three important properties that are required of self-testers, self-correctors, and result-checkers. For definiteness, we state these for the case of self-testers. First, the self-tester T should be computationally *different* from and more *efficient* than any program that computes f [4]. This restriction ensures that T does not implement the obvious algorithm to compute f (and hence could harbor the same set of bugs, or be computationally inefficient). Furthermore, this ensures that the running time of T is *asymptotically better than* the running time of the best *known* algorithm for f. The second important property required of T is that it should not require the knowledge of too many correct values of f. In particular, this rules out the possibility that T merely keeps a large table of the correct values of f for all inputs. The third important property required of a self-tester is *efficiency*: an efficient self-tester should only make $O(1/\epsilon, \lg(1/\rho))$ calls to P. For constant ϵ and ρ, an efficient self-tester makes only $O(1)$ calls to the program. (In the rest of the paper, we often write $O(1)$ as a shorthand for $O(1/\epsilon, \lg(1/\rho))$, particularly when discussing the dependence on other parameters of interest.)

The following well-known lemma summarizes some relationships between the notions of self-testers, self-correctors, and result-checkers. For the reader's convenience, we sketch the idea of the proof of this lemma, suppressing the details of the accuracy and confidence parameters.

Lemma 1.4 (see [5]). (a) *If f has a self-tester and a self-corrector that make $O(1)$ calls to the program, then f has a result-checker that makes $O(1)$ calls to the program.*

(b) *If f has a result-checker, then it has a self-tester.*

Proof (sketch). For part (a), suppose that f has a self-tester and a self-corrector. Given an input y and oracle access to a program P, first self-test P to ensure that it doesn't err too often. If the self-tester finds P to be too erroneous, output FAIL. Otherwise, compute $f(y)$ by using the self-corrector for $P_{\rm sc}$ and the program P, and output PASS iff $P(y) = P_{\rm sc}(y)$.

Clearly a perfect program always passes. Suppose $P(y) \neq f(y)$. Then one of the following two cases must occur. The program is too erroneous, in which case the self-tester, and hence the checker, outputs FAIL. The program is not too erroneous, in which case the self-corrector computes $f(y)$ correctly with high probability, so the checker detects that $P(y) \neq P_{\mathrm{sc}}(y)$ and outputs FAIL.

For part (b), suppose that f has a result-checker. By using the result-checker to test if $P(x) = f(x)$ for many randomly chosen inputs x, the fraction of inputs x for which $P(x) \neq f(x)$ can be estimated. Output PASS iff this fraction is less than ϵ. □

A useful tool in constructing self-correctors is the notion of *random self-reducibility*. The fine details of this notion are beyond the scope of this paper, and we refer the reader to the papers [3, 17] (see also the survey paper [11]). Informally, a function f is randomly self-reducible if evaluation of f on an input can be reduced efficiently to the evaluation of f on one or more random inputs. For a brief example, note that linear functions are randomly self-reducible: to compute $f(x)$, it suffices to pick a random r, compute $f(x+r)$ and $f(r)$, and finally obtain $f(x) = f(x+r) - f(r)$. All functions that we consider in this paper are efficiently randomly self-reducible; therefore, whenever required we will always assume that efficient self-correction is possible.

1.2. Building self-testers using properties.

The process of self-testing whether a program P computes a function f correctly on most inputs is usually a two-step strategy. It is necessary to first perform some tests to verify that P agrees on most inputs with a function g that belongs to a certain class $\mathcal{F}$ of functions that contains f. Some additional tests are then needed to verify that the function g is, in fact, the intended function f.

The standard way to test whether P agrees with some function in a class $\mathcal{F}$ of functions is based on the notion of a *robust property*. Informally, property $\mathcal{I}$ is said to be a robust characterization of a function family $\mathcal{F}$ if the following two conditions hold: (1) every $f \in \mathcal{F}$ satisfies $\mathcal{I}$, and (2) if P is a function (program) that satisfies $\mathcal{I}$ for most inputs, then P must agree with some $g \in \mathcal{F}$ on most inputs. For example, Blum, Luby, and Rubinfeld [5] establish that the property of linearity $f(x+y) = f(x) + f(y)$ serves as a robust property for the class of all linear functions, and use this to build self-testers for linear functions. This generic technique was first formalized in [20].

DEFINITION 1.5 (robust property). *A* property *is a predicate* $\mathcal{I}^f(\vec{x} = x_1, \ldots, x_k)$. *A property* $\mathcal{I}^f(\vec{x})$ *is* (ϵ, δ)-robust *for a class of functions* $\mathcal{F}$ *over a domain* $\mathcal{D}$ *if it satisfies the following conditions*:

- $(\forall f)\left[f \in \mathcal{F} \iff (\forall \vec{x} \in \mathcal{D}^k)[\mathcal{I}^f(\vec{x}) = \mathsf{TRUE}]\right]$.
- *If a function (program) P satisfies* $\Pr_{\vec{x} \in \mathcal{D}^k}[\mathcal{I}^P(\vec{x}) = \mathsf{TRUE}] \geq 1 - \epsilon$, *then there is a function g such that*
  - ◦ $(\forall \vec{x} \in \mathcal{D}^k)[\mathcal{I}^g(\vec{x}) = \mathsf{TRUE}]$, *and*
  - ◦ $\Pr_{x \in \mathcal{D}}[g(x) = P(x)] \geq 1 - \delta$,

that is, $(\exists g \in \mathcal{F})$ *such that P agrees with g on all but δ fraction of inputs.*

We now outline the process of building self-testers using robust properties (cf. [5]). Let $\mathcal{D}$ be a (finite) group with generators $e_1, \ldots, e_n$, and let $\mathcal{F}$ denote some class of functions from $\mathcal{D}$ into some range $\mathcal{R}$. Further assume that the functions in $\mathcal{F}$ possess the property of random self-reducibility and can hence be self-corrected efficiently. Suppose P is a program that purports to compute a specific function $f \in \mathcal{F}$. Let $\mathcal{I}^f(\vec{x})$ be a robust property that characterizes $\mathcal{F}$.

As mentioned earlier, the process of building self-testers is a two-step process. In the first step, we will ensure that that the program P agrees with some function

$g \in \mathcal{F}$ on most inputs. To do this, we use the fact that $\mathcal{I}^f$ is a robust property that characterizes $\mathcal{F}$. Specifically, the self-tester will estimate the fraction of k-tuples $\vec{x} \in \mathcal{D}^k$ for which $\mathcal{I}^f(\vec{x})$ holds. If this fraction is at least $1-\epsilon$, then by the robustness of $\mathcal{I}^f$, it follows that there is some $g \in \mathcal{F}$ that agrees with P on all but δ fraction of $\mathcal{D}$. The required estimation can be carried out by a random sampling of $\vec{x}$ and testing the property $\mathcal{I}^f$.

The next step is to verify that the function g is the same as the function f that P purports to compute. This is achieved by testing that $g(e_i) = f(e_i)$ for every *generator* of the group $\mathcal{D}$. If this is true, then by an easy induction it would follow that $g \equiv f$. An important point to be made here is that the self-tester has access only to P and not to g; the function g is only guaranteed to exist. Nevertheless, the required values of g may be obtained by using a self-corrected version P_{sc} of P. Another point worth mentioning is that to carry out this step, the self-tester needs to know the values of f on every generator of $\mathcal{D}$.

1.3. The generator bottleneck. An immediate application of the basic method outlined above to functions whose domains are vector spaces of large dimension suffers from a major efficiency drawback. For example, if the inputs to the function f are n-dimensional vectors (or $n \times n$ matrices), then the number of generators of the domain is n (resp., n^2). The straightforward approach of exhaustively testing if P_{sc} agrees with f on each generator makes n (resp., n^2) calls to P; furthermore, the self-tester built on this approach requires the knowledge of the correct value of f on n (resp., n^2) generators. When n is large, this makes the overhead in the self-testing process too high. This issue is called the *generator bottleneck* problem.

In this paper, we address the generator bottleneck problem and solve it for a fairly large class of functions that satisfy some nice structural properties. The self-testers that we build are not only useful in themselves, but are also useful in building efficient result-checkers, which are important for practical applications.

1.4. Our results. We present a fairly general method of overcoming the generator bottleneck and testing multivariate functions by making only $O(1)$ calls to the program being tested.

First we investigate the problem of *multivariate linear functions* (i.e., the functions f satisfying $f(\vec{x})+f(\vec{y}) = f(\vec{x}+\vec{y})$). We show a general technique that can be applied in a natural vector space setting. The main idea is to obtain an easy and uniform way of "generating" all generators from a single generator. Using this idea, we give a simple and powerful condition for a linear function f to be efficiently self-testable on a large vector space. We then apply this scheme to obtain very efficient self-testers for many functions. This includes polynomial differentiation (of arbitrary order), polynomial integration, polynomial "mod" function, etc. We also obtain the first efficient self-tester for Fourier transforms.

We then extend this method to the case of *multilinear functions* (i.e., functions f that are linear in each variable when the other variables are fixed). We build an efficient tester for polynomial multiplication as a consequence. Another application we give is for large finite fields: we show that multilinear functions over finite field extensions of dimension n can be efficiently self-tested with $O(1)$ calls, independent of the dimension n. We also provide a new efficient self-tester for matrix multiplication.

We next extend the result to some *nonlinear functions.* We give self-testers for exponentiation functions that avoid the generator bottleneck. For example, consider the function that computes the square of a polynomial over a finite field: $f(q) = q^2$. Here we do not have the linearity property that is crucial in the proof for the linear functions. Instead, we use the fact that the Lagrange interpolation identity (cf. Fact 4.1) for polynomials gives a robust characterization. We exhibit a self-tester for the function $f(q) = q^d$ that makes $O(d)$ calls to the program being tested. Extending the technique when f is a constant degree exponentiation to the case when f is a constant degree polynomial (e.g., $f(q) = q^d + q + 1$, where q is a polynomial over a finite field) is much harder. First we show a reduction from multiplication to the computation of low-degree polynomials. Using this reduction and the notion of a result-checker, we construct a self-tester for degree d polynomials over finite field extensions of dimension n that make $O(2^d)$ calls to the program being tested.

1.5. Related work. One method that has been used to get around the generator bottleneck has been to exploit the property of *downward self-reducibility* [5]. The self-testers that use this property, however, have to make $\Omega(\log n)$ calls to the program depending on the way the problem decomposes into smaller problems. For instance, a tester for the permanent function of $n \times n$ matrices makes $O(n)$ calls to the program, whereas a tester for polynomial multiplication that uses similar principles makes $O(\log n)$ calls. In [5] a bootstrap tester for polynomial multiplication that makes $O(\log n)$ calls to the program being tested is given. It is already known that matrix multiplication can be tested (without any calls to the program) using a result-checker due to Freivalds [13]. The idea of Freivalds' matrix multiplication checker can also be adapted to build testers for polynomial multiplication that make no calls to the program being tested. This approach, however, requires the underlying field to be large (have at least $(2+\gamma)n$ elements, where n is the degree of the polynomials being multiplied, and γ is a positive constant). Moreover, this scheme requires the tester to perform polynomial evaluations, whereas ours does not. For Fourier transforms, a different result-checker that uses preprocessing has been given independently in [6].

A useful fact. The following fact, a variant of the well-known Chernoff–Hoeffding bounds, is often very useful in obtaining error-bounds in sampling 0/1 random variables [15].

FACT 1.6. *Let $Y_1, Y_2, \ldots$ be independently and identically distributed 0/1 random variables with means μ. Let $\theta \leq 2$. If $N \geq (1/\mu)(4\ln(2/\rho)/\theta^2)$, then $\Pr[(1-\theta)\mu \leq \widetilde{Y} \leq (1+\theta)\mu] \geq 1 - \rho$, where $\widetilde{Y} = \sum_{i=1}^{N} Y_i/N$.*

Organization of the paper. Section 2 discusses the scheme for linear functions over vector spaces; section 3 extends the scheme for multilinear functions; section 4 outlines the approach for nonlinear functions.

2. Linear functions over vector spaces.

In this section, we address the problem of self-testing linear functions on a vector space without the generator bottleneck. We demonstrate a general technique to self-test without the generator bottleneck and provide several interesting applications of our technique.

Definitions. Let V be a vector space of finite dimension n over a field $\mathbb{K}$, and let f be a function from V into a ring R. We are interested in building a self-tester for the case where $f(\cdot)$ is a *linear* function, that is, $f(c\alpha + \beta) = cf(\alpha) + f(\beta)\ \forall\ \alpha, \beta \in V$ and $c \in \mathbb{K}$. For $1 \leq i \leq n$, let e_i denote the unit vector that has a 1 in the ith position and 0's in the other positions. The vectors $e_1, e_2, \ldots, e_n$ form a collection of basis vectors that span V. Viewed as an Abelian group under vector addition, V is generated by $e_1, \ldots, e_n$. We assume that the field $\mathbb{K}$ is finite, since it is not clear how to choose a random element from an infinite field.

The property of linearity $\mathcal{I}^f(\alpha, \beta) \equiv [f(\alpha + \beta) = f(\alpha) + f(\beta)]$ was shown to be robust in [5]. Using this and the generic construction of self-testers from robust properties, one obtains the following self-tester for the function f:

```
Property Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $\alpha, \beta \in_R V$
    Verify $P(\alpha+\beta) = P(\alpha) + P(\beta)$
    Reject if the test fails

Generator Tests:
  For $i = 1$ to $n$
    Verify $P_{\text{sc}}(e_i) = f(e_i)$
```

If P passes the `Property Test` then we are guaranteed the existence of a linear function g that is close to P. There are, however, two problems with the `Generator Tests`: one is that the self-tester is inefficient—if the inputs are vectors of size n, the self-tester makes $O(n)$ calls to the program, which is not desirable. Second, the self-tester needs to know the correct value of f on n different points, which is also undesirable. Our primary interest is to avoid this generator bottleneck and solve both of the problems mentioned. The key idea is to find an easy and uniform way that "converts" one generator into the next generator. We illustrate this idea through the following example.

Example. Let $\mathcal{P}_n$ denote the additive group of all degree n polynomials over a field $\mathbb{K}$. The elements $1, x, x^2, \ldots, x^n$ generate $\mathcal{P}_n$, and multiplying any generator x^k by x gives the next generator x^{k+1}. For a polynomial $q \in \mathcal{P}_n$ and a scalar $c \in \mathbb{K}$, let $E_c(q)$ denote the function that evaluates $q(c)$. Clearly E_c is linear and satisfies the simple relation $E_c(xq) = cE_c(q)$. Suppose P is a program that purports to compute E_c, and assume that P has passed `Property Test` given above. Then we know by robustness of linearity that there is a linear function g that agrees with P on most inputs. Note that g can be computed correctly with high probability via the self-corrector (which is easy to construct for linear functions [5]). Now, rather than verify that $g(x^k) = E_c(x^k)$ for all generators x^k of $\mathcal{P}_n$, we may instead verify that g satisfies the property $g(xq) = cg(q)$ everywhere. By an easy induction, this implies that g agrees with E_c at all the generators. By linearity of g, it follows that g agrees with E_c on all inputs.

We are now faced with the task of verifying $g(xq) = cg(q)\ \forall\ q \in \mathcal{P}_n$, which is too expensive to be tried explicitly and exhaustively. Instead, we prove that it suffices to check with $O(1)$ tests that $g(xq) = cg(q)$ almost every place that we look at. That is, pick many random $q \in \mathcal{P}_n$, ask the program P_{sc} to compute the values of $g(q)$ and $g(xq)$, and cross-check that $g(xq) = cg(q)$ holds. In other words, we prove that the property $\mathcal{J}^g(q) \equiv [g(xq) = cg(q)]$ is robust (in a restricted sense) under the assumption that g is linear. (In its most general interpretation, robustness guarantees the existence of h that satisfies $h(xq) = ch(q)\ \forall\ q \in \mathcal{P}_n$, and that agrees with g on a large fraction of inputs. We actually show that $h \equiv g$, hence the "restricted sense.") The self-tester needs to know the value of f on only one point, in contrast to n, as in the original approach of [5].

Generalization via the basis rotation function θ. We note that this idea has a natural generalization to vector spaces. Let θ denote the *basis rotation function*, i.e., the linear operator on a vector space V that "rotates" the coordinate axes that span V. θ, which can be viewed as a matrix, defines a one-to-one correspondence from the set of basis vectors to itself: for every i, $\theta(e_i) = e_{(i+1) \bmod n}$. The computational payoff is achieved when there is a simple relation between $f(\alpha)$ and $f(\theta(\alpha))$ for all vectors $\alpha \in V$. More specifically, we show that the generator bottleneck can be avoided if there is an easily computable function $h_{\theta,f}$ such that

$$f(\theta(\alpha)) = h_{\theta,f}(\alpha, f(\alpha)),$$

$\forall\ \alpha \in V$. (For instance, for polynomial evaluation E_c, $f(\theta(q)) = c \cdot f(q), \forall q \in \mathcal{P}_n$.) From here on, when obvious, we drop the suffix f and simply denote $h_{\theta,f}$ as h_θ. If the function f is linear, the linearity of θ implies that h_θ is linear in its second argument in the following sense: $h_\theta(\alpha+\beta, f(\alpha+\beta)) = h_\theta(\alpha, f(\alpha)) + h_\theta(\beta, f(\beta))$. What is more important is that h_θ be easy to compute, given just α and $f(\alpha)$. Using this scheme, we show that many natural functions f have a suitable candidate for h_θ. The `Generator Tests` of [5] can now be replaced by:

```
Basis Test:
  Verify $P_{\rm sc}(e_1) = f(e_1)$

Inductive Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $\alpha \in_R V$
    Verify $P_{\rm sc}(\theta(\alpha)) = h_\theta(\alpha, P_{\rm sc}(\alpha))$
    Reject if the test fails
```

The following theorem proves that this replacement is valid.

THEOREM 2.1. *Suppose f is a linear function from the vector space V into a ring R, and suppose P is a program for f.*

(a) *Let $\epsilon < 1/2$, and suppose P satisfies the following condition*:

(1) $\Pr_{\alpha,\beta\in V}[P(\alpha+\beta) \neq P(\alpha) + P(\beta)] \leq \epsilon$. *Then the function g defined by $g(\alpha) = \text{majority}_{\beta\in V}\{P(\alpha+\beta) - P(\beta)\}$ is a linear function on V, and g agrees with P on at least $1-2\epsilon$ fraction of the inputs.*

(b) *Furthermore, suppose $h_\theta(\alpha, f(\alpha)) = f(\theta(\alpha))$ and g satisfies the following conditions*:

(2) $g(e_1) = f(e_1)$.

(3) $\Pr_{\alpha\in V}[g(\theta(\alpha)) \neq h_\theta(\alpha, g(\alpha))] \leq \epsilon$, *where α is such that $\theta(\alpha)$ is defined. Then $g(\alpha) = f(\alpha)\ \forall\ \alpha \in V$.*

Remarks. The above theorem merely lists a set of properties. The fact that this set yields a self-tester is presented in Theorem 2.2. Note that hypotheses (1), (2), and (3) above are *conditions* on P and g, not *tests* performed by a self-tester.

Proof. The proof that the function g is linear and $P_{\rm sc}$ computes g (with high probability) is due to [5]. For the rest of this proof, we will assume that g is linear and that it satisfies conditions (2) and (3) above.

We first argue that it suffices to prove that if the conditions hold, then for every $\alpha \in V$, $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$. By condition (2), g agrees with f on the first basis vector. For $i > 1$, the basis vector e_i can be obtained by $\theta(e_{i-1})$. If g satisfies $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$ everywhere, it would follow that g computes f correctly on all the basis vectors. Finally, since g is linear, it computes f correctly on all of V, since the vectors in V are just linear combinations of the basis vectors.

Now we show that condition (3) implies that $\forall \alpha \in V$, $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$. Fix an arbitrary element $\alpha \in V$. We will show that the probability over a random $\beta \in V$

that $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$ is positive. Since the equality is independent of β and holds with nonzero probability, it must be true with probability 1. Now

$$\begin{aligned}\Pr_{\beta\in V}\big[g(\theta(\alpha)) &= g(\theta(\beta+\alpha-\beta))\\ &= g(\theta(\beta)+\theta(\alpha-\beta))\\ &= g(\theta(\beta))+g(\theta(\alpha-\beta))\\ &= h_\theta(\beta, g(\beta)) + h_\theta(\alpha-\beta, g(\alpha-\beta))\\ &= h_\theta(\beta+\alpha-\beta, g(\beta)+g(\alpha-\beta))\\ &= h_\theta(\alpha, g(\alpha))\big] \geq 1-2\epsilon > 0.\end{aligned}$$

The first equality in the above is just rewriting. The second equality follows from the linearity of θ. The third equality follows from the fact that g is linear. If the random variable β is distributed uniformly in V, the random variables β and $\alpha - \beta$ are distributed identically and uniformly in V. Therefore, by the assumption that g satisfies condition (3), the fourth equality fails with probability at most 2ϵ. The fifth equality uses the fact that h_θ is linear, and the last equality uses the fact that g is linear. □

The foregoing theorem shows that if P (and g) satisfy certain conditions, then g, which can be computed using P, is identically equal to the function f. The self-tester comprises the following tests: `Linearity Test`, `Basis Test`, and `Inductive Test`.

THEOREM 2.2. *For any $\rho < 1$ and $\epsilon < 1/2$, the above three tests compose a 2ϵ-self-tester for f. That is, if a program P computes f correctly on all inputs, then the self-tester outputs* PASS *with probability* 1, *and if P computes f incorrectly on more than 2ϵ fraction of the inputs, then the self-tester outputs* FAIL *with probability at least $1-\rho$.*

Proof. In performing the three tests, the self-tester is essentially estimating the probabilities listed in conditions (1), (2), and (3) of the hypothesis of Theorem 2.1. Note that condition (2) does not involve any probability; rather, the self-tester uses P_{sc} to compute $g(e_1)$. By choosing $O((1/\epsilon)\log(1/\rho))$ samples in `Linearity Test` and `Inductive Test` and by using the self-corrector with confidence parameter $\rho/3$ in `Basis Test` the self-tester ensures that its confidence in checking each condition is at least $1-(\rho/3)$.

Clearly if P always computes f correctly, the tester always outputs PASS. Conversely, suppose the tester outputs PASS. Then with probability $1-\rho$, the hypotheses of Theorem 2.1 are true. By the conclusion of Theorem 2.1, it follows that a function g that is identical to f exists, and that g equals P on at least $1-2\epsilon$ fraction of the inputs. □

2.1. Applications. We present some applications of Theorem 2.2. We remind the reader that a linear function f on a vector space V is efficiently self-testable without the generator bottleneck if there is a (linear) function h_θ that is easily computable and that satisfies $f(\theta(\alpha)) = h_\theta(\alpha, f(\alpha))\ \forall\ \alpha \in V$. In each of our applications f, we show that a suitable function $h = h_{\theta,f}$ exists that satisfies the above conditions. Recall the example of the polynomial evaluation function $E_c(q) = q(c)$, where the identity $E_c(xq) = cE_c(q)$ holds; in the applications below, we will establish only similar relationships. Also, for the sake of simplicity, we do not give all the technical parameters required; these can be computed by routine calculations following the proofs of the theorems in the last section.

Our applications concern linear functions of polynomials. We obtain self-testers for polynomial evaluation, Fourier transforms, polynomial differentiation, polynomial integration, and the *mod* function of polynomials. Moreover, the vector space setting lets us state some of these results in terms of the matrices that compute linear transforms of vector spaces.

Let $\mathcal{P}_n \subseteq \mathbb{K}[x]$ denote the group of polynomials in x of degree $\leq n$ over a field $\mathbb{K}$. The group $\mathcal{P}_n$ forms a vector space under usual polynomial addition and scalar multiplication by elements from $\mathbb{K}$. The polynomials $1, x, x^2, \ldots, x^n$ span $\mathcal{P}_n$, and a polynomial $q(x) = \sum_{i=0}^{n} q_i x^i$ has the vector representation $(q_n, q_{n-1}, \ldots, q_1, q_0)$. The basis rotation function θ in this case is just multiplication by x, thus $\theta(q) = xq$. Note that multiplying q by x results in a polynomial of degree $n+1$. To handle this minor detail, we will assume that the program works over the domain $\mathcal{P}_{n+1}$, and we conclude correctness over $\mathcal{P}_n$.

Polynomial evaluation. For any $c \in \mathbb{K}$, let $E_c(q)$ denote, as described before, the function that returns the value $q(c)$. This function is linear. Moreover, the relation between $E_c(xq)$ and $E_c(q)$ is simple and linear: $E_c(xq) = cE_c(q)$. To self-test a program P that claims to compute E_c, the `Inductive Test` is simply to choose many random q's, and verify that $P_{\text{sc}}(xq) = cP_{\text{sc}}(q)$ holds.

Vandermonde operators and the discrete Fourier transform. If $u_1, u_2, \ldots, u_{n+1}$ are $n+1$ distinct elements of $\mathbb{K}$, then one may wish to evaluate a polynomial $q \in \mathcal{P}_n$ simultaneously on all $n+1$ points. The ideas for E_c extend easily to this case, for $E_u(xq) = uE_u(q)$ for any $u \in \mathbb{K}$, and these relations hold simultaneously.

Let ω be a principal $(n+1)$-st root of unity in $\mathbb{K}$. The operation of converting a polynomial from its coefficient representation to pointwise evaluation at the powers of ω is known as the discrete Fourier transform (DFT). DFT has many fundamental applications that include fast multiplication of integers and polynomials. With our notation, the DFT of a polynomial $q \in \mathcal{P}_n$ is simply $F(q) = (E_{\omega^0}(q), E_{\omega^1}(q), \ldots, E_{\omega^n}(q))$. The DFT F is linear, and $F(xq) = (\omega^0 E_{\omega^0}(q), \ldots, \omega^n E_{\omega^n}(q))$. Notice that here the function h is really n coordinate functions h_{ω^i} for $0 \leq i \leq n$. The self-tester will simply choose q's randomly, request the program to compute $F(q)$ and $F(xq)$, and verify for each i, $0 \leq i \leq n$, that $(F(xq))[i] = \omega^i (F(q))[i]$ holds.

This suggests the following generalization (for the case of arbitrary vector spaces). Simultaneous evaluation of a polynomial at $d+1$ points $u_1, \ldots, u_{n+1}$ corresponds to multiplying the vector p by a Vandermonde matrix M, where $M_{ij} = u_i^{j-1}$. The ideas used to test simultaneous evaluation of polynomials and the DFT extend to give a self-tester for any linear transform that is represented by a Vandermonde matrix.

The matrix for the DFT can be written as a Vandermonde matrix F, where $F_{ij} = \omega^{ij}$. The inverse of the DFT, that is, converting a polynomial from pointwise representation to coefficient form, also has a Vandermonde matrix whose entries are given by $\widetilde{F}_{ij} = (1/\det F)\omega^{-ij}$. It follows that the inverse Fourier transform can be self-tested efficiently. Another point worth mentioning here is that in carrying out the `Inductive Test` the self-tester does not have to compute $\det F$. All it needs to do is verify that for many randomly chosen q's, the identity $(\widetilde{F}(xq))[i] = \omega^{-i}(\widetilde{F}(q))[i]$ holds.

Operators in elementary Jordan canonical form. A linear operator M is said to be in elementary Jordan canonical form if all the diagonal entries of M are c for some $c \in \mathbb{K}$, and all the elements to the left of the main diagonal (the first non principal diagonal in the lower triangle of M) are 1's. It is easy to verify that $M\theta = \theta M + M'$, where M' is a matrix that has a -1 in the top left corner and a 1 in the bottom right corner and zeroes elsewhere. Therefore, for every $v = (v_n, v_{n-1}, \ldots, v_2, v_1)^T$ in the vector space, $M(\theta(v)) = \theta(M(v)) + (-v_n, 0, \ldots, 0, v_1)^T$. This gives an easy way to implement the `Inductive Test` in the self-tester.

An attempt to extend this to matrices in Jordan canonical form, or even to diagonal matrices, seems not to work. If, however, a diagonal or shifted diagonal matrix has a special structure, then we can obtain self-testers that avoid the generator problem. For example, the matrix corresponding to the differentiation of polynomials has a special structure: it contains the entries $n, n-1, \ldots, 1$ on the diagonal above the main diagonal.

Differentiation and integration of polynomials. Differentiation of polynomials is a linear function $D : \mathcal{P}_n \to \mathcal{P}_{n-1}$. We have the explicit form for h: $D(xq) = q + xD(q)$. Integration of polynomials is a linear function $I : \mathcal{P}_n \to \mathcal{P}_{n+1}$. The explicit form for h is $I(xq) = xI(q) - I(I(q))$. Even though this does not readily fit into our framework (since it is not of the form $I(xq) = h(q, I(q))$), the proof of Theorem 2.1 can be easily modified to handle this case using the linearity of I. For completeness, we spell out the details for the robustness of the `Inductive Test`, which is the only change required.

LEMMA 2.3. *If $g : \mathcal{P}_n \to \mathcal{P}_{n+1}$ is a linear function that satisfies $\Pr_{q\in\mathcal{P}_n}[g(xq) \neq xg(q) - g(g(q))] = \epsilon < 1/2$, then $\Pr_{q\in\mathcal{P}_n}[g(xq) = xg(q) - g(g(q))] = 1$.*

Proof.

$$\begin{aligned}\Pr_{r\in\mathcal{P}_n}\big[g(xq) &= g(x(r+q-r))\\ &= g(xr + x(q-r))\\ &= g(xr) + g(x(q-r))\\ &= xg(r) - g(g(r)) + xg(q-r) - g(g(q-r))\\ &= x(g(r) + g(q-r)) - g(g(r) + g(q-r))\\ &= xg(q) - g(g(q))\big] \geq 1 - 2\epsilon > 0.\end{aligned}$$

Since the event $g(xq) = xg(q) - g(g(q))$ holds with positive probability and is independent of r, it holds with probability one. ☐

Thus we can avoid the generator bottleneck for these functions. This can be considered as a special case of the previous application.

Higher order differentiation of polynomials. Let D^k denote the kth differential operator. It is easy to write a recurrence-like identity for D^k in terms of $D^j, j < k$. This gives us a self-tester only in the library setting described in [5, 21], where one assumes that there are programs to compute all these differential operators. If we wish to self-test a program that only computes D^k and have no library of lower-order differentials, this assumption is not valid. To remedy this, we will use the following lemma, which is proved in the appendix.

LEMMA 2.4. *If q is a polynomial in x of degree $\geq k$, then*

$$\sum_{i=0}^{k} \binom{k}{i} (-x)^{k-i} D^k(x^i q) = k!q.$$

Using this identity, the self-tester can perform an `Inductive Test`. The robustness of the `Inductive Test` can be established as in the proof of Theorem 2.1. For completeness, we outline the key step here. Let c_i denote the coefficient of the term $D^k(x^i q)$ in the sum in Lemma 2.4. Thus $c_i = \binom{k}{i}(-x)^{k-i}$.

LEMMA 2.5. *If g is a linear function that satisfies* $\Pr_{q\in\mathcal{P}_n}[\sum_i c_i g(x^i q) \neq k!q] = \epsilon < 1/2$, *then* $\Pr_{q\in\mathcal{P}_n}[\sum_i c_i g(x^i q) = k!q] = 1$.

Proof.

$$\begin{aligned}\Pr_{r\in\mathcal{P}_n}\Big[\sum_i c_i g(x^i q) &= \sum_i c_i g(x^i(r+q-r)) \\ &= \sum_i c_i g(x^i r) + \sum_i c_i g(x^i(q-r)) \\ &= k!r + k!(q-r) \\ &= k!q\Big] \geq 1-2\epsilon > 0.\end{aligned}$$

Here the first equality is rewriting; the second equality holds with probability $1-2\epsilon$ by the assumption that $\Pr_q[\sum_i c_i g(x^i q) \neq k!q] = \epsilon$. Since the event $\sum_i c_i g(x^i q) = k!q$ holds with positive probability and is independent of r, it holds with probability one. ∎

Thus, testing if g satisfies this identity for most q suffices to ensure that g satisfies this identity everywhere. If g does satisfy this identity, then we know the following: if $g(q) = D^k(q), g(xq) = D^k(xq), g(x^2q) = D^k(x^2q), \ldots, g(x^{k-1}q) = D^k(x^{k-1}q)$, then $g(x^kq) = D^k(x^kq)$. To conclude that $g \equiv D^k$ by induction, we need to modify `Basis Test` to test k base cases: if $g(1) = D^k(1), g(x) = D^k(x), g(x^2) = D^k(x^2), \ldots, g(x^{k-1}) = D^k(x^{k-1})$, i.e., if $g(1) = g(x) = g(x^2) = \cdots = g(x^{k-1}) = 0$.

Mod function. Let $\alpha \in \mathbb{K}[x]$ be a monic irreducible polynomial. Let $M_\alpha(q)$ denote the mod function with respect to α, that is, $M_\alpha(q) = q \bmod \alpha$. This is a linear function when the addition is interpreted as mod α addition. Since α is monic, the degree of $M_\alpha(q)$ is always less than $\deg\alpha$. If $c \in \mathbb{K}$ is the coefficient of the highest degree term in $M_\alpha(q)$, we have

$$M_\alpha(xq) = \begin{cases} xM_\alpha(q) & \text{if } \deg xM_\alpha(q) < \deg\alpha, \\ xM_\alpha(q) - c\alpha & \text{if } \deg xM_\alpha(q) = \deg\alpha. \end{cases}$$

As before, in testing if a program P computes the function M_α, step (3) of the self-tester is to choose many q's at random, compute $P_{\mathrm{sc}}(q)$ and $P_{\mathrm{sc}}(xq)$, and verify that one of the identities $P_{\mathrm{sc}}(xq) = xP_{\mathrm{sc}}(q)$ or $P_{\mathrm{sc}}(xq) = xP_{\mathrm{sc}}(q) - cq$ holds (depending on the degree of q).

3. Multilinear functions.

In this section we extend the ideas in section 2 to multilinear functions. A k-variate function f is called *k-linear* if it is linear in each of its variables when the other variables are fixed, i.e., $f(\alpha_1, \ldots, \alpha_{i-1}, \alpha_i + \beta_i, \alpha_{i+1}, \ldots, \alpha_k) = f(\alpha_1, \ldots, \alpha_{i-1}, \alpha_i, \alpha_{i+1}, \ldots, \alpha_k) + f(\alpha_1, \ldots, \alpha_{i-1}, \beta_i, \alpha_{i+1}, \ldots, \alpha_k)$ $\forall\ i = 1, \ldots, k$.

Our main motivating example for a multilinear function is polynomial multiplication $f : \mathcal{P}_{n-1} \times \mathcal{P}_{n-1} \to \mathcal{P}_{2n-1}$, which is bilinear. Note that the domain of f is generated by n^2 generators of the form (x^i, x^j) for $0 \leq i, j \leq n$ (i.e., pairs of generators of $\mathcal{P}_{n-1}$). Now, suppose we wish to test P that purports to compute f. The naive approach would require doing the `Generator Tests` at these n^2 generators. This requires $O(n^2)$ calls to P, rendering the self-tester highly inefficient. Blum, Luby, and Rubinfeld [5] give a more efficient bootstrap self-tester that makes $O(\log^{O(k)} n)$ calls to P. It can be seen that for general k-linear functions, their method can be extended to yield a tester that makes $O(\log^{O(k)} n)$ calls to P. (In our context, it is allowable to think of k as a constant since changing k results in an entirely different function f.) We are interested in reducing the number of calls to P with respect to the problem size n for a specific function f. The complexity of the tester we present here is independent of n, and the self-tester is required to know the correct value of f at only one point. As in the previous section, our result applies to many general multilinear functions over large vector spaces.

As before, we define a set of properties depending on f, that, if satisfied by P, would necessarily imply that P must be the same as the particular multilinear function f. For simplicity, we present the following theorem for f that is bilinear. This is an analog of Theorem 2.1 for multilinear functions.

THEOREM 3.1. *Suppose f is a bilinear function from V^2 into a ring R, and suppose P is a program for f.*

(a) *Let $\epsilon < 1/4$, and suppose P satisfies the following condition*:

(1) $\Pr_{\alpha_1,\alpha_2,\beta_1,\beta_2 \in V}[P(\alpha_1+\beta_1,\alpha_2+\beta_2) \neq P(\alpha_1,\alpha_2)+P(\alpha_1,\beta_2)+P(\beta_1,\alpha_2)+P(\beta_1,\beta_2)] \leq \epsilon$.

Then the function g defined by $g(\alpha_1,\alpha_2) = \text{majority}_{\beta_1,\beta_2\in V}\{P(\alpha_1-\beta_1,\alpha_2-\beta_2)+P(\alpha_1-\beta_1,\beta_2)+P(\beta_1,\alpha_2-\beta_2)+P(\beta_1,\beta_2)\}$ is a bilinear function on V^2, and g agrees with P on at least $1-2\epsilon$ fraction of the inputs.

(b) *Furthermore, suppose g satisfies the following conditions*:

(2) $g(e_1,e_1) = f(e_1,e_1)$,

(3)

$$\Pr_{\alpha_1,\alpha_2\in V}[g(\theta(\alpha_1),\alpha_2) \neq h_\theta^{(1)}(\alpha_1, g(\alpha_1,\alpha_2))] \leq \epsilon,$$

$$\Pr_{\alpha_1,\alpha_2\in V}[g(\alpha_1,\theta(\alpha_2)) \neq h_\theta^{(2)}(\alpha_2, g(\alpha_1,\alpha_2))] \leq \epsilon.$$

Then $g(\alpha_1,\alpha_2) = f(\alpha_1,\alpha_2)\ \forall\ \alpha_1,\alpha_2 \in V$.

Proof. A simple extension of the proof in [5] shows that g is bilinear. (Better bounds on ϵ via a different test can be obtained by appealing to [2].) As in the proof of Theorem 2.1, it suffices to show that given the three conditions, a stronger version of condition (3) holds: $g(\theta(\alpha_1),\alpha_2) = h_\theta^{(1)}(\alpha_1, g(\alpha_1,\alpha_2))$ and $g(\alpha_1,\theta(\alpha_2)) = h_\theta^{(2)}(\alpha_2, g(\alpha_1,\alpha_2))$ for every $\alpha_1,\alpha_2 \in V$. With the addition of this last property, it can be shown that $g \equiv f$. Taking condition (2) that $g(e_1,e_1) = f(e_1,e_1)$ as the base case and inducting by obtaining (e_{i+1},e_i) and (e_i,e_{i+1}) from (e_i,e_i) via an application of θ to either generator $\forall\ 1 \leq i < n$, it can be shown that $g(e_i,e_j) = f(e_i,e_j)$ for any bases elements e_i, e_j of V. This, combined with the bilinearity property of g, implies the correctness of g on every input.

Now we proceed to show the required intermediate result that given conditions (1) and (2), g satisfies the stronger version of condition (3) that we require above:

$$\begin{aligned}
\Pr_{\beta_1,\beta_2\in V^2}\big[&g(\alpha_1,\theta(\alpha_2))\\
&= g(\beta_1+\alpha_1-\beta_1, \theta(\beta_2+\alpha_2-\beta_2))\\
&= g(\beta_1,\theta(\beta_2+\alpha_2-\beta_2)) + g(\alpha_1-\beta_1,\theta(\beta_2+\alpha_2-\beta_2))\\
&= g(\beta_1,\theta(\beta_2)) + g(\beta_1,\theta(\alpha_2-\beta_2)) + g(\alpha_1-\beta_1,\theta(\beta_2)) + g(\alpha_1-\beta_1,\theta(\alpha_2-\beta_2))\\
&= h_\theta^{(2)}(\beta_2, g(\beta_1,\beta_2)) + h_\theta^{(2)}(\alpha_2-\beta_2, g(\beta_1,\alpha_2-\beta_2))\\
&\quad + h_\theta^{(2)}(\beta_2, g(\alpha_1-\beta_1,\beta_2)) + h_\theta^{(2)}(\alpha_2-\beta_2, g(\alpha_1-\beta_1,\alpha_2-\beta_2))\\
&= h_\theta^{(2)}(\alpha_2, g(\beta_1,\alpha_2)) + h_\theta^{(2)}(\alpha_2, g(\alpha_1-\beta_1,\alpha_2))\\
&= h_\theta^{(2)}(\alpha_2, g(\alpha_1,\alpha_2))\big] \geq 1-4\epsilon > 0.
\end{aligned}$$

The first equality is a rewriting of terms. Multilinearity of g implies the second and third equalities. If the probability $\Pr[g(\alpha_1, \theta(\alpha_2)) \neq h_\theta^{(2)}(\alpha_2, g(\alpha_1, \alpha_2))] < \epsilon$, then the fourth equality fails with probability less than 4ϵ. The rest of the equality follows from the multilinearity of $h_\theta^{(2)}$ and g. If $\epsilon < 1/4$, this probability is nonzero. Since the first and last terms are independent of β_1, β_2, and are equal with nonzero probability, the result follows. A similar approach works for $h_\theta^{(1)}$ as well. ∎

```
Multilinearity Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $\alpha_1, \alpha_2, \beta_1, \beta_2 \in_R V$
    Verify $P(\alpha_1+\beta_1, \alpha_2+\beta_2) = P(\alpha_1, \alpha_2) + P(\beta_1, \alpha_2) +$
          $P(\alpha_1, \beta_2) + P(\beta_1, \beta_2)$
    Reject if the test fails

Basis Test:
  Verify $P_{\text{sc}}(e_1, e_1) = f(e_1, e_1)$

Inductive Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $\alpha_1, \alpha_2 \in_R V$
    Verify $P_{\text{sc}}(\theta(\alpha_1), \alpha_2) = h_\theta^{(1)}(\alpha_1, P_{\text{sc}}(\alpha_1, \alpha_2))$
    Verify $P_{\text{sc}}(\alpha_1, \theta(\alpha_2)) = h_\theta^{(2)}(\alpha_2, P_{\text{sc}}(\alpha_1, \alpha_2))$
    Reject if the test fails
```

Note that in the latter two tests we use a self-corrected version P_{sc} of P. The notion of self-correctors for multilinear functions over vector spaces is implied by random self-reducibility.

It is easy to see that Theorem 3.1 extends to an arbitrary k-linear function so long as $\epsilon < 1/2^k$. Thus, we obtain the following theorem whose proof mirrors that of Theorem 2.2.

THEOREM 3.2. *If f is a k-variate linear function, then for any $\rho < 1$ and $\epsilon < 1/2^k$, the above three tests comprise a $2^k\epsilon$-self-tester for f that succeeds with probability at least $1-\rho$.*

3.1. Applications. Let q_1, q_2 denote polynomials in x. The function $M(q_1, q_2)$ that multiplies two polynomials is symmetric and linear in each variable. Moreover, since $M(xq_1, q_2) = M(q_1, xq_2) = xM(q_1, q_2)$, polynomial multiplication has an efficient self-tester.

An interesting application of polynomial multiplication, together with the mod function described in section 2.1, is the following. It is well known that a degree n (finite) extension $\mathbb{K}$ of a finite field $\mathbb{F}$ is isomorphic to the field $\mathbb{F}[x]/(\alpha)$, where α is an irreducible polynomial of degree n over $\mathbb{F}$. Under this isomorphism, each element of $\mathbb{K}$ is viewed as a polynomial of degree $\leq n$ over $\mathbb{F}$; addition of two elements $q_1, q_2 \in \mathbb{K}$ is just their sum $q_1 + q_2$ as polynomials; and multiplication of $q_1, q_2 \in \mathbb{K}$ is given by $q_1 q_2 \bmod \alpha$. It follows that field arithmetic (addition and multiplication) in finite extensions of a finite field can be self-tested without the generator bottleneck, that is, the number of calls made to the program being tested is independent of the degree of the field extension.

3.2. Matrix multiplication. Let $\mathcal{M}_n$ denote the algebra of $n \times n$ matrices over $\mathbb{Z}_p$, and let $f : \mathcal{M}_n \times \mathcal{M}_n \to \mathcal{M}_n$ denote matrix multiplication. Matrix multiplication is a bilinear function; however, since it is a matrix operation rather than a vector operation, it requires a slightly different treatment from the general multilinear functions. $\mathcal{M}_n$, viewed as an additive group, has n^2 generators; one possible set of generators is $\{E_{i,j} \mid 1 \leq i, j \leq n\}$, where each generator $E_{i,j}$ is a matrix that has a 1 in position (i, j) and 0's elsewhere. We note that any generator $E_{i,j}$ can be converted into any other generator $E_{k,\ell}$ via a sequence of horizontal and vertical rotations obtained by multiplications by the special *permutation matrix* Π:

$$\Pi = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \dots & 0 \end{bmatrix} \Rightarrow \; E_{i,j} = \Pi^{k-i} \cdot E_{k,\ell} \cdot \Pi^{\ell-j}.$$

The rotation operations correspond to the θ operator of the model for multilinear functions. There are, however, two different kinds of rotations—horizontal and vertical—due to the two-dimensional nature of the input, and the function h defining the behavior of the function with respect to these rotations is not always easily computable, short of actually performing a matrix multiplication. We therefore exploit some additional properties of the problem to come up with a set of conditions that are sufficient for P to be computing matrix multiplication f. Let $\mathcal{M}'_n$ denote the subgroup of $\mathcal{M}_n$ that contains only matrices with columns $2, \dots, n$ all-zero.

THEOREM 3.3. *Let P be a program for f and $\epsilon < 1/8$.*

(a) *Suppose P satisfies the following*:

(1) $\Pr_{X,Y,Z,W \in \mathcal{M}_n}[P(X+Z, Y+W) \neq P(X,Y)+P(X,W)+P(Z,Y)+P(Z,W)] \leq \epsilon$.

Then the function g defined by $g(X,Y) = \text{majority}_{Z,W \in \mathcal{M}_n}\{P(X-Z, Y-W)+P(X-Z, W)+P(Z, Y-W)+P(Z,W)\}$ is a bilinear function on $\mathcal{M}_n^2$, and g agrees with P on at least $1-2\epsilon$ fraction of the inputs.

(b) *Furthermore, suppose g satisfies the following conditions*:

(2) $\Pr_{X \in \mathcal{M}'_n, Y \in \mathcal{M}_n}[g(X,Y) \neq f(X,Y)] \leq \epsilon$,

(3) $\Pr_{X \in \mathcal{M}_n}[g(X,\Pi) \neq f(X,\Pi)] \leq \epsilon$ *and* $\Pr_{X \in \mathcal{M}_n}[g(\Pi, X) \neq f(\Pi, X)] \leq \epsilon$,

(4) $\Pr_{X,Y,Z \in \mathcal{M}_n}[g(X, g(Y,Z)) \neq g(g(X,Y),Z)] \leq \epsilon$.

Then $g(X,Y) = f(X,Y)\ \forall\ X, Y \in \mathcal{M}_n$.

To be able to prove this theorem, we first need to show that the conditions recounted have stronger implications than their statements. Then we will show that these strengthened versions of the conditions imply Theorem 3.3.

First, we show that condition (2) implies a stronger version of itself.

LEMMA 3.4. *If condition* (2) *in Theorem* 3.3 *holds, then $g(E_{i,1}, Y) = f(E_{i,1}, Y)$ $\forall\ Y \in \mathcal{M}_n$ and $1 \leq i \leq n$.*

Proof. We in fact show something stronger. We show that $g(X,Y) = f(X,Y)\ \forall\ X \in \mathcal{M}'_n$ and $Y \in \mathcal{M}_n$:

$$\begin{aligned}\Pr_{Z \in \mathcal{M}'_n, W \in \mathcal{M}_n}\big[g(X,Y) &= g(Z, Y-W) + g(X-Z, Y-W) + g(Z,W) + g(X-Z,W) \\ &= f(Z, Y-W) + f(X-Z, Y-W) + f(Z,W) + f(X-Z,W) \\ &= f(X,Y)\big] > 1-4\epsilon > 0.\end{aligned}$$

The second equality holds with probability $\geq 1-4\epsilon$ by condition (2). All the rest hold from linearity of g and f. The result follows since $\epsilon < 1/4$. The lemma follows since $E_{i,1} \in \mathcal{M}'_n$. ∎

An immediate adaptation of the proof of Lemma 3.4 can be used to extend condition (3) to hold for all inputs.

Next, we show that the linearity of g makes it possible to conclude from hypothesis (4) that g is associative.

LEMMA 3.5. *If condition* (4) *in Theorem* 3.3 *holds, then g is always associative.*

Proof.

$$\begin{aligned}
&\Pr_{X_1,Y_1,Z_1\in\mathcal{M}_n}\Big[g(g(X,Y),Z)\\
&= g(g(X_1,Y_1),Z_1) + g(g(X-X_1,Y_1),Z_1) + g(g(X_1,Y-Y_1),Z_1)\\
&\quad + g(g(X-X_1,Y-Y_1),Z_1) + g(g(X_1,Y_1),Z-Z_1) + g(g(X-X_1,Y_1),Z-Z_1)\\
&\quad + g(g(X_1,Y-Y_1),Z-Z_1) + g(g(X-X_1,Y-Y_1),Z-Z_1)\\
&= g(X_1,g(Y_1,Z_1)) + g(X-X_1,g(Y_1,Z_1)) + g(X_1,g(Y-Y_1,Z_1))\\
&\quad + g(X-X_1,g(Y-Y_1,Z_1)) + g(X_1,g(Y_1,Z-Z_1)) + g(X-X_1,g(Y_1,Z-Z_1))\\
&\quad + g(X_1,g(Y-Y_1,Z-Z_1)) + g(X-X_1,g(Y-Y_1,Z-Z_1))\\
&= g(X,g(Y,Z))\Big] \geq 1-8\epsilon > 0.
\end{aligned}$$

The first equality holds from the linearity of g after expanding X, Y, Z as $X_1 + (X - X_1), Y_1 + (Y - Y_1), Z_1 + (Z - Z_1)$, respectively. The second equality is true by the condition (4) with probability $\geq 1 - 8\epsilon$. The last equality is a recombination of terms using linearity. ∎

We now have the tools to prove Theorem 3.3.

Proof. The bilinearity of g follows from the proof of Theorem 3.1.

From Lemmas 3.4 and 3.5, we have that condition (2) can be extended such that $g(E_{i,1}, X) = f(E_{i,1}, X)\ \forall\ 1 \leq i \leq n$ and $X \in \mathcal{M}_n$, and conditions (3) and (4) on g hold for all inputs.

To show that these properties are sufficient to identify g as matrix multiplication, note that from the multilinearity of g, we can write

$$g(X,Y) = g\left(\sum_{1\leq i,j\leq n} x_{i,j}\cdot E_{i,j},\ \sum_{1\leq k,\ell\leq n} y_{k,\ell}\cdot E_{k,\ell}\right) = \sum_{1\leq i,j,k,\ell\leq n} x_{i,j}\cdot y_{k,\ell}\cdot g(E_{i,j},E_{k,\ell}).$$

If $g(E_{i,j}, E_{k,\ell}) = f(E_{i,j}, E_{k,\ell})\ \forall\ i, j, k, \ell$, then multilinearity implies that g is the same as f. Now, using our assumptions, we proceed to show the former holds:

$$\begin{aligned}
g(E_{i,j},E_{k,\ell}) &= g(f(E_{i,1},\Pi^{j-1}), f(\Pi^{k-1},E_{1,\ell}))\\
&= g(g(E_{i,1},\Pi^{j-1}), g(\Pi^{k-1},E_{1,\ell}))\\
&= g(E_{i,1}, g(\Pi^{j+k-2}), E_{1,\ell})\\
&= g(E_{i,1}, E_{k+j-2,\ell})\\
&= f(E_{i,1}, E_{k+j-2,\ell})\\
&= f(E_{i,j}, E_{k,\ell}).
\end{aligned}$$

The first equality is just a rewriting of the two generators in terms of other generators. The second one follows from the strengthening of condition (3) that g computes f whenever one of its arguments is equal to a power of Π. The third one follows from the associativity of g, and the fourth one holds because g is the same as f when the first input is a power of Π and by rewriting $E_{k+j-2,\ell}$. The fifth equality is true because g

computes f correctly when its first argument is $E_{i,1}$ (as the consequence of condition (2), see Lemma 3.4). The last one is a rewriting of the previous equality, using the associativity of multiplication. Therefore, g is the same function as f. □

We now present the test for associativity:

```
Associativity Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $X, Y, Z \in_R \mathcal{M}_n$
    Verify $P_{\rm sc}(X, P_{\rm sc}(Y, Z)) = P_{\rm sc}(P_{\rm sc}(X, Y), Z)$
    Reject if the test fails
```

A self-tester can be built by testing conditions (1), (2), (3), and (4), which correspond to the `Property Test`, the `Basis Test`, the `Inductive Test`, and the `Associativity Test`, respectively. Note that testing conditions (2) and (3) involve knowing the value of f at random inputs. These inputs, however, come from a restricted subspace which makes it possible to compute f both easily and efficiently. The following theorem is immediate.

THEOREM 3.6. *For any $\rho < 1$ and $\epsilon < 1/8$, there is an ϵ-self-tester for matrix multiplication that succeeds with probability at least $1 - \rho$.*

4. Nonlinear functions.

In this section, we consider nonlinear functions. Specifically, we deal with exponentiation and constant degree polynomials in the ring of polynomials over the finite fields $\mathbb{Z}_p$. It is obvious that exponentiation and constant degree polynomials are clearly defined over this ring.

4.1. Constant degree exponentiation.

We first consider the function $f(q) = q^d$ for some d (that is, raising a polynomial to the dth power). Suppose a program P claims to perform this exponentiation for all degree n polynomials $q \in \mathcal{P}_n \subseteq \mathbb{K}[x]$. Using the low-degree test of Rubinfeld and Sudan [20] (see also [14]) we can first test if the function computed by P is close to some degree d polynomial g. As before, using the self-corrected version $P_{\rm sc}$ of P, we can also verify that $g(e_1) = f(e_1)$.

The induction identity $f(xq) = x^d f(q)$ also applies, and one can test whether P satisfies this property on most inputs. Now it remains to show that this implies $g(xq) = x^d g(q)\ \forall\ q \in \mathcal{P}_n$. We follow a strategy similar to the case of linear functions, this time using the Lagrange interpolation formula as the robust property that identifies a degree d polynomial. We note that this idea is similar to the use of the interpolation formula by Gemmell et al. [14], which extends the [5] result from linear functions to low-degree polynomials. Before proceeding with the proof, we state the following fact concerning the Lagrange interpolation identity.

FACT 4.1. *Let g be a degree d polynomial. For any $q \in \mathcal{P}_n$, if $q_1, q_2, \ldots, q_{d+1}$ are distinct elements of $\mathcal{P}_n$,*

$$g(q) = \sum_{i=1}^{d+1} g(q_i) \prod_{j\neq i} \frac{q - q_j}{q_i - q_j} \qquad \textit{and} \qquad g(xq) = \sum_{i=1}^{d+1} g(xq_i) \prod_{j\neq i} \frac{q - q_j}{q_i - q_j}.$$

The self-tester for $f(q) = q^d$ comprises the following tests:

```
Degree Test:
  Verify $P$ is close to a degree $d$ polynomial $g$ (low-degree test)
  Reject if the test fails

Basis Test:
  Verify $P_{\rm sc}(e_1) = f(e_1)$

Inductive Test:
  Repeat $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ times
    Pick $\alpha \in_R V$
    Verify $P_{\rm sc}(\theta(\alpha)) = x^d P_{\rm sc}(\alpha)$
    Reject if the test fails
```

Let β denote the probability that the $d+1$ random choices from the domain produce distinct elements. We will assume that the domain is large enough so that β is close to 1.

Assume that P passes the `Degree Test` and $P_{\rm sc}$ passes the `Basis Test`, that is, P agrees with some degree-d polynomial g on most inputs. We note that the low-degree of test of [20] makes $O((1/\epsilon)\log(1/\rho))$ calls to render a decision with confidence $1-(\rho/3)$. Furthermore, $P_{\rm sc}$ also requires only $O((1/\epsilon)\log(1/\rho))$ calls to compute g correctly with probability $1-(\rho/3)$. Below we sketch the proof that if $\epsilon < \beta/(d+1)$ and $P_{\rm sc}$ passes the `Inductive Test`, then g satisfies $g(xq) = x^d g(q)$ everywhere. Note that $(1/\epsilon) = \Theta(d)$, so the time taken by the tester is only $\Theta(d)$ (when ρ is a constant).

$$\begin{aligned}\Pr_{q_1,\dots,q_{d+1}}[g(xq) &= \sum_{i=1}^{d+1} a_i g(xq_i)\\ &= \sum_{i=1}^{d+1} a_i x^d g(q_i)\\ &= x^d \sum_{i=1}^{d+1} a_i g(q_i)\\ &= x^d g(q)] \geq \beta - (d+1)\epsilon > 0.\end{aligned}$$

Here $a_i = \prod_{j\neq i}(q-q_j)/(q_i-q_j)$. The first equality is Fact 4.1, and applies since g has been verified to be a degree d polynomial. Since the q_i's are uniformly and identically distributed, by `Inductive Test` the second equality fails with probability $< (d+1)\epsilon$. The third equality is just rewriting, and the fourth equality is due to Fact 4.1 (the interpolation identity), which can be applied so long as the q_i's are distinct, an event that occurs with probability β. Since the equality $g(xq) = x^d g(q)$ holds independent of q_i's, if $\epsilon < \beta/(d+1)$, it holds with probability 1.

THEOREM 4.2. *The function $f(q) = q^d$ (where $q \in \mathcal{P}_n$) has an $O(1/d)$-self-tester that makes $O(d)$ queries.*

4.2. Constant degree polynomials. Next we consider extending the result of section 4.1 to arbitrary degree-d polynomials $f : \mathcal{P}_n \to \mathcal{P}_{nd}$. Clearly the low-degree test and the basis test work as before. The interpolation identity is also valid. The missing ingredient is the availability of an identity like "$f(xq) = x^d f(q)$," which, as we have shown above, is a robust property that can be efficiently tested. We show how to get around this difficulty; this idea is based on a suggestion by R. Rubinfeld [18].

Suppose $f : \mathcal{P}_n \to \mathcal{P}_{nd}$ is a degree-d polynomial (e.g., $f(q) = q^2 + q + 1$), and suppose a program P purports to compute f. Our strategy is to design a self-corrector R for P and to then estimate the fraction of inputs q such that $P(q) \neq R(q)$. The difficulty in implementing this idea by directly using the random self-reducibility of f is that the usefulness of the self-corrector (to compute f correctly on every input) depends critically on our ability to certify that P is correct on most inputs. Since checking whether P is correct on most inputs is precisely the task of self-testing, we seem to be going in cycles.

To circumvent this problem, we will design an intermediate multiplication program Q that uses P as an oracle. To design the program Q, we prove the following technical lemma that helps us express d-ary multiplication in terms of f—that is, we establish a reduction from the multilinear function $\prod_{i=1}^{d} q_i$ to the nonlinear function f. This reduction is a generalization to degree d of the elementary polarization identity $xy = ((x+y)^2 - (x-y)^2)/4$, but slightly stronger in that it works for arbitrary polynomials of degree d, not just degree-d exponentiation.

LEMMA 4.3. *For $x \in \{0,1\}^d$, let x_i denote the ith bit of x. For any polynomial $G(q) = \sum_i p_i q^i$ of degree d,*

$$\sum_{x \in \{0,1\}^d} \left(\prod_{i=1}^{d} (-1)^{x_i} \right) G\left(\sum_{i=1}^{d} (-1)^{x_i} q_i \right) = p_d 2^d d! \prod_{i=1}^{d} q_i.$$

Using the reduction given by Lemma 4.3, we will show how to construct an ϵ-self-tester T for f (for $\epsilon = \Theta(2^{-d})$), following the outline sketched.

(1) First we build a program Q that performs c-ary multiplication for any $c \leq d$ (if $c < d$, we can simply multiply by extra 1's). The program Q is then self-tested efficiently (without the generator bottleneck) by using a $(1/2^{d+1})$-self-tester for the d-variate multilinear multiplication function from section 3. The number of queries made to P in this process is $O(1)$, where the constant depends on d (the degree of f) but not on n (the dimensionality of the domain of f). Thus, if Q passes this self-testing step, then it computes multiplication correctly on all but $1/2^{d+1}$ fraction of the inputs. If Q fails the self-testing process, the self-tester T rejects.

(2) Next we build a reliable program Q_{sc} that self-corrects Q using the random self-reducibility of the multilinear multiplication function. That is, Q_{sc} can be used to compute c-ary multiplication (for any $c \leq d$) correctly for every input with probability at least $1 - \rho$ for any constant $\rho > 0$. In particular, by making $O(2^d \log d)$ calls to Q (and, hence, $O(2^{2d} \log d)$ calls to P), Q_{sc} can be used to compute multiplication correctly for every input with probability at least $1 - (1/10d)$.

(3) Next, we use Q_{sc} to build the program R that computes $f(q)$ in a straightforward way by using Q_{sc} to compute the d required multiplications. If Q_{sc} computes multiplication correctly with probability $1 - (1/10d)$, then R computes f correctly for any input with probability at least 0.9.

(4) Finally, T randomly picks $N = O((1/\epsilon)\log(1/\rho))$ many samples q and checks if $P(q) = R(q)$, and outputs PASS iff $P(q) = R(q)$ for all the chosen values of q.

It is easy to see that if P computes f correctly on all inputs, the self-tester T will output PASS with probability one. For the converse, suppose that $\delta =_{\mathrm{def}} \Pr_q[P(q) \neq f(q)] > \epsilon$, and yet T outputs PASS. We will upper bound the probability of this event by ρ.

Since Q passes the self-testing step (step (1)), it computes multiplication correctly on all but $1/2^{d+1}$ fraction of the inputs, and therefore, the use of the self-corrector

$Q_{\rm sc}$ as described in step (2) is justified. This, in turn, implies the guarantee made of R in step (3): for every input q, $R(q) = f(q)$ with probability at least 0.9. In step (4), the probability that $P(q) \neq R(q)$ for a single random q is at least $(0.9)\delta$. The probability that $P(q) = R(q)$ for every random input q chosen in step (4) is therefore at most $(1-(0.9)\delta)^N = (1-(0.9)\delta)^{O((1/\epsilon)\log(1/\rho))} < (1-(0.9)\delta)^{O((1/\delta)\log(1/\rho))} < \rho$. Thus the probability that T outputs PASS, given that $\Pr_q[P(q) \neq f(q)] > \epsilon$, is at most ρ. The following theorem is proven, modulo the proof of Lemma 4.3.

THEOREM 4.4. *The function $f(q)$, where f is a polynomial in $q \in \mathcal{P}_n$ of degree d, has an $O(1/2^d)$-self-tester that makes $O(2^d)$ queries.*

Even though our self-tester makes $O(2^d)$ queries to test degree-d exponentiation, the number of queries is independent of n, the dimensionality of the domain. Thus, our self-tester is attractive if n is large and d is small. In particular, in conjunction with the testers for finite field arithmetic described in section 3.1, the self-testers described here help us to efficiently self-test constant degree polynomials on finite field extensions of large dimension.

It remains to prove Lemma 4.3. This lemma is a direct corollary of the following lemma, which illustrates a method to express d-ary multiplication in terms of f. The proof of the next lemma is given in the Appendix.

LEMMA 4.5. *Let $p_1, \ldots, p_d$ be distinct variables. For $x \in \{0,1\}^d$, let x_i denote the ith bit of x. Then*

$$\sum_{x\in\{0,1\}^d} \left(\prod_{i=1}^{d} (-1)^{x_i}\right) \left(\sum_i (-1)^{x_i} p_i\right)^c = \begin{cases} 0 & \text{if } c < d, \\ 2^d d! \prod_{i=1}^d p_i & \text{if } c = d. \end{cases}$$

Appendix. Proof of Lemmas 2.4 and 4.5.

LEMMA 2.4. *If q is a polynomial in x of degree $\geq k$, then*

$$\sum_{i=0}^{k} \binom{k}{i} (-x)^{k-i} D^k(x^i q) = k!q.$$

Proof. The proof is by induction on k. The base case $(k=0)$ is obviously true. Let $k > 0$. We have $\mathcal{S}_k = \sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^{k+1}(x^i q)$. Since $D^{k+1}(x^i q) = D^k(ix^{i-1}q + x^i D(q))$ and since differentiation is linear, we have

$$\mathcal{S}_k = \left[\sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^k(ix^{i-1}q)\right] + \left[\sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^k(x^i D(q))\right].$$

Since the first term $(i=0)$ in the first sum vanishes, and since $i\binom{k+1}{i} = (k+1)\binom{k}{i-1}$, the first sum evaluates to $(k+1)\sum_{i=1}^{k+1} \binom{k}{i-1} (-x)^{k+1-i} D^k(x^{i-1}q)$, which equals $(k+1)!q$ by the inductive hypothesis. Hence it suffices to show that the second sum evaluates to 0. This summation can be written as $\sum_{i=0}^{k} \binom{k+1}{i} (-x)^{k+1-i} D^k(x^i D(q))$ $+D^k(x^{k+1}D(q))$. By Pascal's identity, this sum can be split into the terms $\sum_{i=0}^{k} \binom{k}{i-1} (-x)^{k+1-i} D^k(x^i D(q))$, $\sum_{i=0}^{k} \binom{k}{i} (-x)^{k+1-i} D^k(x^i D(q))$, and $D^k(x^k \cdot x D(q))$. The second term can be seen to be $(-x)k!D(q)$ using the induction hypothesis. The first and third terms can be combined to obtain $k!(xD(q))$, again using the induction hypothesis. Thus, the entire expression equals $(-x)k!D(q) + k!(xD(q)) = 0$. ∎

LEMMA 4.5. *Let $p_1, \dots, p_d$ be distinct variables. For $x \in \{0,1\}^d$, let x_i denote the ith bit of x. Then*

$$\sum_{x\in\{0,1\}^d}\left(\prod_{i=1}^d(-1)^{x_i}\right)\left(\sum_i(-1)^{x_i}p_i\right)^c = \begin{cases} 0 & \text{if } c<d,\\ 2^d d!\prod_{i=1}^d p_i & \text{if } c=d.\end{cases}$$

Proof. The proof uses the Fourier transform on the Boolean cube $\{0,1\}^d$ (using the standard isomorphism between $\{0,1\}^d$ and $\mathbb{Z}_2^d$). Let $\mathcal{F}$ denote the space of functions from $\mathbb{Z}_2^d$ into $\mathbb{C}$. $\mathcal{F}$ is a (finite) vector space of functions of dimension 2^d. Define the inner product between functions $f, g \in \mathcal{F}$ by $\langle f,g\rangle = 2^{-d}\sum_x f(x)g(x)$. For $\alpha \in \mathbb{Z}_2^d$, define the function $\chi_\alpha : \mathbb{Z}_2^d \to \mathbb{C}$ by $\chi_\alpha(x) = \prod_{i=1}^d(-1)^{x_i\alpha_i}$, where x_i and α_i denote the ith bits, respectively, of x and α. It is easy to check that $\chi_\alpha(x)\chi_\alpha(y) = \chi_\alpha(x+y)$, whence every χ_α is a character of $\mathbb{Z}_2^d$. Furthermore, it is easy to check that $\langle\chi_\alpha,\chi_\beta\rangle$ equals 0 if $\alpha \neq \beta$, and equals 1 if $\alpha = \beta$. Therefore, the characters χ_α form an orthonormal basis of $\mathcal{F}$, and every function $f : \mathbb{Z}_2^d \to \mathbb{C}$ has a unique expansion in this basis as $f = \sum_\alpha \widehat{f}_\alpha\chi_\alpha$. This is called the Fourier transform of f, and the coefficients $\widehat{f}_\alpha$ are called the Fourier coefficients of f; by the orthonormality of the basis, $\widehat{f}_\alpha = \langle\chi_\alpha, f\rangle$. An easy property of the Fourier transform is that for $\alpha \neq 0^d$, $\sum_x \chi_\alpha(x) = 0$ (in fact, this is true for any nontrivial character of any group).

For the proof of the lemma, we note that it suffices to prove the lemma for all complex numbers p_i. Fix a list of complex numbers $p_1, \dots, p_d$, and define the function $f : \mathbb{Z}_2^d \to \mathbb{C}$ by $f(x) = \left(\sum_i(-1)^{x_i}p_i\right)^c$. Then the left-hand side of the statement of the lemma is just $2^d\widehat{f}_\alpha$, where $\alpha = 1^d$. Thus,

$$\begin{aligned}
&\sum_{x\in\mathbb{Z}_2^d}\left(\prod_{i=1}^d(-1)^{x_i}\right)\left(\sum_i(-1)^{x_i}p_i\right)^c\\
&= 2^d\widehat{f}_\alpha\\
&= \sum_x \chi_\alpha(x)f(x)\\
&= \sum_x \chi_\alpha(x) \sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\dots,n_d\le c}} \binom{c}{n_1,\dots,n_d}\prod_{i=1}^d(-1)^{x_in_i}p_i^{n_i}\\
&= \sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\dots,n_d\le c}} \binom{c}{n_1,\dots,n_d}\prod_{i=1}^d p_i^{n_i}\sum_x\prod_{i=1}^d \chi_\alpha(x)\chi_\beta(x), \qquad \text{where } \beta_i = n_i \bmod 2\\
&= \sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\dots,n_d\le c}} \binom{c}{n_1,\dots,n_d}\prod_{i=1}^d p_i^{n_i}\sum_x\prod_{i=1}^d \chi_{\alpha+\beta}(x).
\end{aligned}$$

The innermost sum is zero if $\alpha+\beta \neq 0^d$, i.e., if $\alpha \neq \beta$, and equals 2^d otherwise. Thus $\beta_i = \alpha_i = 1$ for every i, that is, $n_i \equiv 1 (\text{mod } 2)$ for every i. If $c < d$, this is impossible, since $\sum_i n_i = c$. If $c = d$, then the only way this can happen is if $n_i = 1\ \forall\ i$; otherwise, for some i, $n_i > 1$, and since $\sum_i n_i = d$, some $n_i = 0$. When $n_i = 1\ \forall\ i$, it is easy to see that we have $2^d\widehat{f}_\alpha = 2^d d!\prod_{i=1}^d p_i$. $\square$

Acknowledgments. We are very grateful to Ronitt Rubinfeld for her valuable suggestions and guidance. We thank Manuel Blum and Mandar Mitra for useful discussions. We thank Dexter Kozen for his comments. We are grateful to the two anonymous referees for valuable comments that resulted in many improvements to our exposition. The idea of describing the proof of Lemma 4.5 using Fourier transforms also comes from one of the referees.

REFERENCES

[1] S. Ar, M. Blum, B. Codenotti, and P. Gemmell, *Checking approximate computations over the reals*, in Proceedings 25th Annual ACM Symposium on the Theory of Computing, 1993, pp. 786–795.

[2] L. Babai, L. Fortnow, and C. Lund, *Non-deterministic exponential time has two-prover interactive protocols*, Comput. Complexity, 1 (1991), pp. 3–40.

[3] D. Beaver and J. Feigenbaum, *Hiding instances in multioracle queries*, in Proceedings 7th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Sci. 415, Springer, New York, 1990, pp. 37–48.

[4] M. Blum and S. Kannan, *Designing programs that check their work*, in Proceedings 21st Annual ACM Symposium on the Theory of Computing, 1989, pp. 86–97.

[5] M. Blum, M. Luby, and R. Rubinfeld, *Self-testing/correcting with applications to numerical problems*, J. Comput. System Sci., 3 (1993), pp. 549–595.

[6] M. Blum and H. Wasserman, *Program result-checking: A theory of testing meets a test of theory*, in Proceedings 35th Annual IEEE Symposium on Foundations of Computer Science, 1994, pp. 382–392.

[7] M. Blum and H. Wasserman, *Reflections on the Pentium division bug*, IEEE Transactions on Computers, 4 (1996), pp. 385–393.

[8] E. Castillo and M. R. Ruiz-Cobo, *Functional Equations and Modeling in Science and Engineering*, Marcel Dekker, New York, 1992.

[9] R. Cleve and M. Luby, *A Note on Self-Testing/Correcting Methods for Trigonometric Functions*, Technical Report 90-032, ICSI, Berkeley, 1990.

[10] F. Ergün, *Testing multivariate linear functions: Overcoming the generator bottleneck*, in Proceedings 27th Annual ACM Symposium on the Theory of Computing, 1995, pp. 407–416.

[11] J. Feigenbaum, *Locally random reductions in interactive complexity theory*, in Advances in Computational Complexity Theory, DIMACS Ser. Discrete Math. Theoret. Comput. Sci., J.-Y. Cai, ed., AMS, 1993, Providence, RI, pp. 73–98.

[12] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy, *Approximating clique is almost NP-complete*, in Proceedings 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 2–12.

[13] R. Freivalds, *Fast probabilistic algorithms*, in Mathematical Foundations of Computer Science, Lecture Notes in Comput. Sci. 74, Springer, New York, 1979, pp. 57–69.

[14] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson, *Self-testing/ correcting for polynomials and for approximate functions*, in Proceedings 23rd Annual ACM Symposium on the Theory of Computing, 1991, pp. 32–42.

[15] R. Karp, M. Luby, and N. Madras, *Monte-Carlo approximation algorithms for enumeration problems*, J. Algorithms, 3 (1989), pp. 429–448.

[16] S. R. Kumar and D. Sivakumar, *On self-testing without the generator bottleneck*, in Proceedings 15th Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 1026, Springer, New York, 1995, pp. 248–262.

[17] R. Lipton, *New directions in testing*, in Proceedings DIMACS Workshop on Distributed Computing and Cryptography, 1991, pp. 191–202.

[18] R. Rubinfeld *private communication*.

[19] R. Rubinfeld and M. Sudan, *Testing polynomial functions efficiently and over rational domains*, in Proceedings 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1992, pp. 23–43.

[20] R. Rubinfeld and M. Sudan, *Robust characterizations of polynomials with applications to program testing*, SIAM J. Comput., 2 (1996), pp. 252–271.

[21] R. Rubinfeld, *A Mathematical Theory of Self-Checking, Self-Testing, and Self-Correcting Programs*, Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1990.

[22] R. Rubinfeld, *Robust functional equations with applications to self-testing/ correcting*, in Proceedings 35th Annual IEEE Symposium on Foundations of Computer Science, 1994, pp. 288–299.

[23] M. Sudan, *On the role of algebra in the efficient verification of proofs*, in Proceedings Workshop on Algebraic Methods in Complexity Theory, M. Agrawal, V. Arvind, and M. Mahajan, eds., 1994, pp. 58–68.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1652–1682

SAFE CONSTRAINT QUERIES*

MICHAEL BENEDIKT† AND LEONID LIBKIN‡

Abstract. We extend some of the classical characterization theorems of relational database theory—particularly those related to query safety—to the context where database elements come with fixed interpreted structure and where formulae over elements of that structure can be used in queries. We show that the addition of common interpreted functions, such as real addition and multiplication, to the relational calculus preserves important *characterization theorems* of the relational calculus and also preserves certain *combinatorial properties* of queries. Our main result of the first kind is that there is a syntactic characterization of the collection of safe queries over the relational calculus supplemented by a wide class of interpreted functions—a class that includes addition, multiplication, and exponentiation—and that this characterization gives us an interpreted analog of the concept of range-restricted query from the uninterpreted setting. Furthermore, our range-restricted queries are particularly intuitive for the relational calculus with real arithmetic and give a natural syntax for safe queries in the presence of polynomial functions. We use these characterizations to show that safety is decidable for Boolean combinations of conjunctive queries for a large class of interpreted structures. We show a dichotomy theorem that sets a polynomial bound on the growth of the output of a query that might refer to addition, multiplication, and exponentiation.

We apply the above results for finite databases to get results on constraint databases representing potentially infinite objects. We start by getting syntactic characterizations of the queries on constraint databases that preserve geometric conditions in the constraint data model. We consider classes of convex polytopes, polyhedra, and compact semilinear sets, the latter corresponding to many spatial applications. We show how to give an effective syntax to safe queries and prove that for conjunctive queries the preservation properties are decidable.

Key words. constraints, databases, first-order logic, query safety

AMS subject classifications. Primary, 68P15; Secondary, 03C07, 03C10, 03C13, 03C50, 52B11

PII. S0097539798342484

1. Introduction.

The power of classical query languages is linked to the fact that these queries express a restricted class of declarative programs. The class of semantic objects expressible through queries in the relational calculus, for example, is limited in a number of helpful ways: each such query is polynomial-time computable, each is local, and each has well-defined asymptotics. Although relational calculus queries may not return finite results, a natural subclass of the relational calculus does, namely, the class of *range-restricted queries*. This class gives guarantees of finite output and is complete in this respect: it captures all relational calculus queries whose outputs are always finite, the *safe* queries [1, 21, 39].

The relational theory on which these results are based deals only with *pure* relational queries, that is, those containing no interpreted predicates. Practical query languages, in contrast, contain interpreted functions such as $+$ and $*$. The resulting queries, then, make use of the domain semantics, rather than being independent of them, as pure relational queries are. For example, if the underlying

*Received by the editors July 22, 1998; accepted for publication (in revised form) August 10, 1999; published electronically March 17, 2000. Part of this work was done while visiting INRIA Rocquencourt, Le Chesnay Cedex, France.

http://www.siam.org/journals/sicomp/29-5/34248.html

†Bell Laboratories, 263 Shuman Boulevard, Naperville, IL 60566 (benedikt@research.bell-labs.com).

‡Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 (libkin@research.bell-labs.com).

structure is the field of real numbers $\langle \mathbb{R}, +, *, 0, 1, < \rangle$, the extension of relational calculus is achieved by using polynomial (in)equalities. For example, the query $\varphi(x, y) \equiv \exists z. R(x, z) \wedge R(z, y) \wedge x^2 + y^2 = z$ defines a subset of the self-join with the condition that in joinable tuples (x, z) and (z, y), z must be the sum of squares of x and y. A natural question, then, is what sort of restrictions still apply to queries given with interpreted structure.

Clearly, many standard results fail in the presence of interpreted structure; for example, queries may no longer express only local properties of inputs. Complexity bounds are often dependent on fragile properties of both the functions present and the encodings of the structures in some computational model. In contrast, we show here that certain kinds of *structural* properties of relational calculus queries remain when a reasonable interpreted structure is present. These include the classical equivalence of safe and range-restricted queries, decidability of safety for restricted classes of queries, as well as *combinatorial* properties of the queries, for example, restrictions on the growth rate of the result sets. A primary example of well-behaved combinatorics of these structures is a *growth dichotomy theorem*, which says that the output of a query is either polynomial in the database or infinite. We show that the well-behavedness of a structure, together with the decidability of its first-order theory, has *algorithmic* consequences; for example, the set of range-restricted formulae can be effectively computed.

A problem related to safety is that of *state-safety*, studied in [3]: for a query *and* a database, determine if the output is finite. Unlike the safety problem, which is undecidable (cf. [1]), the state-safety problem is decidable for some domains, for example, natural numbers with the order relation; see [3, 5]. However, there are interpreted structures (even having a decidable first-order theory) for which this problem is undecidable [36]. Moreover, [36] established that for the same interpreted structure, no recursive set of queries captures the class of safe queries; that is, it is impossible to have an analog of the concept of range-restriction. In contrast, we show that for many well-behaved structures, state-safety is decidable. Furthermore, safety over all states is decidable for restricted classes of queries (namely, Boolean combinations of conjunctive queries).

The above results are for the standard relational calculus with interpreted functions on finite structures; we then apply these results to get structural restrictions on the behavior of queries in other models, particularly the constraint database model introduced by Kanellakis, Kuper, and Revesz [20]. This model is motivated by new applications involving spatial and temporal data, which require storing and querying infinite collections. The constraint model generalizes the relational model by means of "generalized relations." These are possibly infinite sets defined by quantifier-free first-order formulae in the language of some underlying infinite structure $\mathcal{M} = \langle \mathcal{U}, \Omega \rangle$. Here $\mathcal{U}$ is a set (assumed to be infinite), and Ω is a signature that consists of a number of interpreted functions and predicates over $\mathcal{U}$. For example, in spatial applications, $\mathcal{M}$ is usually the real field $\langle \mathbb{R}, +, *, 0, 1, < \rangle$, and generalized relations describe sets in $\mathbb{R}^n$.

A database given by a quantifier-free formula $\alpha(x_1, \ldots, x_n)$ in the language of Ω defines a (possibly infinite) subset of $\mathcal{U}^n$ given by $D_\alpha = \{\vec{a} \in \mathcal{U}^n \mid \mathcal{M} \models \alpha(\vec{a})\}$. Such databases are called *finitely representable* [16], as the formula α provides a finitary means for representing an infinite set. For example, if $\alpha(x, y) \equiv (x^2 + y^2 \leq 1)$, then D_α is the circle of radius 1 with the center in $(0, 0)$.

Relational calculus can be straightforwardly extended to this model by incorporating atomic formulas which are Ω-constraints, that is, atomic Ω-formulae. For example, $\varphi(x, y) \equiv (D(x, y) \wedge y = x^2)$ is a first-order query which, on D_α defined above, returns the intersection of the circle with the graph of the function $y = x^2$.

One of the reasons why the constraint model can be used in spatial applications is that such queries admit a form of safety: the closed form evaluation over structures $\langle \mathbb{R}, +, -, 0, 1, < \rangle$ (linear constraints) and $\langle \mathbb{R}, +, *, 0, 1, < \rangle$ (polynomial constraints), most often used to represent spatial data. This sort of closure is a reformulation of the fact that the two structures above admit *effective quantifier-elimination* (QE). To evaluate a query, one can replace each occurrence of a database symbol D by its representation as a collection of constraints, apply the QE procedure to the resulting formula, and obtain a quantifier-free formula giving a finite representation of the output. There has been work in extending these closure properties to other classes of constraint databases and other logics, and this work indicates that the existence of closure properties is often problematic. For example, for integer gap-order constraints $x <_n y$ (meaning $x < y + n$), restrictions guaranteeing safety were studied in [32] for relational calculus and stratified datalog, and the inherent incompleteness of those restrictions was later shown in [38]. However, in this paper we do obtain new positive results on closure properties for constraint queries—albeit of a different nature than [32] and [38]—and we do so in domains that are quite relevant to spatial applications.

For those domains, we consider the preservation of restricted geometric classes of databases within powerful constraint query languages. In particular, we consider the behavior of queries with polynomial functions over *linear* constraint databases. Linear constraints are used to represent spatial data in many applications; see [13, 17, 29] and references therein. Linear constraints have several advantages over polynomial ones: the QE procedure is less costly, and numerous algorithms have been developed to deal with figures represented by linear constraints; cf. [26]. At the same time, the extension of relational calculus with linear constraints has severely limited power as a query language; see [2, 29]. Thus, it appears to be natural to use a more powerful language, such as relational calculus with polynomial constraints, to query databases represented by linear constraints [43].

As soon as the class of constraints used in queries is more general than the class used to define databases, we encounter the safety/closure property again: the output of a query using polynomial constraints may fail to be definable with linear constraints alone! More generally, if spatial databases are required to have certain geometric properties, then the safety problem is whether those geometric properties are preserved by a given query language.

When the underlying query language is relational calculus with polynomial constraints, there is a recursively enumerable class of programs that express exactly those queries that preserve the property of being definable with linear constraints; this follows from the decidability of the latter property, shown in [13]. Corresponding to our results in the first part of the paper, we are interested in getting explicit and natural complete languages for preserving linear constraints, and also natural effective syntax for other geometric properties. We give a general schema for coming up with such languages. As applications, we consider the properties of being a convex polytope, a convex polyhedron, and a compact semilinear set. For those classes, we provide an effective syntax for polynomial constraint queries preserving the properties. We also show that for unions of conjunctive queries (CQs) with polynomial constraints, it is decidable whether the properties of being a convex polytope or a compact semilinear set are preserved. By applying the growth bounds for the relational calculus with interpreted functions, we find restrictions of the growth in the number of vertices of polytopes and use them to show that certain kinds of triangulations cannot be done even with very powerful constraints.

Organization. In section 2, we present the notation. Sections 3, 4, 5, and 6 all deal primarily with the standard relational calculus with interpreted functions (although generalizations to the "natural semantics" hold, and are given here, as well). Section 3 shows that the underlying interpreted structure one uses does matter: we define the concept of a *safe translation* of queries and show that some structures admit it and some don't. It follows that for many common structures the state-safety problem is decidable.

In section 4, we define the concept of range-restriction and show that the classes of safe and range-restricted queries coincide over well-behaved structures. The general concept of range-restriction is based on the notion of algebraic formulae in the underlying model. We show that for polynomial functions, these range-restricted formulae have a particularly nice characterization, namely, as queries that are bounded by roots of polynomials with coefficients from the database. We then show that for underlying structures admitting QE, it is possible to construct, effectively, a range-restricted query that coincides with a given query Q on all databases for which Q is safe.

In section 5 we show that over well-behaved structures, it is decidable whether a Boolean combination of CQs is safe.

Section 6 establishes the dichotomy result: for every query Q over a well-behaved structure, one can find a polynomial p such that the size of $Q(D)$ is either infinite or at most the value of p on the size of D. We also prove a stronger trichotomy theorem for monotone queries.

Section 7 deals with finitely representable databases. We first introduce a general schema for transferring results about query safety to the finitely representable setting. We then apply this to show bounds on the growth of vertices of polytopes in safe constraint queries, and give effective syntax for queries preserving geometric properties, such as the classes of convex polytopes, polyhedra, and compact semi-linear sets (the latter only in the two-dimensional case). We also show that it is decidable whether unions of CQs with polynomial constraints preserve the first and the third properties. Concluding remarks are given in section 8.

An extended abstract of this paper appeared in [9].

2. Notation.

The notation we use is fairly standard in the literature on constraint databases; cf. [6, 8, 7, 28, 29]. We study databases over infinite structures. Let $\mathcal{M} = \langle \mathcal{U}, \Omega \rangle$ be an infinite structure, where $\mathcal{U}$ is an infinite set, called a carrier (in the database literature it is often called a domain), and Ω is a set of interpreted functions, constants, and predicates. For example, for the real field $\langle \mathbb{R}, +, *, 0, 1, < \rangle$, the carrier is $\mathbb{R}$ (the set of real numbers), and the signature consists of the functions $+$ and $*$, constants 0 and 1, and predicate $<$.

A (relational) *database schema SC* is a nonempty collection of relation names $\{S_1, \ldots, S_l\}$ with associated arities $p_1, \ldots, p_l > 0$. Given $\mathcal{M}$, an *instance* of *SC* over $\mathcal{M}$ is a family of finite sets, $\{R_1, \ldots, R_l\}$, where $R_i \subset \mathcal{U}^{p_i}$. That is, each schema symbol S_i of arity p_i is interpreted as a finite p_i-ary relation over $\mathcal{U}$. Given an instance D, $adom(D)$ denotes its *active domain*, that is, the set of all elements that occur in the relations in D. We normally assume $adom(D) \neq \emptyset$. Although often convenient in simplifying notation, this restriction is by no means necessary, as all results straightforwardly extend to empty databases.

As our basic query language, we consider relational calculus, or *first-order logic*, FO, over the underlying models and the database schema. In what follows, $L(SC, \Omega)$ stands for the language that contains all symbols of SC and Ω; by $\mathrm{FO}(SC, \Omega)$ we mean the class of all first-order formulae built up from the atomic SC and Ω-formulae by using Boolean connectives $\vee, \wedge, \neg$ and quantifiers $\forall, \exists$ and $\forall x \in adom, \exists x \in adom$. When Ω is $(+, -, 0, 1, <)$ or $(+, *, 0, 1, <)$ or $(+, *, e^x, 0, 1, <)$, we use the standard abbreviations $\mathrm{FO}+\textsc{Lin}$, $\mathrm{FO}+\textsc{Poly}$, and $\mathrm{FO}+\textsc{Exp}$, often omitting the schema when it is understood. Regardless of whether we are in the "classical" setting, where these queries are applied to finite databases, or in the constraint query setting discussed later in the paper, we will refer to the syntactic query languages as *relational calculus with* Ω *constraints*.

The semantics is as follows. For a structure $\mathcal{M}$ and an SC-instance D, the notion of $(\mathcal{M}, D) \models \varphi$ is defined in a standard way for $\mathrm{FO}(SC, \Omega)$ formulae, where $\exists x \in adom$ is the *active-domain* quantification. That is, $(\mathcal{M}, D) \models \exists x\ \varphi(x, \cdot)$ if for some $a \in \mathcal{U}$ we have $(\mathcal{M}, D) \models \varphi(a, \cdot)$, and $(\mathcal{M}, D) \models \exists x \in adom\ \varphi(x, \cdot)$ if for some $a \in adom(D)$ we have $(\mathcal{M}, D) \models \varphi(a, \cdot)$. If $\mathcal{M}$ is understood, we write $D \models \varphi$. The output of a query $\varphi(x_1, \ldots, x_n)$ on D is $\{\vec{a} = (a_1, \ldots, a_n) \in \mathcal{U}^n \mid D \models \varphi(\vec{a})\}$, and it is denoted by $\varphi[D]$. For example, $\varphi(x, y) \equiv (S(x, y) \wedge y = x * x)$ is an $\mathrm{FO}+\textsc{Poly}$ query over the schema that contains one binary relation S, and $\varphi[D]$ is the set of pairs in (x, y) in D where $y = x^2$.

We now, following [21, 22], say that an $\mathrm{FO}(SC, \Omega)$ query $\varphi(\vec{x})$ is *safe* for an SC-database D if it has finitely many satisfiers for D; that is, $\varphi[D]$ is finite. A query is safe if it is safe for all databases.

As we explained before, we need to distinguish a class of well-behaved models. Following [6, 7, 8], we use *o-minimality* [30, 42] and QE [11] for this purpose. We say that $\mathcal{M}$ is o-minimal if every definable set is a finite union of points and open intervals $\{x \mid a < x < b\}$, $\{x \mid x < a\}$, and $\{x \mid x > a\}$ (we assume that $<$ is in Ω). Definable sets are those of the form $\{x \mid \mathcal{M} \models \varphi(x)\}$, where φ is a first-order formula in the language of $\mathcal{M}$, possibly supplemented with symbols for constants from $\mathcal{M}$. We say that $\mathcal{M}$ admits QE if, for every formula $\varphi(\vec{x})$, there is an equivalent quantifier-free formula $\psi(\vec{x})$ such that $\mathcal{M} \models \forall \vec{x}. \varphi(\vec{x}) \leftrightarrow \psi(\vec{x})$. Below we list the most important examples, which correspond to classes of interpreted structures and constraints often used in applications.

Linear Constraints: $\langle \mathbb{R}, +, -, 0, 1, < \rangle$ is o-minimal and has QE, and its first-order theory is decidable; cf. [11].

Polynomial Constraints: The real field $\langle \mathbb{R}, +, *, 0, 1, < \rangle$ is o-minimal and has QE, and its first-order theory is decidable. This follows from Tarski's theorem; cf. [11, 42].

Exponential Constraints: $\langle \mathbb{R}, +, *, e^x, 0, 1, < \rangle$ is o-minimal [48] but does not have QE [41].

An example of a structure that is not o-minimal is $\langle \mathbb{N}, +, < \rangle$, as the formula $\exists y (x = y + y)$ defines the set of even numbers. Table 2.1 provides examples of some often-encountered structures and their standing with respect to o-minimality and QE.

We will need the following result about o-minimal structures, which will be used numerous times in proofs.

Fact 2.1 (uniform bounds; see [30]). *If $\mathcal{M}$ is o-minimal and $\varphi(x, \vec{y})$ is a first-order formula in the language of $\mathcal{M}$, possibly supplemented with symbols for constants from $\mathcal{M}$, then there is an integer K such that, for each vector $\vec{a}$ from $\mathcal{M}$, the set $\{x \mid \mathcal{M} \models \varphi(x, \vec{a})\}$ is composed of fewer than K intervals.*

If only quantifiers $\forall x \in adom$ and $\exists x \in adom$ are used in a query, it is called an

TABLE 2.1
Examples of structures.

Structure	o-minimal	Has QE
$\langle \mathbb{R}, < \rangle$	Y	Y
$\langle \mathbb{R}, +, -, 0, 1, < \rangle$	Y	Y
$\langle \mathbb{R}, +, *, 0, 1, < \rangle$	Y	Y
$\langle \mathbb{R}, +, *, e^x \rangle$	Y	N
$\langle \mathbb{Q}, < \rangle$	Y	Y
$\langle \mathbb{Q}, +, *, 0, 1, < \rangle$	N	N
$\langle \mathbb{N}, < \rangle$	Y	N
$\langle \mathbb{N}, +, 0, 1, \{\equiv_k\}_{k>1}, < \rangle$	N	Y
$\langle \mathbb{N}, +, *, 0, 1, < \rangle$	N	N

active-semantics query. This is the usual semantics for databases, and it will be the one used in most of the results in this paper. If quantification over the entire infinite universe is allowed, we speak of a *natural-semantics* query. Active-semantics queries admit the standard bottom-up evaluation, while for natural-semantics it is not clear a priori if they can be evaluated at all. However, in many cases one can restrict one's attention to active-semantics queries. The following result was first shown for the pure case (no interpreted structure) in [18] and for linear constraints [28], and then for a large class of structures as follows.

FACT 2.2 (natural-active collapse; see [7, 8]). *If $\mathcal{M}$ is o-minimal and has QE and φ is an arbitrary* $\mathrm{FO}(SC, \Omega)$ *query, then there exists an equivalent* $\mathrm{FO}(SC, \Omega)$ *active-semantics query φ_{act}. Moreover, if the first-order theory of $\mathcal{M}$ is decidable and QE is effective, then the translation $\varphi \to \varphi_{\text{act}}$ is effective.*

We now define the classes of CQs, unions of conjunctive queries (UCQs), and Boolean combinations of conjunctive queries (BCCQs) in the interpreted setting. CQs are built up from atomic *SC* formulae and *arbitrary* Ω-formulae by using $\wedge$ and quantifiers $\exists x$ and $\exists x \in$ *adom*. Note that we can always assume that parameters of each *SC* relation are variables, as Ω-terms can be eliminated by using existential (active-domain) quantifiers. It is easy to see that each CQ can be represented in the form

$$\varphi(\vec{z}) \;\equiv\; \exists \vec{x} \exists \vec{y} \in adom\; S_1(\vec{u}_1) \wedge \ldots \wedge S_k(\vec{u}_k) \wedge \gamma(\vec{x}, \vec{y}, \vec{z}),$$

where S_is are schema relations (not necessarily distinct), $\vec{u}_i$ is a vector of variables from $\vec{x}, \vec{y}, \vec{z}$ of appropriate arity, and γ is an Ω-formula. If $\Omega = \emptyset$ and $\vec{x} = \emptyset$, this is the usual notion of CQs. If γ is quantifier-free, this is the notion of CQs used in [19].

We define UCQs to be built up from atomic *SC* formulae and *arbitrary* Ω-formulae by using $\wedge$, $\vee$, and quantifiers $\exists x$ and $\exists x \in$ *adom*. Again, it is easy to see that those are precisely the queries of the form $\varphi_1 \vee \ldots \vee \varphi_k$, where each φ_i is a CQ. Finally, BCCQs are arbitrary Boolean combinations of CQs.

Although we could define active-domain versions of CQs, the results we state here (e.g., decidability of safety) for the more general classes above will automatically imply the corresponding results for the more restricted class of active-domain conjunctive queries.

For background on finitely representable databases, see the beginning of section 7.

3. Safe translations. The main goal of this section is to show that what kind of interpreted structure one uses does matter when one studies query safety. As a by-product, we show that the state-safety problem is decidable over certain structures.

We study *safe translations*, that is, translations from arbitrary queries into safe ones that do not change the result of a query if it is finite.

Definition 3.1. *We say that there is a safe translation of (active-semantics) first-order queries over $\mathcal{M}$ if there is a function $\varphi \to \varphi_{\text{safe}}$ on (active-semantics) formulae such that for every φ,*

(1) *φ_{safe} is safe, and*

(2) *if φ is safe for D, then $\varphi[D] = \varphi_{\text{safe}}[D]$.*

A translation is canonical if $\varphi_{\text{safe}}[D] = \emptyset$ whenever φ is not safe on D. A translation is recursive if the function $\varphi \to \varphi_{\text{safe}}$ is recursive.

It is known that there are domains over which safe translations do not exist; see [36]. The result of [36] uses quantification over the entire universe in an essential way. Here we restrict our attention to active-domain quantification. The following generalizes our result from [8].

Proposition 3.2. *Let $\mathcal{M}$ be o-minimal, be based on a dense order, admit effective QE, and have a decidable theory. Then there exists a recursive canonical safe translation of active-semantics formulae over $\mathcal{M}$.*

Proof. Given an active-semantics formula φ, let $\alpha(x)$ be a formula defining the active domain of the output of φ. Let Ψ be an active-semantics sentence equivalent to

$$\neg\exists x_1, x_2\ ((x_1 < x_2) \wedge (\forall x\ x_1 < x < x_2 \to \alpha(x)))$$

(it exists by Fact 2.2). We then define φ_{safe} as $\varphi \wedge \Psi$. The proposition then follows from the following claim: $D \models \Psi$ iff $\varphi[D]$ is finite.

First, assume that $D \models \neg\Psi$; then clearly $\varphi[D]$ is infinite because $<$ is dense. Suppose $D \models \Psi$. We then look at all occurrences of SC predicates in α and replace each of them with a disjunction of tuples. This results in $\alpha'(x)$ in the language of Ω and constants for elements of $\mathcal{U}$; further, $D \models \alpha(a)$ iff $\mathcal{M} \models \alpha'(a)$. Let Ψ' be obtained from Ψ by substituting α' for α. We then have $\mathcal{M} \models \Psi'$. Since $\alpha'(\mathcal{M}) = \{a \mid \mathcal{M} \models \alpha'(a)\}$ is a finite union of points and intervals, and since $\mathcal{M} \models \Psi'$, it follows that no nondegenerate interval is in $\alpha'(\mathcal{M})$. Thus, from o-minimality, we get that $\alpha'(\mathcal{M})$ is a finite union of points. Hence, $\{a \mid D \models \alpha(a)\}$ is finite, thus showing finiteness of $\varphi[D]$. ◻

Examples of structures satisfying the conditions of Proposition 3.2 are $\langle \mathbb{R}, +, -, 0, 1, <\rangle$ and $\langle \mathbb{R}, +, *, 0, 1, <\rangle$. An immediate corollary to the proposition above is the following.

Corollary 3.3. *Let $\mathcal{M}$ be as in Proposition* 3.2. Then the state-safety problem over $\mathcal{M}$ is decidable. That is, given a first-order query φ, and a database D, it is decidable whether $\varphi[D]$ is finite. ◻

Proof. We showed that the active-semantics sentence Ψ tests whether $\varphi[D]$ is finite. ◻

We next show that safe translations (recursive or not!) need not exist even when one restricts one's attention to active-semantics queries and all predicates in the signature Ω are computable.

Proposition 3.4. *There is a structure $\mathcal{M} = \langle \mathbb{N}, P\rangle$, where P is a computable predicate, such that there is no safe translation of active-semantics first-order queries over $\mathcal{M}$.*

Proof. Let P be a ternary predicate defined as $P(i, j, k)$ iff the ith Turing machine on the input j makes at least k moves (assuming some standard encoding of machines and inputs). Consider the schema that consists of a single binary relation U. Assume to the contrary that there is a safe translation over $\mathcal{M}$. Let

$\varphi(k) \equiv \exists i, j \in \mathit{adom}\ U(i,j) \wedge P(i,j,k)$, and let $\psi(k)$ be φ_{safe}. Note that ψ is an active-domain formula in the language of U and P. We now show how to use ψ to decide the halting problem.

Suppose we are given the ith machine M_i and the input j. We assume without loss of generality that M_i makes at least one move on j. Define D that consists of a single pair (i,j). Since we know that ψ is safe, we then compute the minimum number l such that $D \not\models \psi(l)$. It is computable since (a) it exists, and (b) for each k, it is decidable whether $D \models \psi(k)$.

Assume that $D \models \varphi(l)$. Then M_i does not halt on j. Indeed, if M_i halts, then $\varphi[D]$ is finite, and hence $\varphi[D] = \psi[D]$, but we have $l \in \varphi[D] - \psi[D]$. Assume that $D \not\models \varphi(l)$. Then M_i makes $k < l$ moves on j, and thus halts. Hence, $D \models \varphi(l)$ iff M_i halts on j. Since it is decidable whether $D \models \varphi(l)$, we get a contradiction. ∎

In the remainder of the paper, we concentrate on *well-behaved* structures, typically o-minimal ones. For computability, we often impose QE and decidability of first-order theory.

4. Range-restriction and safety. Let us informally describe the concept of range-restriction for databases over interpreted structures. It can be seen as a generalization to arbitrary structures of the idea of finiteness dependencies [31]. Consider a query $\varphi(x)$ over a database which is a finite set S of real numbers:

$$\varphi(x,y) \ \equiv\ \exists z\ [S(z) \wedge (x > y) \wedge (x > 0) \wedge (x * x = z) \wedge (y + y = z)].$$

This query defines a set of pairs of reals. Clearly, it is safe, as the size of its output is at most twice the size of the input. Moreover, and this is the key observation, from the query φ and any database S, one can compute an upper bound on the output $\varphi[S]$: indeed, every element in $\mathit{adom}(\varphi[S])$ is either $\sqrt{a}$ or $\frac{a}{2}$ for some element $a \in S$. Equivalently, in this upper bound every element is a solution to either $x^2 = a$ or $2x = a$ when a ranges over S. That is, in this example, an upper bound on the result of a safe query can be found as the set of roots of polynomials with coefficients coming from the active domain of the database and a finite set of constants.

This is essentially the idea of range-restriction: we find, for a safe query, a set of formulae defining an upper bound on the output. A similar approach was used in [14], although the focus of [14] is different: it does not consider how the underlying structure affects safety, but instead gives a syntactically restricted class of queries with interpreted functions that can be translated into algebra expressions, along the lines of [46]. In contrast, we are interested in how the underlying structure interacts with queries. For example, we show that not only does a set of range-restriction formulae exist for any query over a well-behaved structure, but also, under additional conditions, it can be found effectively. The result that we prove is actually a bit stronger, as it shows that the upper bound works not only for safe queries, but for arbitrary queries, provided they are safe on a given database.

The first difficulty we encounter is finding an analog of the set of roots of polynomials, when we deal with arbitrary structures (e.g., $\langle \mathbb{R}, +, *, e^x \rangle$). The solution to this is provided by the model-theoretic notion of *algebraic* formulae, reviewed in section 4.1. The range-restriction theorem is proved in section 4.2. Then, in section 4.3, we give two examples: the pure case, where our main result trivially translates into a classical relational theory result, and a much less trivial FO + POLY case, where we confirm the original intuition that the upper bound is a set of roots of polynomials. We finish the section by giving a couple of extensions of the main result.

4.1. Algebraic formulae over o-minimal structures. In this subsection, we study formulae over $\mathcal{M}$, that is, first-order formulae in the language $L(\Omega)$. We shall consider formulae $\varphi(\vec{x};\vec{y})$ with distinguished parameter variables $\vec{y}$; we use ";" to separate those variables. Assume that $\vec{x}$ is of length n and $\vec{y}$ is of length m. Such a formula (in the language of Ω and constants for elements of $\mathcal{U}$) is called *algebraic* if for each $\vec{b}$ in $\mathcal{U}^m$ there are only finitely many satisfiers of $\varphi(\vec{x},\vec{b})$; that is, the set $\{\vec{a} \in \mathcal{U}^n \mid \mathcal{M} \models \varphi(\vec{a};\vec{b})\}$ is finite. A collection of formulae is algebraic if each of its elements is algebraic.

For example, the formula $\varphi(x;y) \equiv (x^2 = y)$ is algebraic over $\langle \mathbb{R},+,*,0,1,<\rangle$. It can easily be seen that if $\varphi_1(\vec{x};\vec{y})$ and $\varphi_2(\vec{x};\vec{y})$ are algebraic, then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\exists x \varphi_1$, where x is one of the variables in $\vec{x}$; however, algebraic formulae are not closed under negation.

Let us list some simple properties of algebraic formulae over o-minimal structures.

LEMMA 4.1. *Let $\mathcal{M} = \langle \mathcal{U},\Omega\rangle$ be o-minimal and based on a dense order, and let $\gamma(\vec{x};\vec{y})$ be algebraic. Then*

- *There exists a number K such that for any $\vec{b} \in \mathcal{U}^m$, the set $\{\vec{a} \in \mathcal{U}^n \mid \mathcal{M} \models \gamma(\vec{a};\vec{b})\}$ has fewer than K elements.*
- *There is a recursively enumerable collection of algebraic formulae $\{\chi_i(\vec{x};\vec{y})\}$ such that every algebraic formula $\gamma(\vec{x};\vec{y})$ is equivalent to one of the χ_i.*[1] *If $\mathcal{M}$ is decidable, then the collection of algebraic formulae is recursive.*

Proof. For the first item, consider all formulae $\gamma_i(x_i;\vec{y}) \equiv \exists \vec{x}^{(i)}\ \gamma(\vec{x};\vec{y})$, where $\vec{x}^{(i)}$ contains all components of $\vec{x}$ except x_i. By Fact 2.1, there is an integer K_i such that for each $\vec{b}$, $\{a \mid \mathcal{M} \models \gamma_i(a;\vec{b})\}$ is composed of fewer than K_i intervals. Since γ is algebraic and $<$ is dense, it means that $\{a \mid \mathcal{M} \models \gamma_i(a;\vec{b})\}$ has fewer than K_i elements. Hence, there are at most $K = \prod_{i=1}^n K_i$ vectors $\vec{a}$ such that $\mathcal{M} \models \gamma(\vec{a};\vec{b})$.

For the second item, for each $\gamma(\vec{x};\vec{y})$, let $\chi_\gamma^k(\vec{x})$ be a first-order formula

$$\forall \vec{y} \neg \exists \vec{x}_1 \ldots \exists \vec{x}_k . \bigwedge_{i=1}^{k} \gamma(\vec{x}_i;\vec{y}) \wedge \bigwedge_{i\neq j} (\vec{x}_i \neq \vec{x}_j),$$

saying that there are fewer than k vectors $\vec{x}$ that satisfy $\gamma(\vec{x};\vec{y})$ for all $\vec{y}$. Consider the recursively enumerable collection of formulae of the form $\gamma(\vec{x};\vec{y}) \wedge \chi_\gamma^k(\vec{x})$, where γ ranges over $\mathrm{FO}(\Omega)$ and k ranges over $\mathbb{N}$. It follows from the first item that it enumerates all algebraic formulae. If $\mathcal{M}$ is decidable, it is also decidable whether $\gamma(\vec{x};\vec{y})$ is algebraic. Indeed, the latter happens iff $\gamma^*(x;\vec{y}) = \bigvee_i \gamma_i(x;\vec{y})$ is algebraic (see the proof of the first item), which in turn happens iff there is no open interval contained in $\{a \mid \mathcal{M} \models \gamma^*(a;\vec{b})\}$ for any $\vec{b}$. The latter can be formulated as a first-order sentence $\forall \vec{y} \neg \exists u, v \forall z.(u < z < v \rightarrow \gamma^*(z;\vec{y}))$. $\square$

While we do get an enumeration of algebraic formulae from Lemma 4.1, we need one more representation for algebraic formulae as a tool in proofs. This representation is provided below. We first treat the one-variable case, that is, formulae $\varphi(x;\vec{y})$.

Let $\Xi = \{\xi_1(x;\vec{y}),\ldots,\xi_k(x;\vec{y})\}$ be a collection of formulae. Let

$$\mathrm{same}_\Xi(x,x';\vec{y}) \quad\equiv\quad \bigwedge_{i=1}^{k} (\xi_i(x;\vec{y}) \leftrightarrow \xi_i(x';\vec{y})).$$

[1]Provided that Ω is a recursive set.

Now define

$$\beta_\Xi(x;\vec{y}) \ \equiv\ \forall x', x''.x' < x < x'' \to (\exists z.x' \le z \le x'' \wedge \neg\mathrm{same}_\Xi(x,z;\vec{y})).$$

This provides the desired syntactic characterization.

PROPOSITION 4.2. *Let $\mathcal{M}$ be an o-minimal structure based on a dense order. Then a formula $\varphi(x;\vec{y})$ is algebraic (over $\mathcal{M}$) iff there exists a collection of formulae Ξ such that φ is equivalent to β_Ξ. A formula $\varphi(\vec{x};\vec{y})$ is algebraic iff there exists a collection of formulae Ξ in variables $(x;\vec{y})$ and a formula $\psi(\vec{x};\vec{y})$ such that φ is equivalent to $\beta_\Xi(x_1;\vec{y}) \wedge \ldots \wedge \beta_\Xi(x_n;\vec{y}) \wedge \psi(\vec{x};\vec{y})$.*

Proof. Prove the one-variable case first.

Let Ξ be a collection of formulae and assume that β_Ξ is not algebraic. That is, for some $\vec{b}$ over $\mathcal{U}$, $\beta_\Xi(\mathcal{M};\vec{b}) = \{a \mid \mathcal{M} \models \beta_\Xi(a;\vec{b})\}$ is infinite. Since $\mathcal{M}$ is o-minimal, $\beta_\Xi(\mathcal{M};\vec{b})$ is a finite union of points and intervals. Since $<$ is dense, it means that there exist $a_0 < b_0 \in \mathcal{U}$ such that $[a_0,b_0] \subseteq \beta_\Xi(\mathcal{M};\vec{b})$. We now consider the formulae $\xi'_i(x) = \xi_i(x;\vec{b})$ for all $\xi_i \in \Xi$. Since both $\xi'_i(\mathcal{M}) = \xi_i(\mathcal{M};\vec{b})$ and $\neg\xi'_i(\mathcal{M}) = \neg\xi_i(\mathcal{M};\vec{b})$ are finite unions of intervals and $<$ is dense, for every nondegenerate interval J, it is the case that either $J \cap \xi'_i(\mathcal{M})$ or $J \cap \neg\xi'_i(\mathcal{M})$ contains an infinite (closed) interval. Using this, we construct a sequence of intervals as follows: $I_0 = [a_0,b_0]$, $I_1 \subseteq I_0$ is an interval that is contained either in $I_0 \cap \xi'_1(\mathcal{M})$ or in $I_0 \cap \neg\xi'_1(\mathcal{M})$. At the jth step, $I_j \subseteq I_{j-1}$ is an interval that is contained either in $I_{j-1} \cap \xi'_j(\mathcal{M})$ or in $I_{j-1} \cap \neg\xi'_j(\mathcal{M})$. Let $I = I_k$. Then, for any $c, d \in I$, $\mathcal{M} \models \xi_i(c,\vec{b}) \leftrightarrow \xi_i(d;\vec{b})$.

Since $I = [a',b'] \subseteq [a_0,b_0]$ and $\mathcal{M} \models \beta_\Xi(c;\vec{b})$ for all $c \in I$, we obtain that, for every $c \in (a',b')$, there exists $d \in [a',b']$ such that $\mathcal{M} \models \neg\mathrm{same}_\Xi(c,d;\vec{b})$. That is, for some $\xi_i \in \Xi$, $\mathcal{M} \models \neg(\xi_i(c;\vec{b}) \leftrightarrow \xi_i(d;\vec{b}))$, which is impossible by construction of I. This proves that β_Ξ is algebraic.

For the converse, we let Ξ consist of just φ for any $\varphi(x;\vec{y})$. That is, $\beta_\Xi(x;\vec{y})$ is

$$\forall x', x''.x' < x < x'' \to (\exists z.x' \le z \le x'' \wedge \neg(\varphi(x;\vec{y}) \leftrightarrow \varphi(z;\vec{y}))).$$

We claim that φ and β_Ξ are equivalent if φ is algebraic. Fix any $\vec{b}$ of the same length as $\vec{y}$, and assume that $\varphi(a;\vec{b})$ holds. If $\beta_\Xi(a;\vec{b})$ does not hold, then there exist $a' < a < a''$ such that for every $c \in [a',a'']$, $\varphi(c;\vec{b}) \leftrightarrow \varphi(a;\vec{b})$ holds; thus, $\varphi(c;\vec{b})$ holds for infinitely many c, contradicting algebraicity of φ. Hence, $\beta_\Xi(a;\vec{b})$ holds. Conversely, assume that $\beta_\Xi(a;\vec{b})$ holds. If $\varphi(a;\vec{b})$ does not hold, then there is an interval containing a on which $\varphi(\cdot;\vec{b})$ does not hold. Indeed, $\neg\varphi(\mathcal{M};\vec{b})$ is a finite union of intervals whose complement is a finite set of points, so the above observation follows from the density. We now pick $a' < a''$ such that $\varphi(\cdot;\vec{b})$ does not hold on $[a',a'']$. Since $\beta_\Xi(a;\vec{b})$ holds, we find $c \in [a',a'']$ such that $\neg(\varphi(a;\vec{b}) \leftrightarrow \varphi(c;\vec{b}))$ holds; that is, $\varphi(c;\vec{b})$ holds for $c \in [a',a'']$, which is impossible. Thus, we conclude that $\varphi(a;\vec{b})$ holds, proving that for any $\vec{b}$, $\forall x.\ (\varphi(x;\vec{b}) \leftrightarrow \beta_\Xi(x;\vec{b}))$. This finishes the one-variable case.

For the multivariable case, we note that algebraicity of β_Ξ implies that $\varphi'(\vec{x};\vec{y}) = \beta_\Xi(x_1;\vec{y}) \wedge \ldots \wedge \beta_\Xi(x_n;\vec{y}) \wedge \psi(\vec{x};\vec{y})$ is algebraic. Conversely, let $\varphi(\vec{x};\vec{y})$ be algebraic. Consider

$$\varphi_i(x_i;\vec{y}) = \exists x_1 \ldots \exists x_{i-1} \exists x_{i+1} \ldots \exists x_n.\varphi(x_1,\ldots,x_i,\ldots,x_n;\vec{y}).$$

Let $\chi(x;\vec{y})$ be $\varphi_1(x;\vec{y}) \vee \ldots \vee \varphi_n(x;\vec{y})$. Obviously each φ_i is algebraic, and thus $\chi(x;\vec{y})$ is algebraic. Hence, $\chi(x;\vec{y})$ is equivalent to $\beta_\Xi(x;\vec{y})$ for some finite collection

Ξ of formulae in $(x;\vec{y})$. Note that if $\varphi(\vec{a};\vec{b})$ holds and a_i is the ith component of $\vec{a}$, then $\chi(a_i;\vec{b})$ holds and thus $\beta_\Xi(a_i;\vec{b})$ holds. This shows that φ is equivalent to $\beta_\Xi(x_1;\vec{y}) \wedge \ldots \wedge \beta_\Xi(x_n;\vec{y}) \wedge \varphi(\vec{x};\vec{y})$, thus completing the proof. ◻

Corollary 4.3. *If $\mathcal{M}$ is an o-minimal structure based on a dense order, and $\varphi(\vec{x};\vec{y})$ is algebraic over $\mathcal{M}$, then φ is algebraic over any $\mathcal{M}'$ elementary equivalent to $\mathcal{M}$.* ◻

Proof. There exists a collection of formulae Ξ such that $\mathcal{M} \models \forall\vec{x}\forall\vec{y}.\varphi(\vec{x};\vec{y}) \leftrightarrow (\varphi(\vec{x};\vec{y}) \wedge \bigwedge_i \beta_\Xi(x_i;\vec{y}))$. Hence the same sentence is true in $\mathcal{M}'$. Since $\mathcal{M}'$ is also o-minimal and based on a dense order, we get from Proposition 4.2 that φ is algebraic in $\mathcal{M}'$, too. ◻

4.2. Main theorem. We start with a few definitions. For an $L(\Omega)$-formula $\gamma(\vec{x};\vec{y})$ and database D, let

$$\gamma(D) \;=\; \{\vec{a} \mid \exists\vec{b} \in adom(D)^m \text{ such that } D \models \gamma(\vec{a};\vec{b})\}.$$

If Γ is a collection of formulae in $\vec{x};\vec{y}$, define

$$\Gamma(D) = \bigcup_{\gamma\in\Gamma} \gamma(D).$$

Note that if Γ is algebraic and finite, then $\Gamma(D)$ is finite.

Definition 4.4 (range-restriction). *Let $\mathcal{M}$ be an interpreted structure. A* range-restricted query *over $\mathcal{M}$ and a database schema SC is a pair $Q = (\Gamma, \varphi(\vec{x}))$, where Γ is a finite collection $\{\gamma_1(\vec{x};\vec{y}), \ldots, \gamma_m(\vec{x};\vec{y})\}$ of algebraic $L(\Omega)$-formulae, and $\varphi(\vec{x})$ is an $L(SC, \Omega)$ query.*

The semantics of Q is as follows:

$$Q[D] \quad = \quad \{\vec{a} \in \Gamma(D) \mid D \models \varphi(\vec{a})\}.$$

That is, Γ provides an upper bound on the output of a query; within this bound, a usual first-order query is evaluated. For example, let $\varphi(x)$ be the FO + Poly query $S(x) \vee (x > 5)$. Clearly, it is not safe. Now let $\gamma(x;y) \equiv (x * x = y)$ and $Q = (\{\gamma\}, \varphi)$. Then, for any database S (which is a finite set of the reals), $Q[S]$ is the set of those elements a such that $a^2 \in S$ and either $a \in S$ or $a > 5$. Clearly, this is a finite set.

Observation. Every range-restricted query is safe.

We call a range-restricted query (Γ, φ) *active semantics* if φ is an active-semantics formula. Note that Γ does not mention the database. It turns out that range-restricted active queries characterize all the safe active-semantics queries in the following sense.

Theorem 4.5. *Let $\mathcal{M}$ be any o-minimal structure based on a dense linear order. Then there is a function* Make_Safe *that takes as input an active-domain formula $\varphi(\vec{x})$ and outputs a range-restricted active query $Q = (\Gamma, \psi)$ with the property that* Make_Safe(φ) *is equivalent to φ on all databases D for which φ is safe. Furthermore, if $\mathcal{M}$ has effective QE, then* Make_Safe *is recursive.*

Proof. We deal with the one-variable case first. Let

$$\varphi(z) \;\equiv\; Q_1 w_1 \in adom \ldots Q_1 w_l \in adom.\alpha(z, \vec{w}),$$

where each Q_i is $\exists$ or $\forall$ and $\alpha(z,\vec{w})$ is quantifier-free, and all atomic subformulae $R(\cdots)$ contain only variables, excluding z. Any formula can be transformed into such by adding existential quantifiers; cf. [6, 8]. Let $\Xi = \{\xi_i(z,\vec{w}) \mid i = 1, \ldots, k\}$ be the collection of all Ω-atomic subformulae of α. We may assume without loss of generality that the length of $\vec{w}$ is nonzero, and that Ξ is nonempty. (If this is not true for φ, take $\varphi \wedge \forall w \in adom(w = w)$ and transform it to the above form.)

Define $\text{same}_\Xi(a, b, \vec{w})$, as before, to be $\bigwedge_{i=1}^{k}(\xi_i(a, \vec{w}) \leftrightarrow \xi_i(b, \vec{w}))$, and define $\gamma(x; \vec{w})$ to be $\beta_\Xi(x; \vec{w})$; that is, $\gamma(x, \vec{w}) \equiv \forall x', x''. x' < x < x'' \to \exists y.(x' \leq y \leq x'' \wedge \neg\text{same}_\Xi(x, y, \vec{w}))$. Now Γ consists just of γ, with $\vec{w}$ being distinguished parameters. We let Make_Safe(φ) output $(\{\gamma\}, \varphi)$.

Since γ is algebraic by Proposition 4.2, we must show that $\{a \mid D \models \varphi(a)\} = \{a \in \Gamma(D) \mid D \models \varphi(a)\}$ for every nonempty database for which φ is safe.

Assume otherwise; that is, for some nonempty D for which φ is safe, we have $D \models \varphi(a)$ but $a \notin \Gamma(D)$. Let $\vec{c}_1, \dots, \vec{c}_M$ be an enumeration of all vectors of the length of $\vec{w}$ of elements of the active domain. Note that $M > 0$. Since $a \notin \Gamma(D)$, we have that for each $i = 1, \dots, M$, there exist a_i', a_i'' such that $a_i' < a < a_i''$ and $\mathcal{M} \models \text{same}_\Xi(a, c, \vec{c}_i)$ for all $c \in [a_i', a_i'']$.

Let $b' = \max\{a_i'\}, b'' = \max\{a_i''\}$. We have $b' < a < b''$, and for each $\vec{c}$ (of length of $\vec{w}$) over the active domain, we have $\xi_i(a; \vec{c}) \leftrightarrow \xi_i(c, \vec{c})$ for every $c \in [b', b'']$. From this, by a simple induction on the structure of the formula (using the fact that z does not appear in any atomic formula $R(\cdots)$), we obtain that $D \models \alpha(a, \vec{c}) \leftrightarrow \alpha(c, \vec{c})$ for every $\vec{c}$ over $adom(D)$ and every $c \in [b', b'']$, and thus $D \models \varphi(a) \leftrightarrow \varphi(c)$, which implies that φ is not safe for D. This contradiction proves correctness of Make_Safe for the one-variable case.

This completes the proof for the one-variable case. We handle the multivariable case by reducing to the one-variable case.

Let $\mathcal{M}' = \langle \mathcal{U}, \Omega' \rangle$ be a definable extension of $\mathcal{M}$ that has QE. Note that $\mathcal{M}'$ is o-minimal. Let $\varphi(z_1, \dots, z_n)$ be given and define

$$\varphi_i(z_i) \quad \equiv \quad \exists z_1 \dots \exists z_{i-1} \exists z_{i+1} \dots \exists z_n . \varphi(z_1, \dots, z_{i-1}, z_i, z_{i+1}, \dots, z_n).$$

By [7, 8], there is an $L(SC, \Omega')$ *active* formula $\psi_i(z_i)$ such that $D \models \forall z. \psi_i(z) \leftrightarrow \varphi_i(z)$ for all D. Let $(\{\gamma_i(z_i, \vec{w}_i)\}, \psi_i(z_i))$ be the output of Make_Safe on ψ_i. Since $\mathcal{M}'$ is a definable extension, we can assume without loss that γ_i is an Ω-formula.

We now define

$$\gamma(\vec{z}; \vec{w}_1, \dots, \vec{w}_n) \quad \equiv \quad \gamma_1(z_1; \vec{w}_1) \wedge \dots \wedge \gamma_n(z_n; \vec{w}_n),$$

where each $\vec{w}_i$ is of the same length as the vector of distinguished parameters in the formulae γ_i. Finally, Make_Safe(φ) outputs $(\{\gamma\}, \varphi)$. To see that it works, first notice that algebraicity of all γ_i's implies algebraicity of γ. Now assume that $D \models \varphi(\vec{a})$ where $\vec{a} = (a_1, \dots, a_n)$. Then $D \models \varphi_i(a_i)$, and thus for some vector $\vec{c}_i$ of elements of the active domain, we have that $\gamma_i(a_i, \vec{c}_i)$ holds. Thus, if $\vec{c}$ is the concatenation of all $\vec{c}_i$s, then $\gamma(\vec{a}, \vec{c})$ holds, showing that $\vec{a} \in \Gamma(D)$, where $\Gamma = \{\gamma\}$. This completes the proof of the multivariable case.

We finally notice that Make_Safe for one-variable formulae is recursive, no matter what $\mathcal{M}$ is. For the multivariable case, to make it recursive, we need a procedure for converting natural-quantification formulae into active-quantification formulae. Such a procedure exists by Fact 2.2. $\square$

COROLLARY 4.6 (range-restricted = safe). *For any o-minimal structure based on a dense order, the class of safe active-semantics queries is the same as the class of range-restricted queries.* $\square$

Combining this with the natural-active collapse (Fact 2.2), we obtain the following corollary.

Corollary 4.7. *Let $\mathcal{M}$ be any o-minimal structure based on a dense linear order that admits QE. Then there is a function* Make_Safe *(recursive if so is QE and $\mathcal{M}$ is decidable) that takes as input a natural-semantics formula $\varphi(\vec{x})$, and outputs a range-restricted active query $Q = (\Gamma, \psi)$ with the property that* Make_Safe(φ) *is equivalent to φ on all databases D for which φ is safe. In particular, over $\mathcal{M}$, the classes of safe natural-semantics queries and range-restricted queries coincide.* $\square$

Corollary 4.8. *For any o-minimal structure based on a dense order (decidable or not), the collection of safe queries is recursively enumerable.* $\square$

We finish this section with a proposition that follows from the special form of formulae in Γ, as established in the proof of Theorem 4.5.

Proposition 4.9. *Let $\mathcal{M}$ be o-minimal and based on a dense order. Let $\varphi(\vec{x})$ be a first-order query. Then there exists a set Γ of algebraic formulae $\gamma(x; \vec{y})$ (that can be effectively constructed if $\mathcal{M}$ has effective QE and is decidable) such that, for any database D, if $\varphi[D]$ is finite, then $adom(\varphi[D]) \subseteq \Gamma(D)$.*

Proof. Assume φ is active-semantics. Then the proof follows the proof of Theorem 4.5, but at the end we replace $\gamma(x_1, \ldots, x_n; \vec{y})$ by

$$\gamma'(x; \vec{y}) \;=\; \bigvee_{i=1}^{n} \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \; \gamma(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n; \vec{y})$$

to get a bound on the active domain, and output $(\{\gamma'\}, \varphi)$.

For an arbitrary φ, let $\mathcal{M}'$ be a definitional expansion of $\mathcal{M}$ that has QE (such an expansion always exists; cf. [11]). Since $\mathcal{M}$ and $\mathcal{M}'$ are elementary equivalent, $\mathcal{M}'$ is o-minimal [30] and the order is dense. Thus, there exists an active-semantics FO$(\mathcal{M}', SC)$ query $\psi(\vec{x})$ that is equivalent to φ (see Fact 2.2), and, by the above, we have a formula γ_0 in the language of $\mathcal{M}'$ such that $adom(\varphi[D]) = adom(\psi[D]) \subseteq \gamma_0(D)$. Now obtain an $L(\Omega)$ formula γ from γ_0 by replacing each new predicate symbol from $\mathcal{M}'$ by its definition by an $L(\Omega)$ formula. Then $\gamma(D) = \gamma_0(D)$, which proves the proposition. $\square$

4.3. Examples.

Below we give two examples. We consider the pure case, where Theorem 4.5 translates into a well-known result, and we also consider the case of polynomial constraints and show a more explicit form of range-restriction.

The pure case. We assume that we have the pure relational calculus; that is, our underlying structure is $\mathcal{M}^{\emptyset} = \langle \mathcal{U}, \emptyset \rangle$. The reason we can apply Theorem 4.5 is that we can extend $\mathcal{M}$ to $\mathcal{M}^{<} = \langle \mathcal{U}, < \rangle$ by adding a dense order without endpoints on $\mathcal{U}$; although it is never mentioned in queries, $\mathcal{M}^{<}$ is o-minimal (and has QE), and hence the results are applicable. Next, we need the following lemma.

Lemma 4.10. *Let $\gamma(x; \vec{y})$ be algebraic over $\mathcal{M}^{<}$. If $\mathcal{M}^{<} \models \gamma(a; \vec{b})$, then a coincides with one of $\vec{b}$'s components.*

Proof. Assume not and fix a counterexample $a, \vec{b}$ and let B be the set of components of $\vec{b}$ together with $-\infty, \infty$. Let b', b'' in B be such that $b' < a < b''$ and (b', b'') contains no other member of B. Since $\mathcal{M}^{<}$ has QE, γ is equivalent to a quantifier-free formula. For any $a' \in (b', b'')$, $(a, \vec{b})$ and $(a', \vec{b})$ satisfy the same atomic $<$-formulae, and hence $\mathcal{M}^{<} \models \gamma(a, \vec{b}) \leftrightarrow \gamma(a', \vec{b})$, which contradicts algebraicity. $\square$

This immediately implies that for an algebraic Γ and a database D, $\Gamma(D)$ is either empty (if Γ contains no formulae) or $\Gamma(D) = adom(D)$. Thus, the classes of safe pure relational calculus queries and relational calculus queries whose output is restricted to the active domain coincide. Of course, this is a standard result in database theory, proved here in a rather unusual way.

The real field: $\mathrm{FO} + \textsc{Poly}$. Can we find a more concrete representation of range-restricted queries over the real field? Intuitively, it should be sufficient to look for roots of polynomials $p(x, \vec{a})$, where $\vec{a}$ ranges over tuples of elements of the active domain, as was suggested by the example in the beginning of the section. However, even quantifier-free algebraic formulae do not give us this representation directly. Nevertheless, the following can be shown.

Let $p(x, \vec{y})$ be a multivariate polynomial over the real field. Define $\mathit{Root}(p, \vec{a})$ as $\emptyset$ if $p(x, \vec{a})$ is identically zero, and the set of roots of $p(x, \vec{a})$ otherwise. Given a collection P of polynomials $\{p_1(x, \vec{y}), \ldots, p_m(x, \vec{y})\}$ and a database D, let

$$P(D) \; = \bigcup_{i=1}^{m} \; \bigcup_{\vec{a} \subseteq \mathit{adom}(D)} \mathit{Root}(p_i, \vec{a}),$$

where $\vec{a}$ ranges over tuples of elements of $\mathit{adom}(D)$, of the same length as $\vec{y}$.

DEFINITION 4.11. *A query in polynomial range-restricted form is a pair (P, φ), where P is a finite collection of multivariate polynomials and $\varphi(x_1, \ldots, x_n)$ is a* $\mathrm{FO} + \textsc{Poly}$ *query. The semantics is defined as $(P, \varphi)[D] = \varphi[D] \cap P(D)^n$.*

PROPOSITION 4.12. *The class of safe* $\mathrm{FO} + \textsc{Poly}$ *queries (arbitrary or active-semantics) coincides with the class of queries in polynomial range-restricted form. Moreover, for every* $\mathrm{FO}{+}\textsc{Poly}$ *query φ, a collection of polynomials P can be effectively found such that φ and (P, φ) are equivalent on all databases on which φ is safe.*

Proof. Given a query $\varphi(\vec{x})$, find effectively a collection of algebraic formulae $\Gamma = \{\gamma_j(x; \vec{y})\}$ such that for any D for which φ is safe, $\mathit{adom}(\varphi[D]) \subseteq \Gamma(D)$. For each $\vec{a}$ and each $\gamma \in \Gamma$, the set $\gamma[\vec{a}] = \{c \mid \gamma(c; \vec{a})\}$ is finite, and by o-minimality there is a uniform bound M such that $\mathit{card}(\gamma[\vec{a}]) < M$ for all $\vec{a}$ and $\gamma \in \Gamma$.

Now let $\gamma_j^i(x; \vec{y})$, $i < M$, be defined as follows: $\mathcal{M} \models \gamma_j^i(c; \vec{a})$ if either (1) $\gamma_j[\vec{a}]$ has at least i elements, and c is the ith element in the order $<$, or (2) $\gamma_j[\vec{a}]$ is nonempty, has fewer than i elements, and c is the largest element of $\gamma_j[\vec{a}]$, or (3) $\gamma_j[\vec{a}]$ is empty, and $c = 0$. Note that $\gamma_j^i(x; \vec{y})$'s are indeed $L(+, *, 0, 1, <)$ formulae. It is easy to see that each $\gamma_j^i(x; \vec{y})$ defines a function $f_{ij} : \mathbb{R}^m \to \mathbb{R}$, where m is the length of $\vec{y}$, by $f_{ij}(\vec{a}) = c$ iff c is the unique element such that $\gamma_j^i(c; \vec{a})$ holds. Furthermore, this function is semialgebraic and the following property holds: if φ is safe for D, then $\mathit{adom}(\varphi[D])$ is contained in $\bigcup_{i,j} \bigcup_{\vec{a}} f_{ij}(\vec{a})$, where $\vec{a}$ ranges over $\mathit{adom}(D)$.

It follows from [23] that each $f_{ij}(\vec{y})$ is algebraic; that is, there exists a polynomial $p_{ij}(x, \vec{y})$ such that $p_{ij}(x, \vec{y}) = 0$ iff $x = f_{ij}(\vec{y})$. It is easy to see that for $P = \{p_{ij} \mid i < M, \gamma_j \in \Gamma\}$, $\Gamma(D) \subseteq P(D)$ and $P(D)$ is always finite. To complete the proof, we must show effectiveness. We can effectively construct Γ, and thus find M (by writing formulae saying that each $\gamma(x; \vec{y})$ has fewer than M satisfiers for each $\vec{y}$, and checking if it is true by applying QE; since it is true for some M, the process terminates). Hence, we can effectively construct γ_j^i's, and the procedure for finding p_{ij}'s is effective (although not stated in [23], it follows from the analysis of the proof there). ☐

4.4. Extensions. Suppose we are given two elementary equivalent structures $\mathcal{M}$ and $\mathcal{M}'$, for example, $\langle \mathbb{R}, +, < \rangle$ and $\langle \mathbb{Q}, +, < \rangle$. Assuming $\mathcal{M}$ is o-minimal based on a dense order, so is $\mathcal{M}'$, and thus the characterization of a safe query applies to $\mathcal{M}'$ as well. However, we would like to know more. Suppose φ is safe over $\mathcal{M}$. Is the same φ safe over $\mathcal{M}'$? The positive answer is provided for o-minimal structures.

PROPOSITION 4.13. *If $\mathcal{M}$ is an o-minimal structure based on a dense order, and φ is a safe active-semantics query, then φ is safe in any structure elementary* *equivalent to $\mathcal{M}$.*

Proof. Let $\varphi(\vec{z})$ be safe, and let $(\{\gamma(\vec{z};\vec{w})\},\varphi)$ be the output of Make_Safe (we just saw that it always can be made to have this form). Since φ is safe, we obtain that for every database D over $\mathcal{M}$ and for every $\vec{a}$ of the same length as $\vec{z}$ of elements of $\mathcal{M}$, $D \models \varphi(\vec{a}) \rightarrow \exists \vec{w} \in adom.\gamma(\vec{a};\vec{w})$. Now putting $\neg\varphi$ in prenex form, where only quantified variables appear inside the schema predicates (which can always be done), we have

$$D \models \exists \vec{w} \in adom.Q_1 y_1 \dots Q_n y_n.(\alpha(\vec{a},\vec{y}) \vee \gamma(\vec{a};\vec{w})),$$

where all quantification $Q_i y_i$ is active and α is the quantifier-free part of $\neg\varphi$. We now claim that the same is true in $\mathcal{M}'$ for every D over $\mathcal{M}'$ and every $\vec{a}$ of elements of $\mathcal{M}'$, as long as $\mathcal{M}'$ is elementary equivalent to $\mathcal{M}$. Note that this would imply that $D \models \varphi(\vec{a}) \rightarrow \exists \vec{w} \in adom.\gamma(\vec{a};\vec{w})$ holds in $\mathcal{M}'$ as well.

To prove this claim, assume to the contrary that this fails for some D and $\vec{a}$ over $\mathcal{M}'$. Let $\{b_1,\dots,b_m\}$ be $adom(D)$. We now define a sequence of formulae as follows: $\chi_n(\vec{a},\vec{b},\vec{w},y_1,\dots,y_{n-1}) = \bigvee_{i=1}^{m} \alpha(\vec{a},y_1,\dots,y_{n-1},b_i) \vee \gamma(\vec{a};\vec{w})$ if Q_n is $\exists$; if Q_n is $\forall$, we change $\bigvee$ to $\bigwedge$. Similarly, $\chi_j(\vec{a},\vec{b},\vec{w},y_1,\dots,y_{j-1}) = \bigvee_{i=1}^{m} \chi_{j+1}(\vec{a},\vec{b},\vec{w},y_1,\dots,y_{j-1},b_i)$ if Q_j is $\exists$; otherwise change $\bigvee$ to $\bigwedge$. Next, define $\chi'(\vec{a},\vec{b})$ as $\bigvee_{\vec{w}\in B} \chi_1(\vec{a},\vec{b},\vec{w})$, where B is the collection of all vectors from $\{b_1,\dots,b_m\}$ of the same length as $\vec{w}$. Notice that the only occurrence of the schema predicates is of the form $R(\cdot)$, where we list some b_i's as parameters. We finally replace those by *true* or *false*, depending on whether a particular tuple is in the database D. This results in a formula $\chi(\vec{a},\vec{b})$ that does not mention the schema predicates and has the property that $\mathcal{M}' \models \chi(\vec{a},\vec{b})$ iff $\exists \vec{w} \in adom.Q_1 y_1 \dots Q_n y_n.(\alpha(\vec{a},\vec{y}) \vee \gamma(\vec{a};\vec{w}))$ holds for $\vec{a}$ and the given D with the active domain $\{b_1,\dots,b_m\}$. Now the assumption gives us that $\mathcal{M}' \models \exists \vec{a}\exists \vec{b}.(\neg\chi(\vec{a},\vec{b}) \wedge \bigwedge_{i\neq j} \neg(b_i = b_j))$, thereby showing that the same sentence is true in $\mathcal{M}$. Then, by picking $\vec{a}$ and $\vec{b}$ in $\mathcal{M}$ that witness the failure of χ and defining a database D' on $\vec{b}$ in the same way as D was defined, we see, by an argument similar to the one above, that $D' \models \neg\chi'(\vec{a},\vec{b})$ and thus $\exists \vec{w} \in adom.Q_1 y_1 \dots Q_n y_n.(\alpha(\vec{a},\vec{y}) \vee \gamma(\vec{a};\vec{w}))$ fails in D', which is impossible. This finishes the proof of the claim.

Thus, $D \models \varphi(\vec{a}) \rightarrow \exists \vec{w} \in adom.\gamma(\vec{a};\vec{w})$ holds in $\mathcal{M}'$, and all we need to show to prove safety of φ is that γ is algebraic in $\mathcal{M}'$. From Proposition 4.2 we know that $\forall \vec{z}\forall \vec{w}.\gamma(\vec{z};\vec{w}) \leftrightarrow (\bigwedge_i \beta_\Xi(z_i;\vec{w}) \wedge \gamma(\vec{z};\vec{w}))$ holds in $\mathcal{M}$ for an appropriately chosen Ξ, and thus in $\mathcal{M}'$. Since both o-minimality and density are preserved under elementary equivalence, we get that β_Ξ is algebraic in $\mathcal{M}'$, and hence γ is algebraic in $\mathcal{M}'$, too. $\square$

5. Deciding safety of conjunctive queries and relatives. Safety of arbitrary calculus queries is undecidable even in the pure case [47], and of course it remains undecidable when interpreted functions are present. The main goal of this section is to show that safety is *decidable* for Boolean combinations of conjunctive queries in the presence of an interpreted structure such as $\langle \mathbb{R},+,*,0,1,<\rangle$. In particular, safety of FO + Poly and FO + Lin conjunctive queries is decidable.

Recall that we are using CQ, UCQ, and BCCQ for conjunctive, unions of conjunctive, and Boolean combinations of conjunctive queries (see section 2 for their definition in the presence of an interpreted structure). CQs, having dominated early research in relational theory because of their nice properties, resurfaced recently in a number of new applications; cf. [24, 19, 40]. Our proof will be by reduction to the containment problem, which is decidable for UCQs over certain structures. (Of course, *without* an interpreted structure, this is well known [35], as is the decidability of safety for BCCQs; cf. [1].) Note that CQs and UCQs are monotone (that is, $D \subseteq D'$ implies $\varphi[D] \subseteq \varphi[D']$). Since there are nonmonotone BCCQs, the class of UCQs is strictly contained in the class of BCCQs.

The main result is Theorem 5.1.

THEOREM 5.1. *Let $\mathcal{M}$ be o-minimal, based on a dense order, and decidable, and let it admit effective QE. Then it is decidable if a given BCCQ $\varphi(\vec{x})$ over $\mathcal{M}$ is safe.*

The proof is contained in the following two lemmas, which are of independent interest, and will be used later in section 7. Recall that by containment $\varphi \subseteq \psi$ we mean $\varphi[D] \subseteq \psi[D]$ for any D.

LEMMA 5.2. *Let $\mathcal{M}$ be o-minimal and based on a dense order, and $\varphi(\vec{x})$ be a first-order query. Then there exists an active-semantics CQ $\psi(\vec{x})$ such that φ is safe iff $\varphi \subseteq \psi$.*

Proof. The proof follows from Proposition 4.9: take $\Gamma = \{\gamma_1(x;\vec{y}), \ldots, \gamma_k(x;\vec{y})\}$ given by the proposition; let $\gamma = \bigvee_i \gamma_i$ and let $\psi(x_1, \ldots, x_n)$ be

$$\exists \vec{y}_1 \in adom \ldots \exists \vec{y}_n \in adom\ \gamma(x_1; \vec{y}_1) \wedge \ldots \wedge \gamma(x_n; \vec{y}_n).$$

If $\varphi \subseteq \psi$, then φ is safe since all γ_i's are algebraic. If φ is safe, then $adom(\varphi[D]) \subseteq \Gamma(D)$ for every D, which implies $\varphi \subseteq \psi$. ∎

LEMMA 5.3. *Let $\mathcal{M}$ be as in Theorem* 5.1. *Then containment of a BCCQ in a UCQ is decidable; that is, for a BCCQ $\varphi(\vec{x})$ and a UCQ $\psi(\vec{x})$ it is decidable if $\varphi \subseteq \psi$. This continues to hold if both φ and ψ are active-semantics queries.*

Proof. We start with the following claim.

CLAIM 5.4. *Given φ and ψ, one can effectively find a number k such that $\varphi \subseteq \psi$ iff for every database D with at most k tuples, $\varphi[D] \subseteq \psi[D]$.*

This clearly implies the result, as the latter condition can be expressed as an $L(\Omega)$ sentence. For example, if the schema contains one relational symbol S, this sentence is $\forall \vec{x}_1 \ldots \vec{x}_k \forall \vec{x}.\ \varphi(\vec{x})[\{\vec{x}_i\}/D] \rightarrow \psi(\vec{x})[\{\vec{x}_i\}/D]$, where $\varphi(\vec{x})[\{\vec{x}_i\}/D]$ is obtained from $\varphi(\vec{x})$ by replacing each occurrence of $S(\vec{z})$ by $\bigvee_i(\vec{z} = \vec{x}_i)$, and similarly for an arbitrary schema. The decidability of $\mathcal{M}$ now implies the lemma.

The proof of the claim proceeds similarly to [19]. Note that every BCCQ α can be represented as $\bigvee_{i=1}^{n}(\chi_i(\vec{x}) \wedge \neg\xi_i(\vec{x}))$, where each χ_i is a CQ and each ξ_i is a UCQ; this follows if one writes α as a disjunctive normal form (DNF). We take k to be the maximum length of χ_i (measured as the sum of the number of atomic formulae and the number of quantified variables).

Assume that $\varphi[D] \not\subseteq \psi[D]$ for some D; that is, we have $\vec{a} \in \varphi[D] \not\subseteq \psi[D]$. Assume that $D \models \chi_i(\vec{a}) \wedge \neg\xi_i(\vec{a})$ and let $\chi_i(\vec{x}) = \exists \vec{y} \exists \vec{z} \in adom\ \bigwedge_{j=1}^{l} \alpha_j(\vec{x}, \vec{y}, \vec{z})$. Then, for some $\vec{b}$ over $\mathcal{U}$ and $\vec{c}$ over $adom(D)$, we get $D \models \bigwedge_{j=1}^{l} \alpha_j(\vec{a}, \vec{b}, \vec{c})$. Consider those α_j's which are SC-formulae. For each such α_j, which is of the form $R(\cdots)$ where $R \in SC$, there is a tuple in D that satisfies it. Select one such tuple for each SC-atomic α_j, and let D' be D restricted to those tuples. Choose a set of at most $length(\vec{c})$ tuples in D containing all the components of $\vec{c}$, and add it to D'. Let the resulting database be D''. Clearly, it has at most k tuples.

Note that $D'' \models \bigwedge_{j=1}^{l} \alpha_j(\vec{a}, \vec{b}, \vec{c})$, and thus $D'' \models \chi_i(\vec{a})$ since $\vec{c} \subseteq adom(D'')$. On the other hand, $D'' \models \neg\xi_i(\vec{a})$ by monotonicity of ξ_i. Thus, we get that $\vec{a} \in \varphi[D''] - \psi[D'']$, where D'' has at most k tuples. This implies that each counterexample to containment is witnessed by a $\leq k$-element database, and finishes the proof Lemma 5.3. ∎

To complete the proof of Theorem 5.1, note that under the assumption on $\mathcal{M}$, the

CQ ψ such that $\varphi \subseteq \psi$ is equivalent to the safety of φ can be constructed effectively; this follows from the procedure for constructing Γ given in Proposition 4.9. The theorem now follows from Lemma 5.3. ∎

COROLLARY 5.5. *It is decidable whether any Boolean combination of* FO + LIN *or* FO + POLY *conjunctive queries is safe.* ∎

Note, however, that safety of CQs is not decidable over every structure. For example, for $\langle \mathbb{N}, +, *, 0, 1, < \rangle$, decidability of CQ safety would imply decidability of checking whether a Diophantine equation has finitely many solutions, which is known to be undecidable [12].

6. Dichotomy theorem and outputs of queries. The main result of this section is a simple but powerful combinatorial structure theorem, saying that over a well-behaved structure, outputs of safe queries cannot grow arbitrarily large in terms of the size of the input. In fact, we prove a *dichotomy* result: either a query φ is not safe on D, or $\varphi[D]$ is at most polynomial in the size of D, where the bounding polynomial depends only on φ. This result shows tame behavior of relational calculus queries over some important interpreted structures, in particular those giving rise to linear, polynomial, and exponential constraints. It can be used to show negative results, that is, new expressivity bounds, as well as positive results: the dichotomy theorem is a key ingredient in the decidability results of the next section.

We use the notation $\mathsf{size}(D)$ for the size of a database, measured here as the number of tuples. It can equivalently be measured as the cardinality of the active domain, or the number of tuples multiplied by their arity, and all the results will hold.

THEOREM 6.1 (dichotomy theorem). *Let $\mathcal{M}$ be o-minimal and based on a dense order. Let $\varphi(\vec{x})$ be a first-order query. Then there exists a polynomial $p_\varphi : \mathbb{R} \to \mathbb{R}$ such that, for any database D, either $\varphi[D]$ is infinite or $\mathsf{size}(\varphi[D]) \le p_\varphi(\mathsf{size}(D))$.*

Proof. Consider Γ given by Proposition 4.9. Since each $\gamma \in \Gamma$ is algebraic, there exists, by Lemma 4.1, a number c_γ such that $card(\{x \mid \gamma(x;\vec{y})\}) < c_\gamma$ for every $\vec{y}$. Thus, if $adom(\varphi[D])$ is finite, then its cardinality is at most

$$\sum_{\gamma\in\Gamma} c_\gamma \cdot n^{m_\gamma},$$

where m_γ is the number of $\vec{y}$ variables in γ and n is the size of the active domain of D. This is clearly bounded by $card(\Gamma) \cdot C_\Gamma \cdot n^{M_\Gamma}$, where $C_\Gamma = \max c_\gamma$ and $M_\Gamma = \max m_\gamma$.

Now notice that for every schema SC, there exist constants $c_0, d_0, c_1 > 0$ such that $c_1 \cdot card(adom(D)) \le \mathsf{size}(D) \le c_0 \cdot card(adom(D))^{d_0}$ for every D. Hence, for appropriately chosen $c_0, d_0 > 0$, we obtain that if $\varphi[D]$ is finite, then

$$\mathsf{size}(\varphi[D]) \le c_0 \cdot card(adom(\varphi[D]))^{d_0} \le c_0 \cdot (card(\Gamma) \cdot C_\Gamma \cdot n^{M_\Gamma})^{d_0} \le cn^d,$$

where $n = card(adom(D))$ and c, d are constants that depend only on φ. Hence, for some $c_1 > 0$ that depends on the schema only, we have $\mathsf{size}(\varphi[D]) \le c \cdot (\frac{n}{c_1})^d$, which proves the theorem. ∎

Are the assumptions on a structure important for the dichotomy result? That is, can we find structures over which it fails? The following is a simple counterexample to the dichotomy theorem: Let $\mathcal{M} = \langle \mathbb{N}, < \rangle$. Let $\varphi(x)$ be the following active-semantics query: $\exists y \in adom. x < y$. Clearly, $\varphi(x)$ is safe, but $\mathsf{size}(\varphi[D])$ can be arbitrarily large even for a database whose active domain consists of just one element.

The dichotomy theorem can also be stated in terms of a function measuring the growth of the output size. Formally, given a query φ, define $growth_\varphi : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$

as

$$growth_\varphi(n) = \max\{\mathsf{size}(\varphi[D]) \mid \mathsf{size}(D) = n\}.$$

COROLLARY 6.2. *Let $\varphi(\vec{x})$ be an* FO + LIN, *or* FO + POLY, *or* FO + EXP *query. Then there exists a polynomial p_φ such that, for every $n \in \mathbb{N}$, either $growth_\varphi(n) = \infty$ or $growth_\varphi(n) \leq p_\varphi(n)$.* □

Note that for the query $\varphi(x)$ over $\langle \mathbb{N}, < \rangle$, which we used as a counterexample to the dichotomy theorem, $growth_\varphi(n) = \infty$ for all $n > 0$. Thus, we have a question whether Corollary 6.2 fails over some structures. The following proposition provides an example.

PROPOSITION 6.3. *Let $\mathcal{M} = \langle \mathbb{N}, +, <, 1 \rangle$. Then there exists an active-semantics first-order query $\varphi(x)$ over $\mathcal{M}$ such that $growth_\varphi(n) = 2^n$ for every $n > 0$.*

Proof. Let SC consist of one unary relation S. We show that there exists an FO$(\mathcal{M}, SC)$ sentence Ψ such that $S \models \Psi$ iff S is of the form $S_n = \{2^i \mid 1 \leq i \leq n\}$. This is done by letting Ψ be

$$\begin{array}{ll} & (\exists x \in adom. x = 1 + 1 \wedge S(x)) \\ \wedge & (\forall x \in adom. x = 1 + 1 \vee x > 1 + 1) \\ \wedge & (\forall x \in adom. x = 1 + 1 \vee \exists y \in adom. y + y = x) \\ \wedge & (\forall x \in adom. (\forall y \in adom. y < x \vee y = x) \vee (\exists y \in adom. y = x + x)). \end{array}$$

Now define $\varphi(x)$ as $\Psi \wedge \neg(x < 1) \wedge (\exists y \in adom. x < y \vee x = y)$. Then, for S not of the form S_n, we have $\varphi[S] = \emptyset$, and $\varphi[S_n] = \{1, 2, 3, \ldots, 2^n\}$. Since $card(S_n) = n$, this implies $growth_\varphi(n) = 2^n$ for $n > 0$. □

The dichotomy theorem gives easy expressivity bounds based on the growth of the output size, in fact, sometimes somewhat surprising ones: even if we use exponentiation, we still cannot express any queries with superpolynomial growth. For example, consider the following query Q: given a binary relation S containing $n+1$ distinct points $x_0, x_1, \ldots, x_n$ on a plane, return the vertices of the projection of an n-dimensional cube $[0,1]^n$, where the edges along the axes are projected onto $\vec{x_0 x_1}, \ldots, \vec{x_0 x_n}$. It is easy to see that for each fixed n, this query is expressible in FO + POLY. As a consequence of the dichotomy theorem, we conclude that Q cannot be expressed uniformly for all n even as an FO + EXP query.

For monotone queries, we can do better. We prove a trichotomy theorem that provides us with a lower bound as well. Recall that monotone queries are those for which $D_1 \subseteq D_2$ implies $\varphi[D_1] \subseteq \varphi[D_2]$. For example, any union of CQs is monotone.

THEOREM 6.4. *Let $\mathcal{M}$ be o-minimal based on a dense order. Then, for each monotone query $\varphi(\vec{x})$, there exist two polynomials p^1_φ and p^2_φ such that either $growth_\varphi$ is bounded by a constant or, for every n, either $p^1_\varphi(n) \leq growth_\varphi(n) \leq p^2_\varphi(n)$ or $growth_\varphi(n) = \infty$.*

Proof. In view of the previous theorem, it remains to show that if $growth_\varphi$ is not bounded by a constant, then it is bounded below by a polynomial. Assuming $growth_\varphi$ is not bounded by a constant, we get a family of databases $\{D_i\}_{i \in \mathbb{N}}$ such that $\mathsf{size}(\varphi[D_i]) > i$ for all i. Because of monotonicity, we can ensure, by adding elements to D_i, that $\mathsf{size}(D_i) \geq i$.

In the proof of Lemma 5.3 we showed that there exists a constant k that depends on φ only, such that $\vec{a} \in \varphi[D]$ iff there is a $D' \subseteq D$ with at most k tuples such that $\vec{a} \in \varphi[D']$. Thus, there is a $D'_i \subseteq D_i$ with at most $k(i+1)$ tuples such that $\mathsf{size}(\varphi[D'_i]) > i$.

Now, for a given n, let i be the maximal such that $k(i+1) \leq n$. Consider D'_i and extend it to contain exactly n tuples. For the resulting D''_i, we have $\mathsf{size}(\varphi[D'_i]) > i$ by monotonicity; hence $growth_\varphi(n) > i$. Since $n \leq k(i+1)$, we get from this that $growth_\varphi(n) \geq \frac{n}{k} - 1$, which completes the proof. □

It also follows from the proof that the lower polynomial bound does not require o-minimality. This gives us some new expressivity bounds. It is possible to find first-order queries with $growth_\varphi = O(f(n))$ for many nonpolynomial functions f. For example, consider a schema consisting of one unary relation X and one binary relation E, and a sentence Ψ saying that E codes the powerset of X; that is, the family of sets $X_a = \{y \mid E(y,a)\}$ is exactly the family of all nonempty subsets of X, when a ranges over the second projection of E. Such a sentence Ψ can be defined in first-order logic; cf. [1]. We now let $\varphi(x) \equiv X(x) \wedge \Psi$; it then follows that $growth_\varphi = O(\log n)$. Similarly, one can find queries with $growth_\varphi = O(\sqrt[k]{n})$ for any constant k. The trichotomy theorem says that such queries cannot be defined as monotone queries (e.g., unions of CQs) over *any* interpreted structure.

7. Preserving geometric properties of constraint databases. In this section, we switch from the finite world to the infinite; that is, we deal with constraint databases that represent potentially infinite objects. The notion of safety over constraint databases is different: we are interested in identifying languages that guarantee *preservation of certain geometric properties.*

To give a very simple example, assume that spatial objects stored in a database are convex polytopes in $\mathbb{R}^n$. A simple query, "return the convex hull of all the vertices x with $\| x \| < 1$" does always return a convex polytope. This query must be written in a rather expressive language: it can be expressed in FO + Poly but not FO + Lin [43]. Now, our question is, Can we ensure in some way that a class of FO + Poly programs preserves a given property, like being a convex polytope? That is, can we find an effective syntax for the class of queries that preserve certain geometric properties?

For FO + Poly and the class of databases definable with linear constraints (semilinear databases), [13] gave a solution, based on deciding semilinearity by an FO + Poly query. The resulting language is not quite natural, and [13] posed a problem of finding natural languages that capture queries with certain preservation properties. Our first goal here is to present a general scheme, different from the decidability approach, for enumerating such queries in FO($\mathcal{M}$). Here $\mathcal{M}$ is some structure on the reals, not necessarily $\langle \mathbb{R}, +, *, 0, 1, < \rangle$. The approach is based on reduction to the finite case and using our results about finite query safety. A similar approach was used in [37], where a coding was applied to reduce certain questions about ordered constraint databases to ones about finite databases.

As it often happens, the general case is solved rather easily, and gives us a pleasant characterization of queries preserving geometric properties, but working out the details of important motivating examples is a painful process. We do so for three properties: a convex polytope, a convex polyhedron, and a compact semilinear set in $\mathbb{R}^2$ (the latter are perhaps the most often-encountered class of constraint databases).

We then use our characterizations together with the dichotomy theorem of the previous section to show a somewhat surprising result that for unions of conjunctive FO + Poly queries, it is *decidable* whether they preserve convex polytopes or compact semilinear sets in $\mathbb{R}^2$.

To define a general framework for talking about queries that preserve geometric properties, we recall some basic definitions on constraint (or finitely representable) databases. As before, we have a language of some underlying structure $\mathcal{M}$ and a schema SC, but now m-relations in SC are given by quantifier-free formulae[2] $\alpha(x_1, \ldots, x_m)$ in $L(\Omega)$. If $\mathcal{M}$ is $\langle \mathbb{R}, +, -, 0, 1, < \rangle$, then sets so defined are called *semi-linear*; for $\langle \mathbb{R}, +, *, 0, 1, < \rangle$ they are called *semialgebraic*; cf. [42]. The query languages for constraint databases are the same as those we considered for finite ones: $\mathrm{FO}(SC, \Omega)$.

If $\mathcal{M} = \langle \mathcal{U}, \Omega \rangle$ is an infinite structure, let $\mathsf{Obj}(\Omega)$ be the class of finitely representable databases over $\mathcal{M}$; that is, $\mathsf{Obj}(\Omega) = \bigcup_{n<\omega} \mathsf{Obj}_n(\Omega)$ and $\mathsf{Obj}_n(\Omega)$ is the collection of subsets of $\mathcal{U}^n$ of the form $\{(x_1, \ldots, x_n) \mid \mathcal{M} \models \alpha(x_1, \ldots, x_n)\}$, where α is a quantifier-free first-order formula in $L(\Omega)$. We use SAlg_n for semialgebraic sets.

Let S be an m-ary relational symbol, and let $\psi(y_1, \ldots, y_n)$ be a first-order formula in the language of S and Ω. Then this query defines a map from $\mathsf{Obj}_m(\Omega)$ to $\mathsf{Obj}_n(\Omega)$ as follows: for any $X \in \mathsf{Obj}_m(\Omega)$, $\psi[X] = \{\vec{y} \mid (\mathcal{M}, X) \models \psi(\vec{y})\}$. Clearly $\psi[X] \in \mathsf{Obj}(\Omega)$ if $\mathcal{M}$ has QE.

Let $\mathcal{C}$ be a class of objects in $\mathsf{Obj}(\Omega)$. We say that a first-order query ψ *preserves* $\mathcal{C}$ if for any $X \in \mathcal{C}$, $\psi[X] \in \mathcal{C}$. For example, $\mathcal{C}$ can be the class of convex polytopes in SAlg.

Thus, the safety question for constraint databases is the following. Is there an effective syntax for the class of $\mathcal{C}$-preserving queries? Below, we show an approach to solution, based on the characterization theorems for the finite case.

DEFINITION 7.1. *The class $\mathcal{C}$ has a canonical representation in $\mathsf{Obj}(\Omega)$ if there is a recursive injective function $g : \mathbb{N} \to \mathbb{N}$ with computable inverse, and for each n, two functions, $code_n : 2^{\mathcal{U}^n} \to 2^{\mathcal{U}^m}$ and $decode_n : 2^{\mathcal{U}^m} \to 2^{\mathcal{U}^n}$, where $m = g(n)$, such that*

(1) *$decode_n \circ code_n(x) = x$ if $x \in \mathsf{Obj}_n(\Omega)$;*

(2) *$\mid code_n(x) \mid < \omega$ if $x \in \mathcal{C}$; $decode_n(x) \in \mathcal{C}$ if x is finite;*

(3) *$code_n$ is $\mathrm{FO}(\Omega)$-definable on $\mathsf{Obj}_n(\Omega)$;*

(4) *$decode_n$ is $\mathrm{FO}(\Omega)$-definable on finite sets.*

Intuitively, the canonical representation is a finite representation of $\mathcal{C}$ within $\mathsf{Obj}(\Omega)$ that can be defined in first-order logic over $\mathcal{M}$. For example, an approach to obtaining a canonical representation of convex polytopes would be to compute their vertices. This suffices to reconstruct the polytope, and the vertices can be defined by a first-order formula. The actual representation (Proposition 7.3) is indeed based on computing the vertices.

Next, we show that canonical representations solve the safety problem. We always assume that the set Ω is recursive.

THEOREM 7.2. *Let $\mathcal{M} = \langle \mathcal{U}, \Omega \rangle$ be o-minimal, based on a dense order, and decidable, and let it have effective QE. Suppose $\mathcal{C}$ is a class that has a canonical representation in $\mathsf{Obj}(\Omega)$. Then there is an effective syntax for $\mathcal{C}$-preserving $\mathrm{FO}(\Omega)$ queries; that is, there exists a recursively enumerable set of $\mathcal{C}$-preserving $\mathrm{FO}(\Omega)$ queries such that every $\mathcal{C}$-preserving $\mathrm{FO}(\Omega)$ query is equivalent to a query in this set.*

Proof. Consider an enumeration of all safe $\mathrm{FO}(\Omega)$ queries $\langle \varphi_i \rangle$ (from Corollary 4.8, we know that it exists). Let φ use the extra relation symbol of arity m and assume that n is such that $g(n) = m$; given the assumptions, we can compute that. Let φ_i have l parameters, and again let k be such that $g(k) = l$. If n and k are found for a given φ_i, we let ψ be

$$decode_k \circ \varphi_i \circ code_n.$$

This produces the required enumeration. So we have to check that every query above

[2]Without loss of generality, we do not assume relational attributes, as in [13, 27] and some other papers. They do not affect our results but would make notation heavier.

preserves $\mathcal{C}$, and for every $\mathcal{C}$ preserving ψ, we can get φ such that $decode \circ \varphi \circ code$ coincides with ψ. The first one is clear: if we have $X \in \mathcal{C}$, then $code_n(X)$ is finite, hence $\varphi_i[code_n(X)]$ is finite too, and applying $decode_k$, we get an object in $\mathcal{C}$.

For the converse, suppose we have a $\mathcal{C}$-preserving query $\psi : \mathsf{Obj}_n(\Omega) \to \mathsf{Obj}_k(\Omega)$. Define α as follows: $\alpha = code_k \circ \psi \circ decode_n$. That is, α is a query $\mathsf{Obj}_m(\Omega) \to \mathsf{Obj}_l(\Omega)$. Given this, notice that

$$decode_k \circ \alpha \circ code_n = decode_k \circ code_k \circ \psi \circ decode_n \circ code_n = \psi$$

on $\mathsf{Obj}_n(\Omega)$. Thus, it remains to show that α is safe, i.e., preserves finiteness. Let X be a finite set in $\mathcal{U}^m$. Then $decode_n(X) \in \mathcal{C}$, $decode_n(X) \subset \mathcal{U}^n$. Since ψ is $\mathcal{C}$-preserving, we get that $Y = \psi[decode_n(X)] \in \mathsf{Obj}_k(\Omega)$ is in $\mathcal{C}$, too, and thus $code_k(Y)$ is finite. This proves finiteness of α and concludes the proof of the theorem. ◻

We now turn to examples in the case when $\Omega = (+, *, 0, 1, <)$; that is, we are looking for canonical representations in SAlg. Let $\mathcal{CPH}$ be the class of convex polyhedra (intersections of a finite number of closed halfspaces) and $\mathcal{CPT}$ be the class of convex polytopes (bounded polyhedra). For the basic facts on convex sets that will be used in the proofs of the propositions below, see [33].

PROPOSITION 7.3. *The class $\mathcal{CPT}$ has canonical representation in* SAlg.

Proof. Given a convex polytope X in $\mathbb{R}^n$, its vertices can be found as $V(X) = \{\vec{x} \in \mathbb{R}^n \mid \vec{x} \in X, \vec{x} \notin \mathrm{conv}(X - \vec{x})\}$. Thus, vertices of convex polytopes are definable in FO$(+, *, 0, 1, <)$, because the convex hull of a finite set of points is definable, and, in view of Carathéodory's theorem, we have

$$V(X) = \{\vec{x} \in \mathbb{R}^n \mid \vec{x} \in X, \forall \vec{x}_1, \ldots, \vec{x}_{n+1} \in X - \vec{x}.\ \vec{x} \notin \mathrm{conv}(\{\vec{x}_1, \ldots, \vec{x}_{n+1}\})\}.$$

We now define $code_n$. To simplify the notation, we let it produce a pair of n-ary relations, but it can be straightforwardly coded by one relation. If $X = \mathrm{conv}(V(X))$, then $code_n(X) = (V(X), \emptyset)$; otherwise, $code_n(X) = (\mathbb{R}^n, X)$. The function $decode_n : 2^{\mathbb{R}^n} \times 2^{\mathbb{R}^n} \to 2^{\mathbb{R}^n}$ is defined as follows:

$$decode_n(Y, Z) = \begin{cases} \bigcup_{(\vec{y}_1, \ldots, \vec{y}_{n+1}) \in Y} \quad \mathrm{conv}(\{\vec{y}_1, \ldots, \vec{y}_{n+1}\}) & \text{if } Y \neq \mathbb{R}^n, \\ \hfill Z & \text{otherwise.} \end{cases}$$

Clearly, $decode_n \circ code_n$ is the identity function for any (semialgebraic) set; these functions are also first-order definable. If X is a polytope, $V(X)$ is finite, and by Carathéodory's theorem each point of X is contained in the convex hull of at most $n + 1$ vertices of X. Hence, $card(code_n(X)) \leq card(V(X))^{n+1} < \omega$. If (Y, Z) is finite, then $decode_n(Y)$ is $\mathrm{conv}(Y)$ and thus a convex polytope. This proves the proposition. ◻

PROPOSITION 7.4. *The class $\mathcal{CPH}$ has canonical representation in* SAlg.

Proof. We first give a brief sketch of the coding scheme. We start by recalling a few basic facts about convex polyhedra (see [15, 33]). Let X be a convex polyhedron in $\mathbb{R}^n$. Then $X = L + (X \cap L^{\perp})$, where L is its lineality space, defined as $\{\vec{y} \mid \vec{y} = \vec{0} \text{ or } \forall \vec{x} \in X \forall \lambda.\ \vec{y} + \lambda \cdot \vec{x} \in X\}$ (it is a subspace of $\mathbb{R}^n$) and $L^{\perp}$ is the orthogonal subspace $\{\vec{y} \mid \forall \vec{x} \in L.\ \langle \vec{x}, \vec{y} \rangle = 0\}$. We shall use X_0 for $X \cap L^{\perp}$ in this proof. It is known that X_0 is a convex polyhedron of lineality zero; that is, it contains no line. By $A + B$ we mean $\{\vec{a} + \vec{b} \mid \vec{a} \in A, \vec{b} \in B\}$. Note the difference between the translate $X - \vec{x} = \{\vec{y} - \vec{x} \mid \vec{y} \in X\}$ and the set-theoretic difference $X - x$; we use $\vec{x}$ to distinguish between them.

For X_0, define its vertices as $x \in X_0$ such that $x \notin \mathrm{conv}(X_0 - x)$. A direction is given by a vector $\vec{y}$ and corresponds to the equivalence class of rays which are translates of each other. Note that each direction can be canonically represented by $\vec{y}$ such that $\parallel \vec{y} \parallel = 1$. A direction $\vec{y}$ is an extreme direction of X_0 if for some vertex $\vec{x}$, the ray through $\vec{x}$ in the direction of $\vec{y}$, $l(\vec{x}, \vec{y}) = \{\vec{x} + \lambda \cdot \vec{y} \mid \lambda \geq 0\}$, is a face of X_0. Since X_0 is polyhedral of lineality zero, the set of vertices and extreme directions is finite. By (generalized) Carathéodory's theorem [15, 33], every point $\vec{z}$ of X_0 is a combination of at most $n+1$ vertices and extreme directions,

$$\lambda_1 \vec{x}_1 + \cdots + \lambda_k \vec{x}_k + \mu_1 \vec{y}_1 + \cdots + \mu_m \vec{y}_m,$$

where $\lambda_i, \mu_j \geq 0, \lambda_1 + \cdots + \lambda_k = 1, k + m \leq n + 1$.

This suggests the following coding scheme. As before, for simplicity of exposition, we assume several coding relations, but they can be combined into one easily. We also do not spell out every first-order formula, but the reader should be convinced from the mathematical definitions that all the concepts we use are first-order definable over the real field. We use relations $LINEAL_k$; each such relation contains a canonical representation (roughly, an orthogonal basis) of the lineality space of X, provided its dimension is k. That is, at most one of these relations actually contains some information. We then have the relations *Vert* and *ExtDir* for storing vertices and extreme directions of X_0. Finally, we have a relation *Points* that contains points that do not belong to $L + X_0$ (recall that the coding scheme applies to any semialgebraic set, so there could be such points, and we need to record them for the *decode* function).

Thus, to code (an arbitrary semialgebraic) set X, we first note that its lineality space $L(X) = \{\vec{y} \mid \forall \vec{x} \in X.\ \vec{x} + \vec{y} \in X\}$ and its orthogonal $L(X)^{\perp} = \{\vec{y} \mid \forall \vec{x} \in L(X).\ \langle \vec{x}, \vec{y} \rangle = 0\}$ are definable in FO + Poly (note that one can define the inner product in FO + Poly). Furthermore, for each $k \leq n$, there exists an FO + Poly sentence dim_k expressing the fact that $L(X)$ is a subspace of $\mathbb{R}^n$ and its dimension is k. This is true because in FO + Poly we can test linear independence; thus, we can check if there exists a system of k linearly independent vectors in L such that every vector in L is a linear combination of them.

Next, we show how to compute $LINEAL_k$ and *VertDir*. We first sketch the coding scheme for $LINEAL_k$. The set $L(X)$ is (FO + Poly)-definable. Assume that it is a k-dimensional linear space (which is tested by $\dim_k$). Let Δ_n be some canonically chosen n-dimensional simplex of diameter 1 such that the origin has barycentric coordinates $(\frac{1}{n}, \ldots, \frac{1}{n})$. Consider the intersection of $L(X)$ with one-dimensional faces of Δ_n (unless $L(X)$ is a line, in which case we consider its intersection with two-dimensional faces of Δ_n). If the intersection is a point, we record that point; if it contains the whole face, we record both endpoints of the face. From the selected points, find a linearly independent subsystem (note that it can be done canonically, for example, by listing the vertices and one-dimensional faces of Δ_n in some order). It then serves as a basis of $L(X)$, which we use to code $L(X)$. Note that $L(X)$ can be reconstructed in FO + Poly from its basis.

Now that we have a representation for the lineal space of X, and a first-order formula defining $L^{\perp}$, we have an FO + Poly formula defining X_0. Using it, we can compute vertices

$$V(X_0) = \{x \in X_0 \mid \neg \exists x_1, \ldots, x_{n+1} \in X_0 - x.\ x \in \mathrm{conv}(\{x_1, \ldots, x_{n+1}\})\}.$$

Clearly, this is a first-order definition. Next, we find the set

$$E(X_0) = \{\vec{y} \mid \langle \vec{y}, \vec{y} \rangle = 1 \text{ and } \exists \vec{x} \in V(X_0).\ l(\vec{x}, \vec{y}) \text{ is a face}\}.$$

A subset Y of X_0 is a face if every closed line segment in X_0 with a relative interior point in Y has both endpoints in Y. Clearly, this is first-order definable, and thus $E(X_0)$, the set of extreme directions of X_0, is first-order definable.

Given two sets V and E in $\mathbb{R}^n$, by $\text{conv}(V, E)$ we denote their convex hull, that is, the set of elements of $\mathbb{R}^n$ definable as $\sum_{i=1}^{k} \lambda_i \cdot \vec{x}_i + \sum_{j=1}^{m} \mu_j \cdot \vec{y}_j$, where $\vec{x}_i \in V$, $\vec{y}_j \in E$, $\sum_{i=1}^{k} \lambda_i = 1$, $\lambda_i, \mu_j \geq 0$, and $k + m \leq n + 1$. Again, this can be done in $\text{FO} + \text{POLY}$.

We now describe code_n. For a semialgebraic set X, it produces a tuple of relations

$$(\mathit{LINEAL}_0, \ldots, \mathit{LINEAL}_n, \mathit{Vert}, \mathit{ExtDir}, \mathit{Point})$$

as follows. It first determines, by computing $L(X)$, $L(X)^{\perp}$, $V(X_0)$, and $E(X_0)$ if it is the case that $L(X)$ is a linear subspace of $\mathbb{R}^n$ and

$$X \;=\; L(X) + \text{conv}(V(X_0), E(X_0)).$$

If this is the case, then LINEAL_k, *Vert*, and *ExtDir* are produced as before, and *Point* is empty. Otherwise, *Point* coincides with X, and all other sets in the coding are taken to be $\mathbb{R}^n$. From the description above it follows that code_n is $(\text{FO} + \text{POLY})$-definable.

To compute decode_n, we first check if the first $n+2$ relations in the code coincide with $\mathbb{R}^n$, and, if this is the case, output the last relation in the code. Otherwise, we use the nonempty LINEAL_k with least k to compute a linear subspace L of $\mathbb{R}^n$ generated by the vectors in LINEAL_k (if all LINEAL_k are empty, we let this subspace be $\{\vec{0}\}$). Next, compute $Y = \text{conv}(\mathit{Vert}, \mathit{ExtDir})$. Note that both are $(\text{FO} + \text{POLY})$-definable. Finally, return $L + Y \cap L^{\perp}$; this is $(\text{FO} + \text{POLY})$-definable also.

We now sketch the proof that this coding scheme satisfies the conditions of the definition of canonical representation. Both *code* and *decode* are $(\text{FO} + \text{POLY})$-definable. If $X \in \mathcal{CPH}$, then $L(X)$ is a linear space, X_0 has finitely many vertices and extreme directions, and $X = L + X_0$ implies that *Point* is empty, thus showing that *code* produces a finite set. Assume that *decode* is given a finite input Y. Then none of the first $n + 2$ relations is $\mathbb{R}^n$, and thus the output of *decode* is the sum of a vector space and a convex hull of a finite set of vertices and directions, and thus a convex polyhedron. To show that $\mathit{decode} \circ \mathit{code}(X) = X$ for any semialgebraic X, consider two cases. If $X = L(X) + \text{conv}(V(X_0), E(X_0))$, then *Point* is empty, and *Vert* and *ExtDir* record all vertices and extreme directions of X_0, and one of LINEAL_k codes the lineality space. Thus, *decode* applied to $\mathit{code}(X)$ will return $L + X_0 = X$. If $X \neq L(X) + \text{conv}(V(X_0), E(X_0))$, then all relations but *Point* coincide with $\mathbb{R}^n$, and *Point* contains X, and thus *decode* returns X. This completes the proof. $\square$

Let $\mathsf{SLinComp}$ be the class of *compact* (closed and bounded) semi-linear sets. We resolve this case for dimension 2.

PROPOSITION 7.5. *The class* $\mathsf{SLinComp}_2$ *has canonical representation in* SAlg_2.

Proof (sketch). An object in $\mathsf{SLinComp}_2$ is a finite union of convex polytopes in $\mathbb{R}^2$—this easily follows from cell decomposition. Any such object X admits a triangulation that does not introduce new vertices [26]. Thus, the idea of the coding is to find the set $V(X)$ of vertices and use as the code triples of vertices (not necessarily distinct) $(\vec{x}, \vec{y}, \vec{z})$ with $\text{conv}(\{\vec{x}, \vec{y}, \vec{z}\}) \subseteq X$. More precisely, a triple $(\vec{x}, \vec{y}, \vec{z})$ belongs to $\mathit{code}(X)$ if either $\vec{x}, \vec{y}, \vec{z} \in V(X)$ and $\text{conv}(\{\vec{x}, \vec{y}, \vec{z}\}) \subseteq X$, or $\vec{x} = \vec{y} = \vec{z}$ and there is no triple of elements of $V(X)$ whose convex hull is contained in X and contains $\vec{x}$. Thus, $\mathit{code}(X) \subseteq \mathbb{R}^6$. For *decode*, we use

$$\mathit{decode}(Y) = \bigcup_{(\vec{x}, \vec{y}, \vec{z}) \in Y} \text{conv}(\{\vec{x}, \vec{y}, \vec{z}\}).$$

Clearly, *decode* $\circ$ *code* is the identity, *decode* is first-order definable, and *decode*(Y) is compact and semilinear when Y is finite. Thus, it remains to show that $V(X)$ is finite and (FO + POLY)-definable. The former is well known (see [25]). For the first-order definition of $V(X)$ we use the following result from [25, 10]. Let X be a finite union of polyhedra (in $\mathbb{R}^n$) and let $B_\epsilon(\vec{x})$ be the ball of radius ϵ around $\vec{x}$. Then for each $\vec{x}$, there exists $\delta > 0$ such that for any $0 < \epsilon_1, \epsilon_2 < \delta$, we have

$$\vec{x} + \bigcup_{\lambda>0} \lambda \cdot [(X \cap B_{\epsilon_1}(\vec{x})) - \vec{x}] \quad = \quad \vec{x} + \bigcup_{\lambda>0} \lambda \cdot [(X \cap B_{\epsilon_2}(\vec{x})) - \vec{x}].$$

We denote this set by $X(\vec{x})$. Define the equivalence relation $\equiv_X$ by $\vec{y} \equiv_X \vec{z}$ if $X(\vec{y}) = X(\vec{z})$. Then the vertices of X are precisely the one-element equivalence classes of $\equiv_X$. It is routine to verify that the above can be translated into an (FO + POLY) definition of vertices. This completes the proof. $\square$

Note that the coding scheme used in the proof of Proposition 7.5 *cannot* be used in higher dimensions. We used the fact there is a triangulation of a two-dimensional polygon that does not introduce new vertices. However, in the three-dimensional case, there exist (nonconvex) polygons for which such a triangulation is impossible; cf. [34]. In fact, [34] shows that the problem of deciding if a three-dimensional polygon admits such a triangulation is NP-complete.

Summing up, we have the following theorem.

THEOREM 7.6. *There exists a recursively enumerable class of* FO + POLY *queries that captures the class of* $\mathcal{CPT}$- *(*$\mathcal{CPH}$- *and* $\mathsf{SLinComp}_2$-*, respectively) preserving queries.* $\square$

The coding technique given here gives more information, however, as shown in the next section.

7.1. Decidability results and geometric bounds.

While the classes of FO + POLY queries preserving certain properties have been shown to be recursively enumerable, in general, testing nontrivial preservation properties for arbitrary first-order queries is undecidable. For example, it is shown in [44] that it is undecidable whether an FO + POLY-query preserves semilinearity. Here, we show that for a restricted class of FO + POLY queries, unions of CQs, preserving two of the properties considered here is decidable. The proofs are based on the representation theorems of this section and the dichotomy theorem of the previous section. We first give the following bound on the behavior of conjunctive queries on convex polytopes.

LEMMA 7.7. *Let* $\varphi(x_1, \ldots, x_n)$ *be a union of* FO + POLY *CQs that mentions one* m*-ary relational symbol* S*. Then one can effectively find two numbers,* k *and* l*, such that* φ *is* $\mathcal{CPT}$*-preserving iff for every convex polytope* D *in* $\mathbb{R}^m$ *with at most* k *vertices, the output* $\varphi[D]$ *is a convex polytope with at most* l *vertices in* $\mathbb{R}^n$.

Proof. We can assume without loss of generality that $\varphi(\vec{x})$ is of the form

$$\bigvee_j \exists \vec{z} \bigwedge_i \alpha_{ij}(\vec{x}, \vec{z}),$$

where each α_{ij} is either $S(\cdots)$ or an $L(+, *, 0, 1, <)$ formula. Let k_0 be the maximal number of S-atomic formulae in a disjunct of φ. Then the argument made in the proof of Lemma 5.3 shows that for each $\vec{a} \in \varphi[D]$, there exists a subset of $D' \subseteq D$ with at most k_0 points such that $\vec{a} \in \varphi[D']$.

Now assume that D is a convex polytope in $\mathbb{R}^m$ and $V(D)$ is the set of its vertices. Then each element of D belongs to conv(V'), where V' is a subset of V of cardinality at most $m+1$. Now let $k' = k_0(m+1)$. Then it follows from monotonicity that for every $\vec{a} \in \varphi[D]$, there is a subset of $V' \subseteq V(D)$ of cardinality at most k such that $a \in \varphi[\text{conv}(V')]$; in particular,

$$\varphi[D] = \bigcup_{V' \subseteq V(D), card(V') \leq k'} \varphi[\text{conv}(V')].$$

We now set $k = 2k'$.

Next, consider the following FO + Poly query φ', which uses one m-ary relational symbol R. First, φ' constructs the convex hull of points in $\mathbb{R}^m$ which are in R. Then it applies φ to the result, to get a set Y. Finally, it returns $V(Y)$, that is, the set $\{y \in Y \mid y \notin \text{conv}(Y - y)\}$. Clearly, φ' is expressible in FO + Poly. From the dichotomy theorem, we know that there is a polynomial p with the following property: if R is finite and contains i points, then either $\varphi'[R]$ is infinite or it contains at most $p(i)$ points. We now let $l = \max\{p(i) \mid i = 1, \ldots, k\}$. Note that both k and l can be effectively calculated for a given φ.

It remains to show that if φ has the property that it sends a convex polytope with $\leq k$ vertices in $\mathbb{R}^m$ into a convex polytope with $\leq l$ vertices in $\mathbb{R}^n$, then it is $\mathcal{CPT}$-preserving. First, assume that D is a convex polytope with $\leq k$ vertices in $\mathbb{R}^m$. Assume $\varphi[D]$ is a convex polytope. If we apply φ' to a relation storing vertices of D, then the result is a finite set of vertices of $\varphi[D]$. Hence, by the dichotomy theorem, it has at most l vertices. That is, it now suffices to show that if φ has the property that it sends a convex polytope with $\leq k$ vertices in $\mathbb{R}^m$ into a convex polytope, then it is $\mathcal{CPT}$-preserving.

Let D be a convex polytope. We know that $\varphi[D] = \bigcup_{V'} \varphi[\text{conv}(V')]$, where V' ranges over subsets of $V(D)$ that have at most $k' \leq k$ elements. Thus, each $\varphi[\text{conv}(V')]$ is a convex polytope. Assume that $\varphi[D]$ is not convex. Then we can find two sets of vertices V_1, V_2, having at most k' elements each, and two points $\vec{a} \in \varphi[\text{conv}(V_1)]$, $\vec{b} \in \varphi[\text{conv}(V_2)]$, and $\vec{c}$ between $\vec{a}$ and $\vec{b}$ such that $\vec{c} \notin \varphi[D]$. Let $V_0 = V_1 \cup V_2$. Then, by monotonicity of φ, $\vec{a}, \vec{b} \in \varphi[\text{conv}(V_0)]$. By the assumption, $\varphi[\text{conv}(V_0)]$ is convex (since $card(V_0) \leq k$) and thus $\vec{c} \in \varphi[D]$. Hence, we showed that $\varphi[D]$ is convex. Since it is convex and a finite union of convex polytopes, it is a convex polytope itself. This completes the proof of Lemma 7.7. ∎

We now prove a similar bound for compact semilinear sets in $\mathbb{R}^2$. When we speak of a triangle, we mean convex hulls of three points in $\mathbb{R}^2$. In particular, a degenerate triangle can be a segment or a point. We now prove the following.

Lemma 7.8. *Let $\varphi(x, y)$ be a union of conjunctive* FO + Poly *queries that mentions one binary relational symbol S. Then one can effectively find two numbers, k and l, such that φ is $\mathsf{SLinComp}_2$-preserving iff for every set $D \subseteq \mathbb{R}^2$ which is a union of at most k triangles in $\mathbb{R}^2$, it is the case that $\varphi[D]$ is a union of at most l triangles in $\mathbb{R}^2$.*

Proof. Assume, as in the proof of Lemma 7.7, that $\varphi(\vec{x})$ is of the form

$$\bigvee_j \exists \vec{z} \bigwedge_i \alpha_{ij}(\vec{x}, \vec{z}),$$

where each α_{ij} is either $S(\cdots)$ or a $L(+, *, 0, 1, <)$ formula, and let k be the maximal number of S-atomic formulae in a disjunct of φ. Then, by the same argument as in the proof of Lemma 7.7, we obtain that if $\vec{x} \in \varphi[D]$, then there exist k points $\vec{z}_1, \ldots, \vec{z}_k$ in D such that $\vec{x} \in \varphi[\{\vec{z}_1, \ldots, \vec{z}_k\}]$. Assume that D is compact and semilinear; since

$D \subseteq \mathbb{R}^2$, it can be triangulated using only vertices of D. Let $V(D)$ be the set of vertices of D, which can be computed by an $\mathrm{FO}+\textsc{Poly}$ query, as shown in the proof of Proposition 7.5. Since every point of D is in the convex hull of a triangle whose vertices come from $V(D)$, we obtain, by monotonicity of φ, that if $\vec{x} \in \varphi[D]$, then there exists a set V' of triples of elements from $V(D)$ such that $\mathit{card}(V') \leq k$ and

$$\vec{x} \quad \in \quad \varphi\left[\bigcup_{(\vec{u},\vec{v},\vec{w})\in V'} \mathrm{conv}(\{\vec{u},\vec{v},\vec{w}\})\right].$$

Next, consider the following $\mathrm{FO}+\textsc{Poly}$ query ψ on finite databases. It uses one 6-ary schema relation R, which can be thought of as storing triples $(\vec{u},\vec{v},\vec{w})$ of points in $\mathbb{R}^2$. First, ψ computes

$$P(R) \;=\; \bigcup_{(\vec{u},\vec{v},\vec{w})\in R} \mathrm{conv}(\{\vec{u},\vec{v},\vec{w}\}),$$

which is a compact semilinear set, and then it computes the set $V(P(R))$ of vertices of $P(R)$, using the technique of [25], that we exploited in the proof of Proposition 7.5: for each $\vec{x}$, the set $P(R)(\vec{x}) = \vec{x}+[(P(R)\cap B_\epsilon(\vec{x}))-\vec{x}]$ does not depend on a particular value ϵ below some threshold δ. We then define vertices as those $\vec{x}$ for which there is no $\vec{x}_0 \neq \vec{x}$ with $P(R)(\vec{x}) = P(R)(\vec{x}_0)$. Thus, the query computing $V(P(R))$ is definable in $\mathrm{FO}+\textsc{Poly}$, and by the dichotomy theorem, there is a polynomial p such that, for each R, either $V(P(R))$ is infinite or it has at most $p(n)$ vertices, where n is the number of tuples in R. We now let l be $\max\{p(i) \mid i = 1,\ldots,k\}$.

To show that these k and l witness the conclusion of the lemma, assume that φ is $\mathsf{SLinComp}_2$-preserving. Then the output of every φ on every union of k triangles is a compact semilinear set. From the construction above, it follows that such an output can have no more than l vertices. Conversely, assume that the output of every union of k or fewer triangles is a union of l or fewer triangles. Since $\varphi[D]$ is the union of $\varphi[\bigcup_{(\vec{u},\vec{v},\vec{w})\in V} \mathrm{conv}(\{\vec{u},\vec{v},\vec{w}\})]$, where V ranges over k-element sets of triples of vertices of D, we obtain that $\varphi[D]$ is a union of a finite number of triangles, and thus compact and semilinear. This concludes the proof of the lemma. $\square$

The promised decidability results now follow from the bounds established in the lemmas above.

Theorem 7.9. *The following two properties of unions of conjunctive* $\mathrm{FO}+\textsc{Poly}$ *queries are decidable:*

(1) *being $\mathcal{CPT}$-preserving;*

(2) *being $\mathsf{SLinComp}_2$-preserving.*

Proof of (1). Note that for each i, there is an $\mathrm{FO}+\textsc{Poly}$ query ψ_i for each i that tests if a set D is a convex polytope with at most i vertices: it checks that the set of vertices $V(D) = \{x \in D \mid x \notin \mathrm{conv}(D-x)\}$ has at most i elements, and that $D = \mathrm{conv}(V(D))$. In order to check if a UCQ φ in $\mathrm{FO}+\textsc{Poly}$ is $\mathcal{CPT}$-preserving, one applies Lemma 7.7 to compute the numbers k and l, and then writes a sentence saying that for every $\leq k$-element set V in $\mathbb{R}^m$, applying φ to $\mathrm{conv}(V)$ yields a polytope with at most l vertices. Since conv and ψ_l are definable, this property can be expressed as an $\mathrm{FO}(+,*,0,1,<)$ sentence and thus it is decidable if it is true. Hence, the property of being $\mathcal{CPT}$-preserving is decidable.

Proof of (2). As in the proof of (1), we notice that the condition of Lemma 7.8

can be written as a first-order $L(+,*,0,1,<)$ sentence equivalent to

$$\forall\{\vec{x}_i^j\}_{i=1,\ldots,k}^{j=1,2,3}\ \exists\{\vec{y}_s^p\}_{s=1,\ldots,l}^{p=1,2,3}\ \forall\vec{x}.\ \left(\vec{x}\in\bigcup_{s=1}^{l}\operatorname{conv}(\{\vec{y}_s^p \mid p=1,2,3\})\right)\leftrightarrow\varphi'(\vec{x}),$$

where φ' is obtained from φ by replacing each occurrence of $S(u,v)$ with a formula expressing the fact that $(u,v)\in\bigcup_{i=1}^{k}\operatorname{conv}(\{\vec{x}_i^j \mid j=1,2,3\})$. Since the convex hull of a finite number of points is (FO + Poly)-definable, the condition of the lemma is indeed definable by an $L(+,*,0,1,<)$ sentence, and thus its validity is decidable. Hence, it is decidable if a union of conjunctive FO + Poly queries is $\mathsf{SLinComp}_2$-preserving. ∎

7.2. New expressivity bounds. We can also obtain new expressivity bounds by combining the dichotomy theorem with the idea of canonical representation. First, as an immediate consequence of the technique of Proposition 7.3, Lemma 7.7, and the dichotomy theorem, we obtain the following.

Corollary 7.10. *Let $\varphi(\vec{x})$ be an* FO + Poly *or* FO + Exp *$\mathcal{CPT}$-preserving query. Then there exists a polynomial p_φ such that, whenever D is a convex polytope with n vertices, $\varphi[D]$ has at most $p_\varphi(n)$ vertices.* ∎

From the proof of Lemma 7.8, one can extract the following, by applying the dichotomy theorem to a query that works on representations of compact semilinear sets in $\mathbb{R}^2$ as finite unions of triangles.

Corollary 7.11. *Let $\varphi(x,y)$ be an* FO + Poly *query that is* $\mathsf{SLinComp}_2$*-preserving. Then there exists a polynomial p_φ such that, whenever D is a compact semilinear set with n vertices, $\varphi[D]$ has at most $p_\varphi(n)$ vertices.* ∎

Consider the following problem: given a polyhedron P and $\epsilon > 0$, find a triangulation of P of mesh $< \epsilon$, that is, a triangulation such that the diameter of each simplex (triangle in dimension 2) is less than ϵ. It is a well-known result that each polyhedron admits such a triangulation [4]. The output of such a query can be structured in several ways, for example, by storing the information about the face structure of the triangulation. We impose only one requirement: that the vertices of the triangulation be computable.

Proposition 7.12. *There is no* FO + Exp *query that finds a triangulation of a given polygon with a given mesh. This continues to hold if we restrict to convex polytopes on a plane.*

Proof. Suppose such a query exists; now consider a new query that does the following. Its input is one binary relation containing a set X of points $\vec{x}_1,\ldots,\vec{x}_n$ on the real plane, and one unary relation containing a single real number $\epsilon > 0$. First, in FO + Poly, construct $\operatorname{conv}(X)$, and then find vertices of a triangulation with mesh $< \epsilon$. This is clearly a safe query, so by the dichotomy theorem, there exists a polynomial p such that the number of vertices of the triangulation is at most $m = p(n+1)$ ($n+1$ is the size of the input). Let d be the maximal distance between the points $\vec{x}_i, \vec{x}_j$ (and thus the diameter of $\operatorname{conv}(X)$). Since the segment $[\vec{x}_i, \vec{x}_j]$ with $d(\vec{x}_i, \vec{x}_j) = d$ must be covered by the simplexes of the triangulation, it is possible to find a number ϵ such that it cannot be covered by fewer than $m+1$ triangles of diameter ϵ, and hence the number of points in the triangulation is $> m$. This contradiction proves the proposition. ∎

8. Conclusion and future work. Let us summarize the main themes of the paper.

- The relational calculus with interpreted functions is a nontrivial and interesting extension of the relational calculus. Many useful properties of the relational calculus remain in place in the presence of built-in functions, but many don't. Identifying the analogs to classical results in the interpreted setting can be tricky; proving them is not necessarily a piece of cake either.
- What sort of interpreted structure one adds matters.
- By combining results on the relational calculus with interpreted functions with some simple canonical representations of constraint databases, one can get interesting bounds on the behavior of constraint queries.

We now discuss extensions of each of these themes to other settings. In the first part of the paper, we identified some helpful properties of the relational calculus that remain in the presence of well-behaved built-in functions, with the real arithmetic functions being our prototypical example. Looking at structures such as real arithmetic or rational addition was quite helpful in discovering these results, but these characterization theorems are by no means limited to functions on real or even rational domains. Results in this paper and in [8] indicate that the safety and bound results fail badly for full integer arithmetic. However, we are currently working on extensions of these results to well-behaved structures over the integers, such as linear integer constraints. Although the growth dichotomy and range-restriction theorems as stated here fail for integer linear constraints, modifications of the characterization results still hold. In addition, several of their algorithmic consequences, such as the decidability results for conjunctive queries, are still valid in the integer case.

In this paper we focused mainly on the relational calculus. Many of the proofs here, such as the results on range-restriction and safety, generalize straightforwardly to higher order logics (fixpoint, second-order). Still, the safety question for many higher order logics, particularly fixpoint logic in its many variations, is quite intricate, and we lack a full picture of what interpreted structures and recursion constructs permit a well-behaved theory of query safety.

Our emphasis here was showing that a wide class of interpreted functions satisfying some weak structural assumptions all exhibit certain kinds of tame behavior. In contrast, papers such as [19] give more detailed algorithmic analyses for specific structures. It still remains to give a complexity-theoretic analysis of both the safe translation problem and the query safety problem for conjunctive queries in the case of polynomial and linear constraints. A related interesting question is the complexity of deciding preservation properties for conjunctive queries over finitely representable databases.

We are working on several kinds of extensions of the growth bound theorems of section 6. Some of our current work is on broadening the class of models these results apply to, and some of it concerns getting more precise bounds on the behavior of the growth function. It is fairly clear why these bounds have never been stated for the pure case: they are completely obvious for any pure query language. However, the pure case does put some strong limits on what sort of more precise information one can obtain on the behavior of the growth function. For example, results in the pure first-order case show that the function $f(n)$ giving the minimum nonzero size of output over models of size n can be sublogarithmic. Well-known results on the spectrum problem give restrictions on the structure of the set $\{n : \mathit{growth}_\varphi(n) = k\}$ for k equal to any constant or ∞. One can, however, give theorems relating the behavior of the growth function of a polynomial constraint query to that of a pure first-order query. We are also working on characterizations of the classes of interpreted structures for which the growth bound dichotomy theorem holds, and on characterizations of structures for which the growth function is bounded not by a polynomial, but by other definable functions of the input size (e.g., exponential).

The second part of the paper deals with applying our results on finite databases to finitely representable ones, with the main technique coming via canonical codes. The main point of our codings was to facilitate this transfer of results. We are working on refining the results here to get natural canonical codings for larger geometric classes, and on studying these codes in themselves. The codings given here often capture a significant model-theoretic observation about the geometric class (e.g., the codings based on Carathéodory's theorem and its generalizations show that membership is determined by a bounded number of elements, and that these elements are definable from the database); they also give quite a bit of intuition on how queries that preserve these classes behave. We think this approach is quite a promising one for arriving at useful languages for queries that preserve geometric structure. In fact, very recently, a syntactically defined subquery language of $\mathrm{FO} + \mathrm{POLY}$ for manipulating semilinear databases was given in [45]. Their approach was to combine Theorems 4.5 and 7.2 with the coding technique of [13] to find a canonical coding for semi-linear sets. Further study of canonical codes may also shed light on decidability of preservation properties for special classes of queries.

Acknowledgments. We thank Ken Clarkson, Steve Fortune, and Jianwen Su for helpful discussions and anonymous referees for numerous comments and suggestions.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.

[2] F. Afrati, S. Cosmadakis, S. Grumbach, and G. Kuper, *Linear vs. polynomial constraints in database query languages*, in Proceedings of Conference on Principles and Practice of Constraint Programming, Springer-Verlag, New York, 1994, pp. 181–192.

[3] A. K. Ailamazyan, M. M. Gilula, A. P. Stolboushkin, and G. F. Shvarts, *Reduction of a relational model with infinite domains to the finite-domain case*, Dokl. Akad. Nauk SSSR, 286 (1986), pp. 308–311 (in Russian); Soviet Phys. Dokl., 31 (1986), pp. 11–13 (in English).

[4] P. S. Aleksandrov, *Combinatorial Topology*, Graylock Press, Albany, NY, 1956.

[5] A. Avron and J. Hirshfeld, *On first order database query languages*, in Proceedings of IEEE Logic in Computer Science, IEEE Press, Piscataway, NJ, 1991, pp. 226–231.

[6] M. Benedikt, G. Dong, L. Libkin, and L. Wong, *Relational expressive power of constraint query languages*, J. Assoc. Comput. Mach., 45 (1998), pp. 1–34.

[7] M. Benedikt and L. Libkin, *On the structure of queries in constraint query languages*, in Proceedings of IEEE Logic in Computer Science, IEEE Press, Piscataway, NJ, 1996, pp. 25–34.

[8] M. Benedikt and L. Libkin, *Languages for relational databases over interpreted structures*, in Proceedings of ACM Symposium on Principles of Databases Systems, ACM, New York, 1997, pp. 87–98.

[9] M. Benedikt and L. Libkin, *Safe constraint queries*, in Proceedings of ACM Symposium on Principles of Databases Systems, ACM, New York, 1998, pp. 99–108.

[10] H. Bieri and W. Nef, *Elementary set operations with d-dimensional polyhedra*, in Proc. Workshop on Computational Geometry (CG'88), Lecture Notes in Comput. Sci. 333, Springer-Verlag, New York, pp. 97–112.

[11] C. C. Chang and H. J. Keisler, *Model Theory*, North-Holland, Amsterdam, 1990.

[12] M. Davis, *On the number of solutions of Diophantine equations*, Proc. Amer. Math. Soc., 35 (1972), pp. 552–554.

[13] F. Dumortier, M. Gyssens, L. Vandeurzen, and D. Van Gucht, *On the decidability of semi-linearity of semi-algebraic sets, and its implications for spatial databases*, in Proceedings of ACM Symposium on Principles of Databases Systems, ACM, New York, 1997, pp. 68–77.

[14] M. Escobar-Molano, R. Hull, and D. Jacobs, *Safety and translation of calculus queries with*

scalar functions, in Proceedings of ACM Symposium on Principles of Databases Systems, ACM, New York, 1993, pp. 253–264.

[15] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, New York, 1993.

[16] S. Grumbach and J. Su, *Finitely representable databases*, J. Comput. System Sci., 55 (1997), pp. 273–298.

[17] S. Grumbach, J. Su, and C. Tollu, *Linear constraint databases*, in Proceedings of Logic and Computational Complexity, Springer-Verlag, New York, 1994, pp. 426–446

[18] R. Hull and J. Su, *Domain independence and the relational calculus*, Acta Inform., 31 (1994), pp. 513–524.

[19] O. H. Ibarra and J. Su, *On the containment and equivalence of database queries with linear constraints*, in Proceedings of ACM Symposium on Principles of Databases Systems, ACM, New York, 1997, pp. 32–43.

[20] P. Kanellakis, G. Kuper, and P. Revesz, *Constraint query languages*, J. Comput. System Sci., 51 (1995), pp. 26–52; extended abstract in *PODS*'90, pp. 299–313.

[21] M. Kifer, *On safety, domain independence, and capturability of database queries*, in Proc. Data and Knowledge Base, Morgan Kaufmann, Jerusalem, 1988, pp. 405–415.

[22] M. Kifer, R. Ramakrishnan, and A. Silberschatz, *An axiomatic approach to deciding query safety in deductive databases*, in Proceedings of the ACM Symposium on Principles of Database Systems, ACM, New York, 1988, pp. 52–60.

[23] D. Marker, M. Messmer, and A. Pillay, *Model Theory of Fields*, Springer-Verlag, New York, 1996.

[24] A. Levy and D. Suciu, *Deciding containment for queries with complex objects*, in Proceedings of the ACM Symposium on Principles of Database Systems, ACM, New York, 1997, pp. 20–31.

[25] W. Nef, *Beiträge zur Theorie der Polyeder*, Herbert Lang, ed., Bern, Switzerland, 1978.

[26] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, Cambridge, UK, 1994.

[27] J. Paredaens, J. Van den Bussche, and D. Van Gucht, *Towards a theory of spatial database queries*, in Proceedings of ACM Symposium on Principles of Database Systems, ACM, New York, 1994, pp. 279–288.

[28] J. Paredaens, J. Van den Bussche, and D. Van Gucht, *First-order queries on finite structures over the reals*, SIAM J. Comput., 27 (1998), pp. 1747–1763.

[29] J. Paredaens, B. Kuijpers, G. Kuper, and L. Vandeurzen, *Euclid, Tarski and Engeler encompassed*, in Proceedings of Database Programming Languages, Lecture Notes in Comput. Sci. 1369, Springer-Verlag, New York, 1998, pp. 1–24.

[30] A. Pillay and C. Steinhorn, *Definable sets in ordered structures.* III, Trans. Amer. Math. Soc., 309 (1988), pp. 469–476.

[31] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz, *Safety of recursive Horn clauses with infinite relations*, in Proceedings of ACM Symposium on Principles of Database Systems, ACM, New York, 1987, pp. 328–339.

[32] P. Revesz, *Safe query languages for constraint databases*, ACM Trans. Database Systems, 23 (1998), pp. 58–99.

[33] R. T. Rockafellar, *Convex Analysis*, Princeton University Press, Princeton, NJ, 1970.

[34] J. Ruppert and R. Seidel, *On the difficulty of triangulating three-dimensional nonconvex polyhedra*, Discrete Computat. Geom., 7 (1992), pp. 227–254.

[35] Y. Sagiv and M. Yannakakis, *Equivalence among relational expressions with the union and difference operators*, J. Assoc. Comput. Mach., 27 (1980), pp. 633–655.

[36] A. Stolboushkin and M. Tsaitlin, *Finite queries do not have effective syntax*, Inform. and Comput., 153 (1999), pp. 99–116.

[37] A. Stolboushkin and M. Tsaitlin, *Linear vs. order constraint queries over rational databases*, in Proceedings of ACM Symposium on Principles of Database Systems, ACM, New York, 1996, pp. 17–27.

[38] A. Stolboushkin and M. Tsaitlin, *Safe stratified datalog with integer order does not have syntax*, ACM Trans. Database Systems, 23 (1998), pp. 100–109.

[39] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, MD, 1988.

[40] J. D. Ullman, *Information integration using logical views*, in Proceedings of International Conference on Database Theory, Lecture Notes in Comput. Sci. 1186, Springer, Berlin, 1997, pp. 19–40.

[41] L. van den Dries, *Remarks on Tarski's problem concerning (R,+,*,exp)*, in Logic Colloquium '82, North-Holland, Amsterdam, 1984, pp. 97–121.

[42] L. van den Dries, *Tame Topology and O-Minimal Structures*, Cambridge University Press, Cambridge, UK, 1998.

[43] L. Vandeurzen, M. Gyssens, and D. Van Gucht, *On the desirability and limitations of linear spatial database models*, Lecture Notes in Comput. Sci. 951, Springer, New York, 1995, pp. 14–28.

[44] L. Vandeurzen, M. Gyssens, and D. Van Gucht, *On query languages for linear queries definable with polynomial constraints*, in Proc. Principles and Practice of Constraint Programming, Lecture Notes in Comput. Sci. 1118, Springer-Verlag, 1996, pp. 468–481.

[45] L. Vandeurzen, M. Gyssens, and D. Van Gucht, *An expressive language for linear spatial database queries*, in Proceedings of ACM Symposium on Principles of Database Systems, ACM, New York, 1998, pp. 109–118.

[46] A. van Gelder and R. Topor, *Safety and translation of relational calculus queries*, ACM Trans. Database Systems, 16 (1991), pp. 235–278.

[47] M. Y. Vardi, *The decision problem for database dependencies*, Inform. Process. Lett., 12 (1981), pp. 251–254.

[48] A. J. Wilkie, *Model completeness results for expansions of the ordered field of real numbers by restricted Pfaffian functions and the exponential function*, J. Amer. Math. Soc., 9 (1996), pp. 1051–1094.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1683–1702

ON BROADCAST DISK PAGING*

SANJEEV KHANNA† AND VINCENZO LIBERATORE‡

Abstract. *Broadcast disks* are an emerging paradigm for massive data dissemination. In a broadcast disk, data is divided into n equal-sized pages, and pages are broadcast in a round-robin fashion by a server. Broadcast disks are effective because many clients can simultaneously retrieve any transmitted data. Paging is used by the clients to improve performance, much as in virtual memory systems. However, paging on broadcast disks differs from virtual memory paging in at least two fundamental aspects:

- A page fault in the broadcast disk model has a variable cost that depends on the requested page as well as the current state of the broadcast.
- Prefetching is both natural and a provably essential mechanism for achieving significantly better competitive ratios in broadcast disk paging.

In this paper, we design a deterministic algorithm that uses prefetching to achieve an $O(n \log k)$ competitive ratio for the broadcast disk paging problem, where k denotes the size of the client's cache. We also show a matching lower bound of $\Omega(n \log k)$ that applies even when the adversary is not allowed to use prefetching. In contrast, we show that when prefetching is not allowed, no deterministic online algorithm can achieve a competitive ratio better than $\Omega(nk)$. Moreover, we show a lower bound of $\Omega(n \log k)$ on the competitive ratio achievable by any nonprefetching randomized algorithm against an oblivious adversary. These lower bounds are trivially matched from above by known results about deterministic and randomized marking algorithms for paging. An interpretation of our results is that in the broadcast disk paging, prefetching is a perfect substitute for randomization.

Key words. design of algorithms, online algorithms, competitive analysis, paging, distributed systems, client-server architecture, broadcast disks

AMS subject classifications. 68A10, 68Q25, 68H

PII. S0097539798341399

1. Introduction.

In traditional client-server architectures, such as the World Wide Web, data transfers are initiated by clients that request information from servers. Such an architecture is said to be a "pull" system because clients pull data from the servers. An emerging alternative to pull systems is the "push" technology. In a push system, the server repeatedly broadcasts data to clients; thus the server now "pushes" information toward the clients.

Broadcast disks are a widely used type of push technology. In broadcast disks, data are divided into pages of equal sizes, and pages are broadcast in a round-robin fashion. The name "broadcast disk" derives from the broadcast program being a circular repetition of pages.

1.1. A widespread application.

Broadcast disks have been deployed since the early 80's by most national television companies in western Europe. Broadcast disk

*Received by the editors June 27, 1998; accepted for publication (in revised form) July 3, 1999; published electronically March 17, 2000. A preliminary version of this paper appeared in *Proceedings of the* 30th *Annual ACM Symposium on the Theory of Computing*, 1998, pp. 634–643.

http://www.siam.org/journals/sicomp/29-5/34139.html

†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 (sanjeev@cis.upenn.edu, http://www.cis.upenn.edu/~sanjeev). This work was done while at Bell Labs, Murray Hill, NJ 07974.

‡UMIACS, A. V. Williams Building, University of Maryland, College Park, MD 20742 (vliberatore@acm.org, http://www.umiacs.umd.edu/users/liberato). This work was done while at Rutgers University and Bell Labs and was supported by NSF Career Development Award CCR-9501942, NATO grant CRG 960215, NSF/NIH grant BIR 94-12594-03-CONF, and an Alfred P. Sloan Research Fellowship.

technology has since attained nationwide diffusion and reaches most households. It provides a continuous information source and has deeply influenced the lifestyle of the countries where it is operational [10, 17]. Broadcast disks have been used in high throughput multiprocessor database systems over high bandwidth networks [6] and wireless communication [11]. An interested reader is referred to the survey lecture by Franklin and Zdonik [9] that reports results and research directions in the field of broadcast disks. The client storage resources have been recently integrated in the broadcast disk approach [1, 2]. The client decides which pages to keep in its local cache and which pages to evict. If the client finds a requested page in its local cache, then the page request can be satisfied at no cost. However, if the client does not have the requested page, it will have to wait for that page to be broadcast again by the server. The client's objective is to minimize the completion time needed to satisfy a sequence of page requests. The resulting paging system has some affinity to the traditional virtual memory systems [4]. We refer to it as *broadcast disk paging (BDP)*. The objective of this paper is to study paging algorithms for BDP in the framework of competitive analysis (see [5, 12, 18], for instance).

1.2. Theoretical significance. BDP is not reducible to traditional online paging and poses several unexplored theoretical problems. It differs from traditional paging in at least two main ways:

- The cost of faulting on any page changes dynamically with time in the case of BDP.
- Prefetching dramatically improves the competitive ratio of online algorithms for BDP and brings in the same advantage as randomization.

We elaborate these two points next. In traditional paging models, either the cost of a page fault is uniform [4] or depends only on the faulting page [15]. In contrast, the cost of a fault in BDP is the time spent waiting for the requested page to be transmitted. It depends on the time step reached during the transmission schedule and could range from 1 (the desired page is currently being broadcast) to n (the desired page was broadcast just before), where n is the total number of pages in the server broadcast. As a result, the design and analysis of BDP algorithms involve techniques and ideas that are often significantly different from those for traditional paging. Moreover, the final behavior of the BDP problem is very different from virtual memory paging as illustrated by the following elementary example. Suppose that the server broadcasts n pages and that the client has a local cache of size k. We show that, when $n = k + 1$, there is a 1-competitive deterministic BDP online algorithm. Briefly, the 1-competitive algorithm keeps in the cache all pages except the one that is about to be broadcast. Such an algorithm pays a constant cost per fault, and after at most n faults it will have in the cache exactly the same pages as the adversary. Hence, the algorithm cost is only an additive factor away from the optimum. This is to be contrasted with virtual memory paging where the adversary can force a worst-case competitive ratio of k.

BDP also differs from the traditional paging in the role played by *prefetching*. Prefetching is essentially ruled out from traditional paging without loss of generality [15]. In this paper, we will show that in BDP prefetching is critical to dramatically improve the competitive ratio of online algorithms. In fact, we will show that prefetching is an exact substitute for randomization in BDP. The importance of prefetching can be intuitively explained as follows. Prefetching makes it possible for a client to reload a page recently evicted and, in some sense, allows the client to dynamically revise its eviction decisions.

1.3. New results. In the absence of any prefetching, the conventional caching analysis can be extended to show that any marking algorithm is $O(nk)$-competitive. But things are far from clear when either the adversary or the online algorithm is allowed to prefetch arbitrary subsets of pages. Our main result is that there exists a deterministic online algorithm that uses prefetching and is $O(n \log k)$-competitive. Our algorithm uses the history of the request sequence and prefetching to develop a strategy that dynamically rectifies any potential eviction mistake. We also show that no deterministic online algorithm can achieve a competitive ratio better than $\Omega(n \log k)$. Therefore, our algorithm is optimal up to a constant factor. In contrast, we prove that in the absence of prefetching, no deterministic algorithm can achieve a competitive ratio better than $\Omega(nk)$, and that even with the use of randomization, there is a lower bound of $\Omega(n \log k)$ against oblivious adversaries. The corresponding randomized upper bound is $O(n \log k)$.

Finally, we extend all our results to a more general model, called the *delay model*, where at any step, the adversary generates either a page request or a delay request. The delay requests capture the fact that a client may use the pages in its cache for variable amounts of time, before ever requesting an outside page. The difficulty is that the adversary may introduce delays to disrupt the current state of the online algorithm. But we show that all of our results hold unchanged in the general model.

1.4. Paging and scheduling. In traditional paging, the cost of a page fault depends on the cache configuration. In BDP, the cost depends on a dynamic state that is not confined only to the contents of the local cache but reflects the configuration of other areas of the system. Specifically, the BDP cost is affected by the time reached along the broadcast schedule. In fact, BDP is only one of many problems where there is a strong interaction between caching and schedules. For example, paging can be integrated with prefetching strategies and yields a problem where performance depends on cache contents as well as on the current disk state [7]. We suspect that the results in this paper might shed light also on other online problems where caching is interrelated with scheduling problems.

1.5. Organization. In section 2, we formalize our problem, define our notation, and establish some basic properties of BDP that are useful to our study. In section 3, we study a simple special case of BDP that highlights an important difference between BDP and virtual memory paging. We next study deterministic as well as randomized BDP algorithms that do not use prefetching in section 4. In section 5, we establish our main results, namely, an $O(n \log k)$-competitive deterministic algorithm and an $\Omega(n \log k)$ lower bound on the deterministic competitive ratio for BDP with prefetching. In section 6, we examine the general delay model. We conclude the paper in section 7.

2. Preliminaries.

In this section, we formally describe various aspects of BDP and establish our notation. In a broadcast disk, a server broadcasts a set $P = \{0, 1, \ldots, n-1\}$ of pages over a network in a round-robin fashion. The pages are received by the clients in the same order as they are transmitted by the server. Each client operates in perfect isolation from other clients and thus the performance of a client is independent of the behavior of other clients. A client can cache at most k pages. We will typically assume that $a \leq k \leq bn$ for some positive integer a and a positive real constant $b < 1$. The assumption reflects the fact that paging is usually done only if there is enough space in the local cache for at least a small number of pages and that the local cache size is typically much smaller than n. Applications

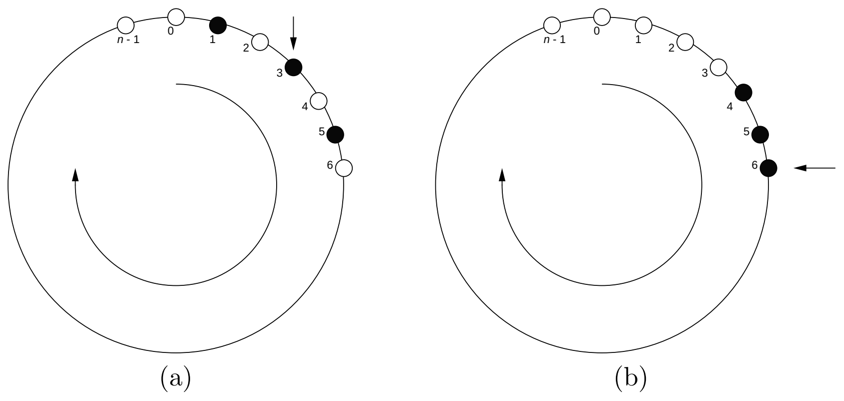


FIG. 2.1. *Examples of client configurations. Black circles represent cached pages and the outer arrow the algorithm position.*

running on the clients generate a sequence σ of page requests. We next define the notion of configuration of a paging algorithm and describe the sequence of actions taken by a client to service a sequence σ of page requests.

2.1. Configuration of an algorithm. The *configuration* of a BDP algorithm G is the contents of its cache along with the step reached along the transmission schedule. Formally, a configuration is a pair $(M, i) \in 2^P \times P$, where

- M is a subset of P of size at most k that is present in G's local cache,
- the index i is the last page that was broadcast and received by the client.

If G's configuration is (M, i), we will say that G is *positioned* over page i or that G's *position* is i. If $M = \{p_1, p_2, \ldots, p_h\}$, then we will denote the memory configuration (M, p_h) as $\{p_1, p_2, \ldots, \triangleright(p_h)\}$.

Example. Figure 2.1(a) represents a broadcast cycle and a configuration (M, i) with $M = \{1, 3, 5\}$ and $i = 3$. The inner arrow represents the order in which pages are broadcast. The outer arrow gives the position of G, that is, the last page that was received by the client. Black circles represent pages in M (contents of G's cache), and white circles represent pages that are not in M.

Let (M, i) be G's configuration. Then, the next page that G will receive is page $i' = (i + 1) \bmod n$. As soon as G has received i', it has the option of loading it into the cache. Thus G's new configuration is of the form (M', i'), where $M' \subseteq M \cup \{i'\}$. Notice that if the cache was full ($|M| = k$) and $i' \in M' - M$, then there must be a page $p \in M - M'$ that is evicted from the cache to make room for i'.

2.2. Page requests. The sequence of page requests generated by a client is denoted by $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle \in P^m$. For notational convenience, let $\sigma_0 = k - 1$. The client services a request $\sigma_j \in \sigma$ in an online fashion by repeatedly performing the following sequence of actions: the client receives the next page i from the broadcast and decides whether to cache it or not. This procedure terminates only when the requested page σ_j is in the cache.

Remark. Notice that the client *can* stop the loop iteration when σ_j is in the cache, but it *does not have* to stop the loop the first time σ_j is in the cache. If the client finds it advantageous to keep listening to the broadcast after σ_j is received, it can do so. On the other hand, if σ_j is already in the cache immediately before the jth request is issued, then the client does not have to execute any loop iteration at all. We will prove in Proposition 2.7 that, without loss of generality, clients will stop as soon as σ_j is in the cache.

The cost of a client is the total number of pages it has received from the broadcast, or equivalently, the number of broadcast slots to which the client has listened.

Example. Assume the same set-up as in Figure 2.1(a) and suppose that a request for page 6 is issued. Since page 6 is not in the client's cache, the client listens to the broadcast. Since the client is positioned over page 3, the next page on the broadcast is page 4. The client receives it and decides whether to cache it or not. If it does, it will have to choose a page in $\{1, 3, 5\}$ to evict. In either case, page 6 is not in the cache, so the client keeps listening to the broadcast. Then, page 5 is broadcast. Again, the client can choose whether to cache page 5 or not and continues listening to the broadcast again. Finally, page 6 is broadcast. The client can choose to cache page 6. If it does, it can also choose whether it will keep listening to the broadcast for page 7 or whether it will remain positioned over page 6 and process the next request in σ. Figure 2.1(b) gives an example when pages 4, 5, and 6 have all been cached and the client has stopped immediately after loading page 6. The total cost incurred is three.

Remark. We remark that, according to the previous definition, only the number of received pages contributes to the cost. The definition above entails that when a requested page is in the cache and the client chooses not to move, the cost for that request is zero. In section 6, we will consider a more complex cost model where local computation could cause delays and force the client to skip items in the broadcast. However, for the time being, we will not charge for page requests directly satisfied in the cache.

Paging over a broadcast disk extends the virtual memory, except that now paging is from the network rather than from a physical disk. Clearly, as in virtual memory paging, a page replacement algorithm affects the performance of the system. Virtual memory paging and BDP differ on the following essential point: paging replacement algorithms aim at minimizing the number of page faults (since each fault costs the same), whereas BDP replacement algorithms aim at minimizing the total time spent waiting for pages to arrive from the network.

In what follows, we assume that G is a broadcast disk page replacement algorithm. We will say that G *prefetches* a page p if G loads p even though p is not the currently requested page. In BDP, some pages can be prefetched at no additional cost.

Example. In the example above, page 6 was requested, and page 4 and page 5 were loaded even though they were not requested. Hence page 4 and page 5 were prefetched. Moreover, the client cost is three, independent of whether pages 4 and 5 are prefetched.

We introduce the following notation. The memory configuration reached by G before the jth request will be denoted as $\mathcal{M}(G, j)$ or simply as $\mathcal{M}(G)$ when the time index is clear from the context. Note that the index i in (M, i), and the index j in $\mathcal{M}(G, j) = (M, i)$, denote different quantities. We will assume without loss of generality that the initial configuration of any algorithm G is $\mathcal{M}(G, 1) = \{0, 1, \ldots, \triangleright(k-1)\}$. If $\mathcal{M}(G) = (M, p)$, then we will write $q \in \mathcal{M}(G)$ if and only if $q \in M$. If the jth request $\sigma_j \notin \mathcal{M}(G, j)$, then we will say that G *misses* or *faults* on page σ_j. If $\mathcal{M}(G, j) \neq \mathcal{M}(G, j+1)$, we will say that G *moves* at step j. The algorithm G moves either when it changes cache contents or when it changes position. The cost of G on σ will be denoted as $c(G, \sigma)$ or simply as $c(G)$.

Example. In the example above, G misses and moves from page 3 to page 6. Eventually, G is positioned over page 6.

If $p \in P$ is a page and i a nonnegative integer, then we will write $p + i$ instead of

$(p+i)$ mod n when no confusion can arise. If $p, r \in P$ are pages then we will write $p - r$ instead of $(p-r)$ mod n when no confusion can arise. Such notation is useful for calculating costs and movements: if G moves from p to q, then its cost is $q - p$, and it is positioned over $q = p + (q - p)$.

2.3. Rotations, transit, and ubiquity. We next define two concepts that we use frequently in our analysis.

Definition 2.1. *The algorithm G executes at least i* rotations *during σ if $c(G, \sigma) \geq i \cdot n$.*

Definition 2.2. *The algorithm G is said to* transit over *page p at step j if it moves from a page $p - h$ to a page $p + l$ ($h > 0, l \geq 0$) at step j.*

Another notion is that of *ubiquitous adversary.* An adversary is said to be ubiquitous if it pays only a unit cost per fault. Since a fault forces any algorithm to receive at least one page, the ubiquitous adversary is at least as strong as an ordinary adversary. We will use the ubiquitous adversary to extend competitiveness results to the delay model. Henceforth, we will assume that the adversary is not ubiquitous unless stated otherwise. A property of the ubiquitous adversary is that it will never prefetch a page [15]. Actually, we could assume without loss of generality that the ubiquitous adversary serves a request sequence in accordance with Belady's algorithm [4], but we will not use this fact.

2.4. Lazy algorithms and hard sequences. We will now define the notions of *lazy algorithms* and *hard sequences* for BDP, compare our definitions with the analogous ones that are given in the context of virtual memory paging, and show that we can assume without loss of generality that algorithms are lazy and sequences hard. First, we define lazy algorithms.

Definition 2.3. *A paging algorithm G is* lazy for virtual memory paging *if G never prefetches a page.*

It can be shown that any algorithm for virtual memory paging can be transformed into a lazy algorithm without any degradation in performance [15]. The definition of lazy algorithms in BDP is as follows.

Definition 2.4. *A paging algorithm G is* lazy for BDP *(or simply* lazy*) if, when G is positioned on page r and faults on page p, its cost is exactly $p - r$.*

In other words, a lazy algorithm stops listening to the broadcast as soon as it receives the faulting page. According to Definition 2.4, a lazy algorithm can prefetch pages as long as those pages are loaded while waiting for the faulting page to be broadcast. Then, a lazy algorithm is not necessarily lazy for virtual memory paging, but a lazy algorithm for virtual memory paging is also lazy for BDP.

Definition 2.5. *A request sequence σ is* hard *for G if G faults on every request in σ.*

The definition above coincides with the one used in virtual memory paging and in BDP. We now claim in the spirit of [15] that it is enough to compare lazy online algorithms to adversaries running lazy algorithms on hard sequences.

Lemma 2.6. *For any (online) algorithm G, there is an (online) algorithm G' that satisfies $c(G') = c(G)$ and that has the property that, when G' faults on page σ_j, it always loads σ_j the first time G' transits over σ_j.*

Proof. The proof is by induction on the length of σ. Suppose that G satisfies the property before request σ_j, but it violates it on σ_j. The first time G transits over σ_j, it does not load it. Therefore, G will transit over σ_j a second time. Hence, G transits over every page in P after it has transited over σ_j for the first time. The algorithm G' will exactly follow G until page σ_j is received for the first time. Then, G' loads

σ_j and evicts a page p. Afterwards, G' emulates G until either G evicts p, in which case G' evicts σ_j, or G receives p. Notice that G receives p before it receives σ_j for the second time. At that point, G' evicts σ_j and reloads p. Now, the configurations of the two algorithms are identical, and the cost of G' is exactly equal to the cost of G. Finally, we notice that G' does not require any knowledge of the future beyond G's. $\square$

PROPOSITION 2.7. *For any (online) algorithm G, there is an (online) algorithm G' that is lazy and that satisfies $c(G') \leq c(G)$.*

Proof. The proof is by induction on the length of σ. Suppose that G is lazy before request σ_j, but it violates that condition on request σ_j. We can assume without loss of generality that G loads σ_j the first time it transits over it. Afterwards, G will receive an additional set Q of pages. The algorithm G' stops immediately after receiving σ_j. If $j = m$, then G' will not perform any further action and $c(G') < c(G)$. Otherwise, G' services σ_{j+1} by first receiving all pages in Q and then by emulating G on σ_{j+1}. Therefore, $\mathcal{M}(G, j+2) = \mathcal{M}(G', j+2)$ and $c(G') \leq c(G)$. Finally, we notice that G' does not require any knowledge of the future beyond G's. $\square$

PROPOSITION 2.8. *If G is an algorithm that does not move for requests that do not cause a fault, that does not base its eviction decision on nonfaulting requests, and that is c-competitive on all of its hard sequences, then G is c-competitive.*

Proof. Let σ be a sequence of request. Define σ' to be the associated hard sequence, that is, the subsequence of σ consisting of all requests where G faults. Then, $c(G,\sigma) = c(G,\sigma')$. Let H be the optimum algorithm. Then, $c(H,\sigma') \leq c(H,\sigma)$. It follows that, for some constant b,

$$c(G,\sigma) = c(G,\sigma') \leq c \cdot c(H,\sigma') + b \leq c \cdot c(H,\sigma) + b. \qquad \square$$

If G is a lazy algorithm and σ is hard, then G is positioned on σ_{j-1} before the jth request.

3. BDP and virtual memory paging: The case $n = k + 1$. In this section, we will analyze the simple case $n = k + 1$ to highlight a fundamental difference between traditional paging and BDP. In traditional paging, the adversary can force a worst-case sequence when $n = k + 1$. In contrast, we show the following result.

PROPOSITION 3.1. *There is a 1-competitive deterministic online algorithm for BDP when $n = k + 1$.*

Proof. Let G be the algorithm that on a faulting request σ_j maintains $\mathcal{M}(G, j+1) = \{\triangleright(\sigma_j)\} \cup \{i \in P : i \neq \sigma_j + 1\}$. In other words, $\sigma_j + 1$ is the only page missing from G's fast memory. When G faults on σ_j, it evicts $\sigma_j + 1$ and loads σ_j. So, G pays a unit cost whenever it faults. Suppose without loss of generality that σ is hard for G, and thus $c(G,\sigma) = m$. In fact, $\sigma_j = k + j$ by induction on j.

Assume without loss of generality that the adversary follows a lazy algorithm H. Suppose that H faults on $\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_h}$. The cost of H on σ is then

$$c(H,\sigma) = \sum_{l=1}^{h} (\sigma_{j_l} - \sigma_{j_{l-1}}) = \sum_{l=1}^{h} (k + j_l - k - j_{l-1}) = \sum_{l=1}^{h} (j_l - j_{l-1}) = j_h - 1.$$

Notice that any subsequence of at most n consecutive requests in σ consists of distinct pages. Also, H does not fault on σ_j when $j > j_h$, and so $|\{\sigma_{j_h+1}, \ldots, \sigma_m\}| \leq k-1 < n$, which implies that $\sigma_{j_h+1}, \ldots, \sigma_m$ are all distinct and $m - j_h \leq k - 1$. Therefore, $c(G,\sigma) - c(H,\sigma) = m - (j_h - 1) \leq k$, and the claim is proven. $\square$

The case $n = k + 1$ sets up the general framework for finding lower bounds on the competitive ratio of traditional paging algorithms. Clearly, the general paradigm does not work for BDP. Instead, we will show that all lower bounds can be ultimately traced back to the notion of algorithm in transit, as defined in the previous section.

4. BDP algorithms without prefetching. In this section, we will examine the competitive ratio of online algorithms that do not allow prefetching. In terms of Definition 2.3, such algorithms are lazy for virtual memory paging.

If G is a lazy algorithm for virtual memory paging, it induces an algorithm for BDP that is lazy and whose memory contents after σ_j are exactly the same as G's. For simplicity, we will denote both the virtual memory paging algorithm and its BDP counterpart with the same symbol. A strongly competitive deterministic paging algorithm never incurs more than k times the optimum number of page faults. Since its BDP counterpart is lazy, it follows that a page fault costs at most $n - 1$; the last page that was broadcasted and received must be in the algorithm's cache. Thus, any strongly competitive paging algorithm, e.g., the marking algorithm [13], is trivially $(k(n-1))$-competitive for broadcast disk paging even against a ubiquitous adversary. In fact, we can show a slightly better competitive ratio.

THEOREM 4.1. *Let G be a lazy α-competitive deterministic algorithm for paging. Then, the competitive ratio of G for BDP is $(\alpha - 1)n + 1$.*

Proof. We will assume without loss of generality that the sequence σ is hard for G and that the adversary uses a lazy algorithm H. First, we notice that the cost of H is at least equal to the number of page faults. Indeed, every time H brings a new page into fast memory, H has to transit over that page, paying a unit cost.

Let r_j be the page where the adversary is positioned before the jth request, and define the potential function as $\Phi(j) = r_j - \sigma_{j-1}$. Let $a_j = (\sigma_j - \sigma_{j-1}) + \Phi(j+1) - \Phi(j)$ be the amortized cost of G to serve request σ_j. Notice that if $r_{j+1} = r_j$, then $a_j = (\sigma_j - \sigma_{j-1}) + (r_j - \sigma_j) - (r_j - \sigma_{j-1}) \le n$. If $\sigma_j \notin \mathcal{M}(H, j)$, then $\Phi(j+1) = 0$ and $a_j = \sigma_j - \sigma_{j-1} + \Phi(j+1) - \Phi(j) = (\sigma_j - r_j) + (r_j - \sigma_{j-1}) - (r_j - \sigma_{j-1})$, which is the real cost of the optimum on that request. Let l be the number of times that the optimum algorithm faults. Then $|\sigma| \le \alpha l + b$ for some constant b. Moreover, $l \le c(H, \sigma)$. The resulting cost of G is $c(G, \sigma) = (|\sigma| - l)n + c(H, \sigma) \le ((\alpha - 1)l + b)n + c(H, \sigma) \le (n(\alpha - 1) + 1)c(H, \sigma) + bn$, which proves the theorem. □

COROLLARY 4.2. *The marking algorithm is $((k-1)n + 1)$-competitive.*

We next establish an essentially matching lower bound.

THEOREM 4.3. *No deterministic online algorithm without prefetching can have a competitive ratio better than $\Omega(nk)$ for $2 \le k \le n - 2$ and any fixed $b < 1$, even if the adversary is not allowed to do prefetching.*

Proof. Let G be an online algorithm without prefetching and H be the algorithm used by the adversary. The adversary proceeds in phases. We will maintain that, at the beginning of a phase, $\mathcal{M}(G)$ contains the same pages as $\mathcal{M}(H)$, and that H is positioned on a page q such that $q + 1, q + 2 \notin \mathcal{M}(H)$. Define the set $W = \mathcal{M}(H) \cup \{q + 1, q + 2\}$ and call W the *working set* of the current phase. The gist of the proof is as follows. Notice that $|W| = k + 2$, and so there are always two pages in the working set W that are not in G's fast memory. The adversary will request those pages, and consequently G will fault at every step. Furthermore, we will use the notion of transit to show that G pays $\Omega(n)$ on every two faults. The adversary will generate $\Omega(k)$ requests in the phase and itself serves them at a cost of two. We now detail the argument.

The adversary starts the phase by requesting page $q + 1$ followed by page $q + 2$.

Thus, both G and H are now positioned at $q+2$. The rest of the phase is divided into segments. We will let r be the page where G is positioned at the beginning of a current segment. For example, in the first segment, $r = q+2$. Let α and β be the two pages that are in the working set W but not in G's fast memory at the beginning of a segment. Assume without loss of generality that $\beta - r > \alpha - r$. Then, the adversary requests β followed by α, and the segment is over. The algorithm G faults on the request for β and transits over α while waiting for β to be transmitted; but it does not load α because it is not allowed to prefetch. Therefore, G faults also on the request for α. On the whole, G transits over α at least twice during the segment, and so it executes at least one rotation per segment. We will now describe how many segments are generated. The adversary counts the number of distinct pages that made G fault during all the segments generated so far. The adversary generates segments until G has faulted on a set F of at least $k-2$ distinct pages. Notice that F is contained in the working set. Moreover, during each segment, G faults on at most two new pages, so that $|F| \in \{k-2, k-1\}$. Hence, at least $\lceil (k-2)/2 \rceil$ segments are generated. We will now specify how H serves the requests in the phase. The algorithm H faults on $q+1$ and $q+2$. At this point, its fast memory contains all the pages of $F \cup \{q+2\}$, plus another arbitrary page from W if $|F \cup \{q+2\}| < k$. Notice that H does not prefetch any page and that H does not fault during any segment. If G and H's fast memory configurations still differ after all segments have been generated, the adversary keeps requesting pages in H's memory that are missing from G's memory, until the two memories coincide in their contents. Notice that if the two memories never coincide, then this step will continue indefinitely, and G is not competitive. Henceforth, assume that G's and H's memory contents will eventually be the same. At this point, the phase is over. The cost of G in the phase is at least the cost in the segment, and so hence at least $n(k-2)/2$. The cost of H in the entire phase is 2—due only to the initial faults on $q+1$ and $q+2$.

The adversary issues $(n-k)/2$ request phases paying only a cost of 2 per phase. Then the adversary pays a cost of k and returns to the starting configuration $\{0, 1, \ldots, \triangleright(k-1)\}$. It then keeps requesting pages $0, 1, \ldots, k-1$ until the fast memory configurations of G and H coincide. Therefore, the cost of the adversary on $(n-k)/2$ phases is n, while the cost of G is at least $((n-k)/2)(n(k-2)/2)$, and the result follows. $\square$

The construction above extends to a lower bound for randomized algorithms against an adaptive online adversary.

THEOREM 4.4. *No randomized online algorithm without prefetching can have a competitive ratio better than $\Omega(nk)$ against an adaptive online adversary for $3 \le k \le bn$ and any fixed $b < 1$, even if the adversary is not allowed to do prefetching.*

Proof. The initial setup of the proof is the same as that of Theorem 4.3. The main difference between the proofs of Theorem 4.3 and Theorem 4.4 is that, at the beginning of a phase, the adversary does not know the set F of pages that make G fault. We overcome the difficulty by letting the adversary guess F and showing that the expected number of segments is $\Omega(k)$.

The adversary proceeds in phases. Assume that j is the step of the first phase request, so that $\mathcal{M}(H, j)$ and $\mathcal{M}(G, j)$ denote the configurations of H and G immediately before the current phase begins. We will maintain that $\mathcal{M}(G, j)$ contains the same pages as $\mathcal{M}(H, j)$, and that H is positioned on a page q such that $q+1, q+2 \notin \mathcal{M}(H)$. Define the set $W = \mathcal{M}(H, j) \cup \{q+1, q+2\}$ and call W the working set of the current phase. The adversary starts the phase by requesting $q+1$

followed by $q+2$. Thus, both G and H are now positioned on $q+2$. The algorithm H makes room for $q+1$ and $q+2$ by evicting two random pages r_1 and r_2 from $\mathcal{M}(H,j)$, so that now H's configuration is $\mathcal{M}(H) = (\mathcal{M}(H)-\{r_1,r_2\})\cup\{q+1,\triangleright(q+2)\}$. Notice that $r_1, r_2 \in \mathcal{M}(H,j)$ implies $\{r_1,r_2\}\cap\{q+1,q+2\}=\emptyset$, a fact that we use later on.

The rest of the phase is divided into segments. A segment is defined exactly as in the proof of Theorem 4.3 and consists of two page requests that force G to execute one rotation. Both segment requests cause a fault for G but no fault for H. Therefore, the adversary generates segments while $|\mathcal{M}(H)-\mathcal{M}(G)|\geq 2$. Notice that the adversary is adaptive, and so it can determine if $|\mathcal{M}(H)-\mathcal{M}(G)|\geq 2$, which pages to request in each segment, and in which order. However, the adversary cannot determine at the beginning of a phase the set F of pages that will make G fault. It remains to estimate the expected number of segments. At all steps, G keeps all the pages in the working set W except two. Let α and β be the two pages that are in the working set but not in G's fast memory immediately after the ith segment has been serviced. Notice that the adversary generates an $(i+1)$st segment only if $|\mathcal{M}(H)-\mathcal{M}(G)|\geq 2$, or, equivalently, only if $\{\alpha,\beta\}\cap\{r_1,r_2\}=\emptyset$. Therefore, G should choose α and β so that $\{\alpha,\beta\}\cap\{r_1,r_2\}\neq\emptyset$. In other words, G should choose the missing pages α and β in order to guess H's missing pages r_1 and r_2. Since $\{r_1,r_2\}\cap\{q+1,q+2\}=\emptyset$, we assume without loss of generality that G always keeps $q+1$ and $q+2$ in fast memory, and so $\{\alpha,\beta\}\cap\{q+1,q+2\}=\emptyset$. We will now estimate the expected number of segments by means of the following simple random experiment. We will have a bin that contains k balls—a white ball for each page in $\mathcal{M}(H,j)-\{r_1,r_2\}$ and a black ball each for r_1 and r_2. In each segment, G extracts two balls α and β from the bin. If both α and β are white, then G gets another round; otherwise, either α or β are black and the experiment stops. The number of segments ℓ is exactly the number of rounds needed to extract a black ball. Clearly, ℓ is minimum if no ball is ever replaced in the bin. Elementary combinatorics now yields $E[\ell]=\Theta(k)$. As an aside, we observe that the case when balls are not replaced in the bin corresponds to the case when G keeps in fast memory a different subset of the working set in different segments.

After the last segment, $|\mathcal{M}(H)-\mathcal{M}(G)|\leq 1$. If $|\mathcal{M}(H)-\mathcal{M}(G)|=0$, then $\mathcal{M}(G)$ contains the same pages as $\mathcal{M}(H)$, and another phase starts. If $|\mathcal{M}(H)-\mathcal{M}(G)|=1$, then the adversary keeps requesting pages in $\mathcal{M}(H)-\mathcal{M}(G)$ until $\mathcal{M}(G)$ contains the same pages as $\mathcal{M}(H)$. Again, the adversary is adaptive and knows if $\mathcal{M}(G)$ coincides with $\mathcal{M}(H)$. There are two cases, depending on whether this step ends or not. The expected cost of G conditioned to the event that $\mathcal{M}(G)$ never coincides with $\mathcal{M}(H)$ is infinite, and so G is not competitive. The expected cost of G conditioned to the event that $\mathcal{M}(G)$ eventually coincides with $\mathcal{M}(H)$ is nonnegative. We will assume from now on that indeed G's memory will eventually coincide with H's. After $\mathcal{M}(G)$ and $\mathcal{M}(H)$ contain the same pages, a new phase starts. The expected cost of G in a phase is at least equal to the expected cost in the segments, and so it is at least $nE[\ell]=\Omega(nk)$. Meanwhile, the cost of H in the entire phase is due only to the initial faults on $q+1$ and $q+2$, and so it is 2. The adversary issues $(n-k)/2$ request phases paying only a cost of two per phase. Then the adversary pays a cost of k and returns to the starting configuration $\{0,1,\ldots,\triangleright(k-1)\}$. It will then keep requesting pages $0,1,\ldots,k-1$ until the fast memory configurations of G and H coincide. Therefore, the cost of the adversary on $(n-k)/2$ phases is n, while the expected cost of G is $\Omega(nk(n-k))$, and the result follows. $\square$

We next establish an $\Omega(n\log k)$ lower bound on the competitive ratio of randomized online algorithms without prefetching against an oblivious adversary. We start with the following simple proposition.

LEMMA 4.5. *Let G be a deterministic lazy algorithm, $\sigma = (\langle \sigma_1, \sigma_2, \ldots \sigma_l \rangle)^h$ for some $l \leq k$. If $\{\sigma_1, \sigma_2, \ldots, \sigma_l\} - \mathcal{M}(G, lh) \neq \emptyset$, then $c(G, \sigma) \geq h - 1$ for all configurations $\mathcal{M}(G, 1)$.*

Proof. The proof is by induction on c. The claim clearly is true for $c = 1$. Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_l\}$ and $\sigma' = \langle \sigma_1, \sigma_2, \ldots, \sigma_l \rangle$. Suppose that $\Sigma - \mathcal{M}(G, l(h)) \neq \emptyset$. Since G is lazy, $\mathcal{M}(G)$ will remain unchanged as soon as $\Sigma \subseteq \mathcal{M}(G)$. Then, $\Sigma - \mathcal{M}(G, l(h-1)) \neq \emptyset$, and by induction hypothesis $c(G, (\sigma')^{h-1}) \geq h-2$. Moreover, G faults at least once during the last repetition of σ', and the lemma follows. ◻

THEOREM 4.6. *No randomized online algorithm without prefetching can have a competitive ratio better than $\Omega(n \log k)$ for $3 \leq k \leq bn$ and any fixed $b < 1$, even if the adversary is not allowed to do prefetching.*

By the minimax theorem [5], it will be enough to show that there is a probability distribution $\mathcal{P}$ over sequences of page requests such that for any lazy deterministic algorithm G, $E_{\mathcal{P}}\{c(G, \sigma)\} \geq c \cdot c(H, \sigma)$ where $c = \Omega(n \log k)$ and H is an optimal offline strategy.

Proof. The main difference between the proofs of Theorem 4.4 and Theorem 4.6 is that the adversary cannot foresee what pages α and β are in the working set but not in G's fast memory. We sidestep the difficulty by showing that with some probability the adversary is still able to force G to execute one rotation.

The request sequence. The page request sequence σ is of the form $\sigma = (\Gamma_1 \ldots \Gamma_{(n-k)/2} \Gamma_0)^l$, where l is a positive integer, $\Gamma_0 = \langle 0, 1, \ldots, k-1 \rangle$, and Γ_i $(1 \leq i \leq (n-k)/2)$ will be defined later. Roughly speaking, the subsequences Γ_i for $i \geq 1$ are the analogues of phases in the previous proofs, and Γ_0 corresponds to the final coordination step that brings about the initial configuration $\{0, 1, \ldots, \triangleright(k-1)\}$. We will now describe the phase Γ_i, a few quantities determined by the Γ_i's, and the value of these quantities at the beginning of Γ_i. A phase Γ_i depends on the set S_{i-1}, which is defined recursively: let $S_0 = \{0, 1, \ldots, k-1\}$, and S_i contains all the pages requested during the phase Γ_i. Throughout we will maintain the following properties:

- $|S_i| = k$ for $1 \leq i \leq (n-k)/2$.
- H holds S_{i-1} in fast memory immediately before the start of phase Γ_i.
- The pages $q_i + 1, q_i + 2 \notin S_{i-1}$, where $q_i = k - 1 + 2i$. In other words, we maintain that $q_i + 1$ and $q_i + 2$ are not in H's fast memory immediately before phase Γ_i starts. For example, k and $k+1$ are not in H's fast memory immediately before Γ_1.

Define the working set in phase Γ_i to be the set $W_i = S_{i-1} \cup \{q_i + 1, q_i + 2\}$. Let $\gamma_1, \gamma_2, \ldots, \gamma_k$ be a random k-permutation of the working set W_i; define $S_i = \{\gamma_1, \gamma_2, \ldots, \gamma_k\}$. The basic plan of the adversary is to request the pages in S_i in the given order $\gamma_1, \gamma_2, \ldots, \gamma_k$. However, the adversary will also request a sequence of other pages between the first request for a page γ_j and the first request for page γ_{j+1}, $1 \leq j \leq (k-1)$, in order to force G's memory configuration. Specifically, $\Gamma_i = \rho_0 \langle \gamma_1 \rangle \rho_1 \langle \gamma_2 \rangle \rho_2 \ldots \langle \gamma_k \rangle$, where

- ρ_0 is a repetition for $c+1$ times of S_{i-1}, and
- $\rho_j = (\langle \gamma_1, \gamma_2, \ldots, \gamma_j \rangle)^{c+1}$, where $1 \leq j \leq k$.

We will now turn to estimate the expected cost of G during Γ_i conditioned to the event that either (i) a page in S_{i-1} is not in G's cache immediately before the first request to γ_1, or (ii) one of the $\gamma_1, \ldots, \gamma_{j-1}$ is not in G's cache immediately before the first request to γ_j $(2 \leq j \leq k)$. Suppose that there is a page in S_{i-1} missing from G's cache immediately before the request to γ_1. By the previous lemma, G pays at least $c = \Omega(n \log k)$ on ρ_0. Analogously, suppose that there is one of $\gamma_1, \ldots, \gamma_{j-1}$

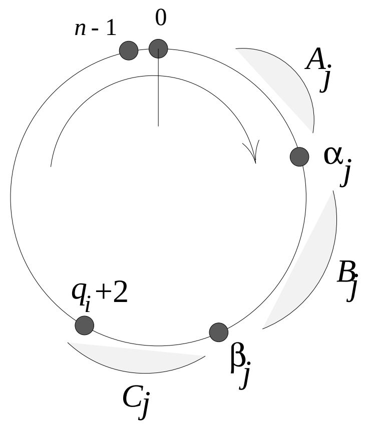


FIG. 4.1. *Lower bound for randomized algorithms without prefetching.*

missing from G's cache immediately before the request to γ_j $(1 \leq j \leq k)$. By the previous lemma, G pays at least $c = \Omega(n \log k)$ on ρ_{j-1}. Therefore, $\Omega(n \log k)$ bounds G's expected cost conditioned to the event that either (i) or (ii) occurred. It remains to show that G's expected cost is $\Omega(n \log k)$ even when neither (i) nor (ii) occurred. We will condition the rest of the analysis to such event, and so we will assume that G has $\gamma_1, \ldots, \gamma_{j-1}$ in fast memory before the request for γ_j where $j = 2, \ldots, k$, and G has all the elements in S_{i-1} before γ_1.

Costs of G and H. The adversary's algorithm H keeps S_i in fast memory throughout Γ_i. Therefore, H pays a cost of at most n on the sequence $\Gamma_1 \ldots \Gamma_{(n-k)/2}\Gamma_0$, exactly as in the previous proofs. We will now show that the expected cost of G on Γ_i $(1 \leq i \leq (n-k)/2)$ is $\Omega(n \log k)$. Let

$$t_j = \begin{cases} 1 & \text{if } G \text{ transits over page 0 while serving } \gamma_j, \\ 0 & \text{otherwise.} \end{cases}$$

The quantity t_j will be referred to as the *transit cost* of G for γ_j. Define $t = \sum_{j=1}^{k} t_j$ to be the transit cost of G in the phase Γ_i. Observe that G executes at least $t-1$ rotations during Γ_i, and so G's cost during Γ_i is at least $n(t-1)$. Hence, it is enough to show that $t = \Omega(\log k)$.

Before estimating t, we will make some definitions and observations. Let Y_{ij} be the set of pages in W_i that have not been requested prior to the first request for γ_j, that is, $Y_{ij} = W_i - \{\gamma_1, \ldots, \gamma_{j-1}\}$, $j = 1, \ldots, k$. Then by assumptions (i) and (ii) above, there are at least two pages in Y_{ij} that are missing from G's fast memory before γ_j is requested. Let $\alpha_j, \beta_j \in Y_{ij}$ be two such pages and assume without loss of generality that $\alpha_j < \beta_j$, as depicted in Figure 4.1. Notice that if G does not fault on γ_j, then G does not move and $\alpha_{j+1} = \alpha_j$ and $\beta_{j+1} = \beta_j$. Let r_j be the position taken by G before the request for γ_j. It can be seen that $r_j \in W_i$ by induction on j. Moreover, G is lazy and so $r_j \neq \alpha_j, \beta_j$. As depicted in Figure 4.1, let

- $A_j = \{p \in W_i : p < \alpha_j\}$,
- $B_j = \{p \in W_i : \alpha_j < p < \beta_j\}$, and
- $C_j = \{p \in W_i : \beta_j < p\}$.

Notice that $r_j \in A_j \cup B_j \cup C_j$.

We now turn to estimate the transit cost t_j. Define the potential of G immediately before γ_j as

$$\Phi(j) = \begin{cases} \frac{1}{2} & \text{if } r_j \in B_j \cup C_j, \\ 0 & \text{otherwise} \end{cases}$$

and the amortized transit cost of G as $t_j + \Phi(j+1) - \Phi(j)$. We will show that the expected amortized transit cost of G during Γ_i is $\Omega(\log k)$. Since the real transit cost of G is equal to the amortized transit cost minus $O(1)$, we obtain that the real transit cost of G during Γ_i is $t = \Omega(\log k)$. Putting together, it will then follow that the actual cost of G during Γ_i is $\Omega(n \log k)$.

If $r_j \in A_j$ and $\beta_j = \gamma_j$, then the potential increases by $1/2$. Moreover, $\beta_j = \gamma_j$ with probability $1/|Y_{ij}| = 1/(k+3-j)$, so that G's expected amortized transit cost when $r_j \in A_j$ is at least $(1/2)/(k+3-j)$. If $r_j \in B_j \cup C_j$ and $\alpha_j = \gamma_j$, then G pays a real transit cost at least equal to 1, and the potential drops by no more than $1/2$. Moreover, $\alpha_j = \gamma_j$ with probability $1/|Y_{ij}| = 1/(k+3-j)$, so that G's expected amortized transit cost when $r_j \in B_j \cup C_j$ is at least $(1/2)/(k+3-j)$. The expected amortized transit cost of G during Γ_i is at least

$$\sum_{j=1}^{k} \left(\frac{1}{2} \frac{1}{k+3-j} \right) = \frac{1}{2} \sum_{j=3}^{k+2} \frac{1}{j} = \Omega(\log k).$$

Consequently, $t = \Omega(\log k)$, and the actual cost of G during Γ_i is $\Omega(n \log k)$.

Finally, we recall that on $\Gamma_1 \Gamma_2 \ldots \Gamma_{(n-k)/2} \Gamma_0$, the cost of H is n, and the cost of G is $\Omega(n^2 \log k)$, which proves the theorem. □

Conversely, the competitive randomized algorithms in [3, 8, 16] immediately imply an $O(n \log k)$-competitive randomized algorithm for BDP even against a ubiquitous adversary.

5. BDP algorithms with prefetching.

In this section, we will establish our main theorems. We will first show a lower bound of $\Omega(n \log k)$ on the competitive ratio of any deterministic online algorithm that uses prefetching, even when compared to an adversary that does not do any prefetching. We will then present a deterministic online algorithm that achieves a competitive ratio of $O(n \log k)$ and therefore is optimal up to a constant factor.

5.1. A lower bound.

We first prove the lower bound.

THEOREM 5.1. *No deterministic algorithm for BDP can achieve a competitive ratio better than $\Omega(n \log k)$ when $3 \le k \le bn$ for any fixed $b < 1$ even if the adversary is not allowed to do prefetching.*

Proof. The main difference between the present proof and that of Theorem 4.3 is that G might reload α when it faults on β, and so the online algorithm cannot be made to execute one rotation every other request. In fact, we will typically need several requests to have the online algorithm complete a rotation. The basic plan of the adversary is as follows: repeatedly request a page that is not in the online algorithm G's fast memory and is farthest from the page where G is positioned. We will assume without loss of generality that G is lazy. Let H be the algorithm that the adversary uses to satisfy the request sequence.

The request sequence. The adversary proceeds in phases. We will maintain that, at the beginning of a phase, G and H hold the same set of pages W in fast memory and that H is positioned on a page q such that $q+1, q+2 \notin W$. In this proof, the set W will take the function that the working set had in the previous proofs, namely, we will extract requests from W in order to make G fault. The adversary starts the phase by requesting page $q+1$ followed by page $q+2$. We will maintain that there are always at least two pages in W that G does not have. The adversary will request one of those pages, and if r is the page where G is currently positioned, the adversary will issue a request for the missing page α that maximizes the quantity $\alpha - r$. However, the adversary will also insert other page requests to force G's fast memory configuration. On the whole, the phase has the form $\langle q+1, q+2\rangle \rho_1 \langle \gamma_1 \rangle \rho_2 \langle \gamma_2 \rangle \dots \rho_{k-2} \langle \gamma_{k-2} \rangle$, where

- γ_j is the page in W that is not in G's fast memory, and that is farthest from the current position r of G, $1 \le j \le k-2$,
- ρ_j denotes the sequence of requests $\rho_j = (\langle q+1, q+2, \gamma_1, \dots, \gamma_{j-1}\rangle)^{c+1}$, where $c = \Omega(n \log k)$ and $1 \le j \le k-2$.

Suppose that G and H do not have the same set of pages in fast memory after the request for γ_{k-2}. In that case, the adversary keeps requesting a page that H has, but G does not. Eventually, G and H will have the same set of pages in fast memory, or else G is not competitive, and another phase starts.

Costs of G and H in each phase. Let $R_j = \{q+1, q+2, \gamma_1, \dots, \gamma_{j-1}\}$ denote the set of pages requested in the sequence ρ_j. Notice that R_j is precisely the set of pages requested since the beginning of the phase until the request γ_j, and that $R_j \subset R_{j+1}$. The algorithm H keeps R_{k-1} throughout the phase and pays a cost of two. We can assume the following about the behavior of the algorithm G by Lemma 4.5: (i) G has R_j in its fast memory before the request for γ_j, and (ii) G does not move at all during the ρ_j's, but that it moves only to service the requests for the γ_j's. Thus from here on we can simply assume that a phase merely consists of the sequence of requests $\langle q+1, q+2, \gamma_1, \gamma_2, \dots, \gamma_{k-2}\rangle$. By this assumption, γ_{j+1} is the page in W that is not in G's fast memory and that maximizes $\gamma_{j+1} - \gamma_j$.

We will now show that the cost of G is $\Omega(n \log k)$ on the sequence $\Gamma = \langle q+1, q+2, \gamma_1, \gamma_2, \dots, \gamma_{k-2}\rangle$. To estimate G's cost, we will divide a phase into subphases with the property that G executes one rotation per subphase. Then, it will be enough to count the number of subphases. Specifically, we define subsequences $Z_1, Z_2, \dots, Z_t$ such that

- $Z_1 Z_2 \dots Z_t = \Gamma$, and
- $Z_1 Z_2 \dots Z_i$ is the smallest prefix of Γ on which G completes i rotations ($1 \le i < t$).

Since G executes at least $t-1$ rotations in the phase, it suffices to show that $t = \Omega(\log k)$. The idea is to show that $|Z_i|$ decreases as i increases, and then it will follow that t is large. Let x_i denote the number of pages in W that have not been requested prior to the start of the subsequence Z_i. Then we claim that $|Z_i| \le \lceil (x_i+1)/2 \rceil$. This easily follows from the observations that at each step G has at least 2 pages missing from the set W and we always request the missing page that is farthest. Thus G must complete a rotation after $\lceil (x_i+1)/2 \rceil$ requests.

Observe that $x_1 = k$ and that upon termination, $x_{t+1} = 2$, and so $2 \le x_i \le k$ for all $i = 1, 2, \dots, t+1$. Furthermore, $x_{i+1} = x_i - |Z_i| \ge \lfloor (x_i - 1)/2 \rfloor \ge (x_i/2) - 1$. It is then straightforward to see that $t = \Omega(\log k)$. The cost of G during a phase is thus at least $nt = \Omega(n \log k)$.

Putting it all together. Observe that the adversary can issue $(n-k)/2$ consecutive phases. Then, it will reload $\{0, 1, \dots, \triangleright(k-1)\}$ and another sequence of $(n-k)/2$ phases starts. The total cost of the adversary in such a sequence of phases is n while the total cost of G is $\Omega(n^2 \log k)$, and the result follows. $\square$

5.2. An upper bound: The gray algorithm. We will now design a paging algorithm $\mathcal{G}$, referred to as the *gray algorithm*, that is $O(n \log k)$-competitive. The gray algorithm uses a set of three marks {black, gray, white} and maintains a mark for each page in P. Notice that $\mathcal{G}$'s marking policy is somewhat different from that of the marking algorithm [13] that uses only two marks and marks only the pages in fast memory. We define b_j to be the number of black pages immediately before the jth request. The gray algorithm $\mathcal{G}$ works as follows:

- Initially, $\mathcal{G}$ marks the pages $\{0, 1, \dots, k-1\}$ black and all other pages white.
- $\mathcal{G}$ ignores all requests that do not cause a fault. Henceforth, assume that σ is hard for $\mathcal{G}$.
- $\mathcal{G}$ works in phases:
  - A new phase is started when $b_j = k$.
  - At the beginning of a phase, $\mathcal{G}$ marks all gray pages white and all black pages gray.
- When $\mathcal{G}$ faults on σ_j, $\mathcal{G}$ loads σ_j and marks it black.
- Before the jth request, $\mathcal{G}$ keeps in its fast memory
  - the b_j black pages, plus
  - the set of $q_j = k - b_j$ gray pages $\gamma_1, \gamma_2, \dots, \gamma_{q_j}$ that have the q_j smallest values of $\sigma_{j-1} - \gamma_i$ $(i = 1, 2, \dots, q_j)$. In other words, the gray pages $\gamma_1, \gamma_2, \dots, \gamma_{q_j}$ are the q_j gray pages that are most expensive to reload from σ_{j-1}.

We have stated the gray algorithm by assuming that a mark is associated with all n pages. In fact, $\mathcal{G}$ needs only to keep track of the black and gray pages, and a straightforward induction shows that, at any step, there are at most $2k$ gray and black pages.

It is not obvious that the gray algorithm could be implemented to maintain the prescribed set of gray pages without paying a large overhead. The following lemma shows that $\mathcal{G}$ is lazy.

LEMMA 5.2. *The gray algorithm $\mathcal{G}$ is lazy.*

Proof. Let σ be a hard sequence for $\mathcal{G}$. Suppose that $p \in \mathcal{M}(\mathcal{G}, j+1) - \mathcal{M}(\mathcal{G}, j)$ and $p \neq \sigma_{j+1}$. First, we show that p is gray at both step j and $j+1$. Since $p \notin \mathcal{M}(\mathcal{G}, j)$, p is not black at step j, and since $p \neq \sigma_{j+1}$, p is not black at step $j+1$. Since $p \in \mathcal{M}(G, j+1)$, p is not white at step $j+1$, and so it is not white at step j. It follows that p is gray at both steps. If p is in the interval from σ_j to σ_{j+1}, then it can be prefetched at no cost while $\mathcal{G}$ moves from σ_j to σ_{j+1}. We now show that no page that lies in the interval from σ_{j+1} to σ_j is prefetched by $\mathcal{G}$. Suppose, by contradiction, that such a page p is prefetched. First, we observe that σ_{j+1} does not start a new phase. Indeed, if σ_{j+1} started a new phase, then $\mathcal{M}(\mathcal{G}, j)$ is exactly the set of gray pages immediately before step $j+1$. However, p is gray and $p \notin \mathcal{M}(\mathcal{G}, j)$. We conclude that σ_{j+1} did not start a new phase. Therefore, $b_{j+1} = b_j + 1$ and so $q_{j+1} = q_j - 1$. Let ζ_j be the number of gray pages remaining after the jth request and notice that $\zeta_{j+1} \geq \zeta_j - 1$. Since $p \in \mathcal{M}(\mathcal{G}, j+1)$, p has one of the q_{j+1} largest values of $p - \sigma_{j+1}$ among all ζ_{j+1} pages that are gray at step $j+1$. Therefore, there are at least $\zeta_{j+1} - q_{j+1} \geq \zeta_j - q_j$ gray pages between σ_{j+1} and p. Since $p \notin \mathcal{M}(\mathcal{G}, j)$, p does not have one of the q_j largest values of $p - \sigma_j$ among all ζ_j gray pages at step j. Therefore, there are at most $\zeta = \zeta_j - q_j - 1$ gray pages between σ_j and p at step j. Since the number of gray pages never increases during a phase, there are at most ζ gray pages between σ_j and p at step $j+1$. However, $p - \sigma_{j+1} < p - \sigma_j$, which is to say that the interval from σ_{j+1} to p is contained in the interval from σ_j to p. Hence, there are at most $\zeta < \zeta_{j+1} - q_{j+1}$ gray pages between σ_{j+1} and p. Thus, we reach a contradiction and the lemma is proven. $\square$

Remark. The gray algorithm is similar to a marking algorithm once we identify black pages with the marked pages and white pages with the unmarked pages. However, the gray algorithm differs from the marking algorithm in one important respect: an evicted page might be prefetched and reloaded later on in the phase, without being requested again. By Lemma 5.2, the prefetch operation is executed at no cost. Thus, the gray algorithm adjusts the eviction pattern dynamically according to the requests in a phase. We would like to point out here that an analysis of the standard marking algorithm explicitly uses three marks and marks all n pages [3].

THEOREM 5.3. *The gray algorithm $\mathcal{G}$ is $O(n \log k)$-competitive even against a ubiquitous adversary.*

In the rest of the section, we will prove Theorem 5.3. Let H be the adversary's algorithm and assume without loss of generality that H is lazy. We will assume without loss of generality that the request sequence σ is hard for $\mathcal{G}$. No page is requested more than once in a phase and there are exactly k requests in a phase. Since all black pages are in $\mathcal{G}$'s fast memory, all requests are for gray and white pages. Correspondingly, we will say that a request is gray (white) if the requested page is gray (white) immediately before it is requested. The first request of a phase is white because all gray pages are in $\mathcal{G}$'s fast memory at the beginning of the phase.

The proof is structured as follows.

- We begin by defining the notion of a *segment*; segments allow us to give a lower bound on the cost of H.
- We proceed to examine $\mathcal{G}$'s cost on gray requests in a given segment, and show that it is at most $O(wn \log k)$, where w is the total number of white requests in the segment.
- Finally, we use a potential function argument to show that the amortized cost of $\mathcal{G}$ is no more than $O(n \log k)$ times the cost of H.

5.3. Segments.

We now define the notion of segments. Segments start from the second request of a phase and end with the first request of the next phase.

DEFINITION 5.4. *A* segment *is a subsequence of the form*

$$\langle \sigma_{ik+2}, \sigma_{ik+3}, \ldots, \sigma_{(i+1)k+1} \rangle$$

for some $i \geq 0$.

The notion of segment will be central to the rest of the proof because segments allow us to compute algorithm cost. We will estimate the cost of $\mathcal{G}$ and H on each segment and prove that the cost of $\mathcal{G}$ in a segment is no more than $O(n \log k)$ the cost of H in the same segment.

From now on, we will fix our attention on the first segment for simplicity of notation and without loss of generality. The first segment consists of the request sequence $\langle \sigma_2, \sigma_3, \ldots, \sigma_{k+1} \rangle$. Recall that σ_{k+1} is a white request.

5.4. Cost of *H*.

We turn now to examine the cost of H in a segment. Define w^N to be the number of white pages that are requested in the segment and that cause H to fault. Clearly, the cost of H is at least w^N. Suppose that on a white request for a page p, H already has p in its fast memory. Then the ratio of the actual costs of $\mathcal{G}$ and H for the request to p is infinity. This scenario thus requires a careful analysis as we describe below.

DEFINITION 5.5. *A page p is* hidden *at step j if p is white before the jth request and $p \in \mathcal{M}(H, j)$.*

Broadly speaking, our objective is to show that H also implicitly pays a cost on the hidden white pages. We will prove that the cost of H on the segment is at least D, where D is the number of hidden pages at step $k+1$.

LEMMA 5.6. *If a page p is hidden at step $k+1$, then p has not been requested in the segment.*

Proof. If $p \in \{\sigma_2, \dots, \sigma_k\}$, then p is black after step k and becomes gray before step $k+1$. Therefore, p is not white at step $k+1$ and so p is not hidden at step $k+1$. ∎

Let p be a hidden page at step $k+1$. Notice that $p \in \mathcal{M}(H, k+1)$, p was not requested in the segment, and H has not prefetched p. We conclude that p has been in $\mathcal{M}(H)$ throughout the segment.

LEMMA 5.7. *The cost of H in a segment is at least D.*

Proof. The proof is reminiscent of those in [12, 13, 18]. Clearly, $\sigma_1 \in \mathcal{M}(H, 2)$. Moreover, the previous lemma implies that all D pages hidden at step $k+1$ are in $\mathcal{M}(H, 2)$. Within a segment, the requests $\sigma_2, \dots, \sigma_k$ are for pages that are not hidden at step $k+1$ and that are not σ_1. At most $k-D-1$ of those $k-1$ pages are in $\mathcal{M}(H, 2)$, and so the cost of H is at least $k-1-(k-D-1) = D$. ∎

On the whole, the cost of H in the segment is at least $\max\{w^N, D\} \geq (w^N + D)/2$.

5.5. Potential function. Let us turn now to evaluate the cost of $\mathcal{G}$ in the same segment. Let $\gamma = 3(n-1) + n \ln k$. We will denote by D_j the number of hidden pages at step j, and so $D = D_{k+1}$. Define the potential function at step j to be

$$\Phi(j) = (3(n-1) + n \ln k) D_j = \gamma D_j.$$

Clearly, $\Phi(j) \geq 0$ for all j's. We will analyze the amortized cost of $\mathcal{G}$ by considering the following cases:

- In the first case, $\mathcal{G}$ does not pay any real cost, but the potential might increase as a consequence of an increase of the number D_j of hidden pages.
- In the next two cases, $\mathcal{G}$ indeed pays a real cost.
  - We will examine the cost of $\mathcal{G}$ on gray requests, and finally
  - we will examine the cost of $\mathcal{G}$ on white requests.

5.6. Potential increase. First, we show that the potential increases only at the end of a phase. We will need the following lemma.

LEMMA 5.8. *If a page p is hidden at step j ($2 \leq j \leq k$), then p was hidden at step 2.*

Proof. It is enough to show that if a page p is hidden at step j $(2 < j \leq k+1)$, then p was hidden at step $j-1$. Suppose to the contrary that p is hidden at step j, but not at step $j-1$. Then, either p is not white at step $j-1$ or $p \notin \mathcal{M}(H, j-1)$. However, $j \leq k$, so that no gray page has turned white. Hence, it must be the case that $p \notin \mathcal{M}(H, j-1)$. Since H does not prefetch, $p = \sigma_{j-1}$, and so p is black at step j, which is a contradiction, and the lemma is proven. ∎

Lemma 5.8 implies that $D_{j-1} \geq D_j$ for $j = 2, \dots, k$. It follows that the potential increase increases only at the end of a phase when some gray pages become hidden white pages. The potential increase is γD.

5.7. Gray requests. We now evaluate $\mathcal{G}$'s cost on the gray requests. Define a *gray block* as a maximal sequence of gray requests followed by a white request. The segment can be partitioned into a sequence of gray blocks that alternate with white requests. Notice that there are at most w gray blocks in a phase, where w was defined as the number of white requests in the segment. Actually, the first phase request and the last segment request are both white, so that w is also the number of white requests in the phase. We will argue that the cost incurred by $\mathcal{G}$ during the gray blocks does not exceed $w(2(n-1) + n \ln k)$.

Consider a gray block $B_i = \langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_{j+q} \rangle$. Since σ_{k+1} is a white request, $j + q \leq k$. $\mathcal{G}$'s cost for σ_j is at most $n - 1$. We now estimate $\mathcal{G}$'s cost in the rest of the block. In what follows, we denote

- by α_l, the number of gray pages that are missing from $\mathcal{G}$'s fast memory before the lth request ($j \leq l \leq j + q$),
- by β_l, the number of pages that are gray at the lth request ($j \leq l \leq j + q$),
- by g_l, the number of gray pages requested from the beginning of the phase up to step $l - 1$ inclusive,
- by w_l, the number of white pages requested from the beginning of the phase up to step $l - 1$ inclusive, and
- by $b_l = w_l + g_l$, the number of black pages before the lth request.

The notation β_l is related to that in Lemma 5.2 by $\beta_l = \zeta_{l-1}$. Notice that $\beta_l = k - g_l$ and that $\alpha_l = \beta_l - q_l = k - g_l - (k - b_l) = g_l + w_l - g_l = w_l = w_j$ is a constant throughout the current gray block. Moreover, $\alpha_l \leq w_{k+1}$. Observe that w_{k+1} is the number of white requests in $\langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$, w is the number of white requests in $\langle \sigma_2, \ldots, \sigma_k, \sigma_{k+1} \rangle$, and both σ_1 and σ_{k+1} are white requests. Therefore, $w_{k+1} = w$ and $\alpha_l \leq w$.

$\mathcal{M}(\mathcal{G}, l)$ contains all but at most w gray pages, and the missing gray pages are the closest to its current position σ_{l-1}. Thus, once $\mathcal{G}$ is positioned at σ_{l-1}, no more than w gray pages lie in the closed interval between σ_{l-1} and σ_l ($l = j + 1, \ldots, j + q$). Hence, if $\mathcal{G}$ starts from σ_j and moves for more than n units, then there are at most $\beta_j - \left\lceil \frac{\beta_j + 1}{w} \right\rceil \leq \beta_j \left(1 - \frac{1}{w}\right)$ pages that are still gray. Hence, if during the block B_i, $\mathcal{G}$ pays $r_i n + a$, for some $a < n$, then $\beta_{j+q+1} \leq (1 - 1/w)^{r_i} \beta_j$. On the other hand, since $\beta_2 = k$, if $\sum r_i > w \ln k$, no gray pages could be remaining since

$$\left(1 - \frac{1}{w}\right)^{\sum r_i} k < \left(\frac{1}{e}\right)^{\ln k} k = 1.$$

We conclude that $\sum r_i \leq w \ln k$. Notice that the quantity a contributes for at most $n - 1$ per gray block to the cost on gray requests. Recall that σ_j contributed for another $n - 1$ term per gray block. Since there are at most w gray blocks, the cost paid during all gray blocks is at most $2w(n - 1) + n \sum r_i \leq w(2(n - 1) + n \ln k)$.

5.8. White requests.

Henceforth, we will assume that $\mathcal{G}$ processes gray requests for free, but each white request is charged $2(n - 1) + n \ln k$. Such a charge is in addition to the cost required to process the white request itself. Notice that on any white request, $\mathcal{G}$ pays a real cost of at most $n - 1$, is charged $2(n - 1) + n \ln k$ for gray requests, and so $\mathcal{G}$'s amortized cost is $3(n - 1) + n \ln k = \gamma$ plus any increase of the potential function. Suppose that σ_j is a white request that causes H to fault. The potential does not increase, and $\mathcal{G}$'s amortized cost is γ. Suppose now that σ_j is a white request that does not cause H to fault. Then, p is hidden, the potential decreases by γ, and $\mathcal{G}$'s amortized cost is naught. Finally, G pays a cost of γD at the end of the phase. On the whole, $\mathcal{G}$'s cost in the segment is $w^N \gamma + D\gamma$, which is no more than $2\gamma = O(n \log k)$ times the actual cost of H in the segment, and the proof is complete.

6. Delay model.

We will now describe the delay model for BDP. In the delay model, the adversary has the power of issuing requests of two types: page requests, like in the ordinary BDP, and delays, where one delay request forces any algorithm to listen to one more page. Hence, if the algorithm G is positioned over page i before a delay request, G will be positioned over $i+1$ after the delay request. The algorithm G is free to decide whether page $i+1$ should be cached or not. Let G be an algorithm for BDP. The algorithm G can be turned into a BDP algorithm G^D in the delay model as follows. On a page request, G^D behaves exactly as the algorithm G would on that request. However, when G^D receives a delay request, it executes at most one full rotation after serving the delay request and returns to the configuration that it had before the delay.

Proposition 6.1. *If G is c-competitive for BDP against a ubiquitous adversary for some $c = \Omega(n)$, then G^D is $O(c)$-competitive in the delay model.*

Proof. Let σ^D be a request sequence consisting of page requests and delays, σ the request sequence where all delays have been removed, and d the number of delay requests in σ^D. Let f be the minimum number of faults on σ and H be the adversary algorithm. Observe that $c(H,\sigma^D) \geq f + d$. Then, $c(G^D,\sigma^D) = c(G,\sigma) + nd \leq cf + nd + b \leq O(c)(f+d) + b \leq O(c)c(H,\sigma^D) + b$ for some constant b. It is then proved that G is $O(c)$-competitive. □

An immediate corollary of the above proposition is as follows.

Corollary 6.2. *There is an $O(n \log k)$-competitive randomized algorithm for the delay model with no prefetching and an $O(n \log k)$ deterministic algorithm for the delay model with prefetching.*

Proof. The first result follows from the $O(\log k)$-competitive lazy randomized algorithm for virtual memory paging [3, 8, 16], whereas the second result follows from the fact that the gray algorithm is $O(n \log k)$-competitive against an ubiquitous adversary. □

7. Concluding remarks. We studied deterministic as well as randomized algorithms for broadcast disk paging. An interesting question not resolved by our work is that of the competitive ratio of randomized prefetching algorithms. A lower bound of $\Omega(n)$ is easy to show. It is conceivable that a simultaneous use of randomization and prefetching yields an algorithm that is $o(n \log k)$-competitive.

Recently, the second author performed an empirical evaluation of the gray algorithm on both synthetic and Web traces and found that the gray algorithm always and consistently outperformed the least recently used (LRU) algorithm [14].

Acknowledgments. We thank Martin Farach-Colton for many discussions, Uli Kremer and Stefan Langerman for telling us about the German and Belgian videotext systems, and two anonymous referees for helping us to improve the presentation of this paper.

REFERENCES

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, *Broadcast disks: Data management for asymmetric communication environments*, in Proceedings of the ACM SIGMOD Conference, San Jose, CA, 1995, pp. 199–210.

[2] S. Acharya, M. Franklin, and S. Zdonik, *Prefetching from a broadcast disk*, in Proceedings of the International Conference on Data Engineering, New Orleans, LA, 1996, pp. 276–285.

[3] D. Achlioptas, M. Chrobak, and J. Noga, *Competitive analysis of randomized paging algorithms*, in Algorithms—ESA '96, Barcelona, Lecture Notes in Comput. Sci. 1136, Springer-Verlag, Berlin, 1996, pp. 419–430.

[4] L. A. Belady, *A study of replacement algorithms for a virtual storage computer*, IBM Systems J., 5 (1966), pp. 78–101.

[5] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, UK, 1998.

[6] T. G. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib, *The Datacycle architecture*, Comm. ACM, 35 (1992), pp. 71–81.
[7] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, *A study of integrated prefetching and caching strategies*, in Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 1995, pp. 188–196.
[8] A. Fiat, R. Karp, M. Luby, L. A. McGeoch, D. Sleator, and N. Young, *Competitive paging algorithms*, J. Algorithms, 12 (1991), pp. 685–699.
[9] M. Franklin and S. Zdonik, *A framework for scalable dissemination-based systems*, in Proceedings of the International Conference Object Oriented Programming Languages Systems, 1997, pp. 94–105.
[10] D. K. Gifford, *Polychannel systems for mass digital communications*, Comm. ACM, 33 (1990), pp. 141–151.
[11] T. Imielinski and B. Badrinath, *Mobile wireless computing: Challenges in data management*, Comm. ACM, 37 (1994), pp. 18–28.
[12] S. Irani and A. R. Karlin, *Online computation*, in Approximation Algorithms for NP-Hard Problems, D. S. Hochbaum, ed., PWS Publishing, Boston, MA, 1997, pp. 521–564.
[13] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 79–119.
[14] V. Liberatore, *Caching and Scheduling for Broadcast Disk Systems*, Technical Report 98-71, UMIACS, University of Maryland, College Park, MD, 1998.
[15] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for server problems*, J. Algorithms, 11 (1990), pp. 208–230.
[16] L. A. McGeoch and D. D. Sleator, *A strongly competitive randomized paging algorithm*, Algorithmica, 6 (1991), pp. 816–825.
[17] E. Sigel, *Videotext: The Coming Revolution in Home/Office Information Retrieval*, Knowledge Industry Publications, White Plains, NY, 1980.
[18] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

SIAM J. COMPUT.
Vol. 29, No. 5, pp. 1703–1739

CONTENTION RESOLUTION IN HASHING BASED SHARED MEMORY SIMULATIONS[*]

ARTUR CZUMAJ[†], FRIEDHELM MEYER AUF DER HEIDE[†], AND VOLKER STEMANN[‡]

Abstract. In this paper we study the problem of simulating shared memory on the distributed memory machine (DMM). Our approach uses multiple copies of shared memory cells, distributed among the memory modules of the DMM via universal hashing. The main aim is to design strategies that resolve contention at the memory modules. Extending results and methods from random graphs and very fast randomized algorithms, we present new simulation techniques that enable us to improve the previously best results exponentially. In particular, we show that an n-processor CRCW PRAM can be simulated by an n-processor DMM with delay $\mathcal{O}(\log\log\log n \log^* n)$, with high probability.

Next we describe a general technique that can be used to turn these simulations into time-processor optimal ones, in the case of EREW PRAMs to be simulated. We obtain a time-processor optimal simulation of an $(n \log\log\log n \log^* n)$-processor EREW PRAM on an n-processor DMM with delay $\mathcal{O}(\log\log\log n \log^* n)$, with high probability. When an $(n \log\log\log n \log^* n)$-processor CRCW PRAM is simulated, the delay is only by a $\log^* n$ factor larger.

We further demonstrate that the simulations presented can not be significantly improved using our techniques. We show an $\Omega(\log\log\log n / \log\log\log\log n)$ lower bound on the expected delay for a class of PRAM simulations, called topological simulations, that covers all previously known simulations as well as the simulations presented in the paper.

Key words. PRAM, distributed memory machine, randomized shared memory simulations, hashing

AMS subject classifications. 68Q05, 68Q10, 68Q25

PII. S009753979529564X

1. Introduction.

Parallel machines that communicate via a shared memory (*parallel random access machines, PRAMs*) are the most commonly used theoretical machine model for describing parallel algorithms (see, e.g., [16, 18, 28]). A PRAM consists of p processors $P_0, \ldots, P_{p-1}$, each having local memory, and a shared memory with cells $U = \{0, \ldots, m-1\}$. The processors work synchronously and have random access to the shared memory cells, each of which can store an integer. In this paper we deal only with *exclusive read exclusive write (EREW)* PRAMs and (ARBITRARY) *concurrent read concurrent write (CRCW)* PRAMs. On the EREW PRAM no pair of processors can simultaneously write to or read from the same memory location. On the (ARBITRARY) CRCW PRAM concurrent reading is allowed and, if several processors want to write to the same memory cell simultaneously, an arbitrary one of them succeeds. The PRAM is relatively comfortable to program, because the programmer does not have to allocate storage within a distributed memory or specify interprocessor communication. On the other hand, shared memory machines are very

[*]Received by the editors December 7, 1995; accepted for publication (in revised form) March 2, 1999; published electronically March 17, 2000. This work was supported by DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik," ME 872/4-1, by DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen," by DFG Leibniz grant Me872/6-1, and by EU ESPRIT long term research project 20244 (ALCOM-IT).

http://www.siam.org/journals/sicomp/29-5/29564.html

[†]Heinz Nixdorf Institute and Department of Computer Science, University of Paderborn, D-33095 Paderborn, Germany (artur@uni-paderborn.de, fmadh@uni-paderborn.de).

[‡]Heinz Nixdorf Institute, University of Paderborn, D-33095 Paderborn, Germany. Current address: Deutsche Morgan Grenfell, Frankfurt, Germany.

unrealistic from the technological point of view, because on large machines a parallel shared memory access can only be realized at the cost of a significant time delay.

A more realistic theoretical model is the *distributed memory machine (DMM)*, in which the memory is divided into a limited number of memory modules, one module per processor. A DMM has n processors $Q_0, \dots, Q_{n-1}$ which are connected by an interconnection network with a distributed memory consisting of n memory modules $M_0, \dots, M_{n-1}$. In this paper we study DMMs with the *complete interconnection network* between processors and modules (cf. [7]). The computation of our DMM is synchronized. In a step each processor either performs a local computation or issues a read or write request for a memory cell x to the module holding x. Each module answers incoming requests. In this paper we shall focus on the DMM in which the modules obey the ARBITRARY conflict resolution rule: If more than one request is directed to a module, it serves an arbitrary one of them and ignores the others. However, the answer to the successful request is available to all processors accessing the module. It is easy to observe that an n-processor DMM can be simulated with constant delay on an n-processor ARBITRARY CRCW PRAM with $\Theta(n)$ shared memory cells and vice versa. Motivated by optical crossbar technology, there is another conflict resolution rule considered in the literature, the c-COLLISION rule. In this model a module can only answer if it gets at most c requests; otherwise a collision symbol is sent to all processors that wanted to access the module.

No matter which conflict resolution rule is used for a DMM, a module can respond to at most a constant number of accesses at a time. Thus DMMs exhibit the phenomenon of *memory contention*, in which an access request is delayed because of concurrent requests to the same module.

In an effort to understand the effects of memory contention on the performance of parallel computers, several authors have investigated the simulation of shared memory machines on DMMs. Often the authors assumed that processors and modules are connected by a bounded degree network (e.g., by a mesh, a butterfly, or an expander), and packet routing is used to access the modules [17, 20, 21, 27, 31]. In this paper we consider DMMs with a complete interconnection between processors and modules, i.e., we focus on the issue of resolving memory contention.

All of our algorithms are randomized, and the time bounds hold *with high probability* (*w.h.p.*), i.e., with probability at least $1 - n^{-\alpha}$ for arbitrary constant $\alpha > 1$; the choice of α will affect the respective running times by at most a constant factor. We focus on simulations that minimize the *delay*, i.e., the time needed to simulate a parallel memory access of a PRAM on a DMM. Furthermore, we are interested in optimal simulations. We say a simulation of a p-processor PRAM on an n-processor DMM is *time-processor optimal* if the delay is $\mathcal{O}(\lceil p/n \rceil)$.

The most efficient simulations of shared memory are based on the idea of hashing, i.e., of distributing the shared memory cells of the PRAM (almost) randomly among the memory modules of the DMM. Mehlhorn and Vishkin [24] design a simple simulation of an n-processor EREW PRAM by an n-processor DMM with expected delay $\mathcal{O}(\log n / \log\log n)$, by storing each cell of the PRAM in a module that is determined by a single hash function. One can easily show that this result cannot be improved when each cell of the PRAM is represented only by one copy. Dietzfelbinger and Meyer auf der Heide [7] extend this result to the CRCW PRAM model and show that one step of an n-processor CRCW PRAM can be simulated by an n-processor DMM with expected delay $\mathcal{O}(\log n / \log\log n)$. Karp, Luby, and Meyer auf der Heide [19] break the $\log n / \log\log n$ bound by applying the idea of redundant storage representation, that is, by storing in the DMM more than one copy of each memory cell of the PRAM. In order to distribute some number of copies of each memory cell, randomly and independently chosen hash functions are used. Karp, Luby, and Meyer auf der Heide obtain a simulation of an n-processor CRCW PRAM on an n-processor DMM with delay $\mathcal{O}(\log\log n)$. They present how to get time-processor optimal simulations. These authors show that any simulation of one step of an n-processor EREW PRAM on an n-processor DMM with delay τ that uses only a constant number of hash functions can be turned into a time-processor optimal simulation with delay $\mathcal{O}(\tau \log^* n)$. In the case of CRCW PRAMs, the simulation is close to optimal: One step of an $(\tau \cdot n)$-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\tau \log^* n)$. Using the majority technique due to Upfal and Wigderson [32], Dietzfelbinger and Meyer auf der Heide [8] extend the $\mathcal{O}(\log\log n)$-delay simulation of Karp, Luby, and Meyer auf der Heide [19] to a much simpler schedule for an $\mathcal{O}(\log\log n)$-time simulation on the weaker c-COLLISION DMM, for some constant $c > 2$. Goldberg, Matias, and Rao [12] show that one can perform a time-processor optimal simulation with delay $\mathcal{O}(\log\log n)$, even on a 1-collision model (called also *optical communication parallel computer*, OCPC). Meyer auf der Heide, Scheideler, and Stemann [25] extend the algorithm from [8] and present a simulation on a DMM (with the ARBITRARY write conflict resolution rule) achieving delay $\mathcal{O}(\log\log n/\log\log\log n)$. This is the first simulation that beats the $\log\log n$ bound, however, it can not be turned into a time-processor optimal one because it uses nonconstant storage redundancy, that is, each memory cell of the PRAM has a nonconstant number of copies in the memory modules of the DMM. The techniques used in these papers do not seem to yield simulations with smaller delay. In particular, MacKenzie, Plaxton, and Rajaraman [22] and, independently, Meyer auf der Heide, Scheideler, and Stemann [25] show lower bounds for classes of algorithms that capture all these algorithms.

In this paper we design new shared memory simulations that improve all previously known results by an exponential decrease of the delay. The core of our simulations is a new analysis of a special sparse almost random graph, the access graph, which represents requests of the PRAM processors and an efficient use of log-star-time randomized algorithms, which leads to fast techniques for exploring neighborhoods of nodes in such sparse random graphs. The key steps of our simulations are partitioning techniques that make it possible to decompose every connected component of a random sparse graph into small pieces. Using different, more and more sophisticated partitioning techniques, we design simulations of one step of n-processor EREW PRAMs on n-processor DMMs with delay $\mathcal{O}(\log\log n/\log\log\log n)$, $\mathcal{O}(\sqrt{\log\log n}\log^* n)$, and finally $\mathcal{O}(\log\log\log n \log^* n)$, refining the simulation techniques step by step. Finally we transform all these bounds into simulations of one step of n-processor CRCW PRAMs on n-processor DMMs without asymptotic time loss.

Next we present a general technique that can be used to transform any simulation of an n-processor EREW PRAM on an n-processor DMM with delay $\tau \geq \log^* n$ that uses a constant number of hash functions into a time-processor optimal simulation. Our transformation relies on a new routing problem, called *all-but-linear routing*, which is a relaxed version of the $k - k$ relation routing problem. Using the ideas for $k - k$ relation routing developed for the OCPC model [1, 11], we show how to solve all-but-linear routing with random or almost random requests optimally for any $k = \Omega(\log^* n)$.

Finally we pinpoint the limit of our techniques. We analyze a *topological game* in graphs that is the essential part of the most efficient previously known simulations

[8, 12, 19, 22, 25], as well as the simulations presented in this paper. We show that a randomized topological game requires $\Omega(\log\log\log n/\log\log\log\log n)$ expected delay. This indicates that the techniques presented in the paper cannot lead to significantly better simulations than the ones presented in this paper.

Organization of the paper. We begin in section 2 with outlining techniques used by our simulations. Section 3 provides some basic tools. In section 4 we define the access graph and the access game and prove some properties about the distribution of sizes of connected components in a random graph that are essential for our proofs of the running time of the simulations. Section 5 contains the first simulation of an EREW PRAM, which has delay $\mathcal{O}(\log\log n/\log\log\log n)$, w.h.p. In section 6 we present two simulations of an EREW PRAM, one with delay $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ and another with delay $\mathcal{O}(\log\log\log n\log^* n)$, w.h.p. Section 7 describes a transformation that turns EREW PRAM simulations into CRCW PRAM simulations. Section 8 analyzes the all-but-linear routing problem which is used in section 9 to get time-processor optimal simulations. In section 10 we present the lower bound for the topological game.

2. Outline of techniques.

We start with the simulation of an n-processor EREW PRAM on an n-processor DMM. Our shared memory simulations are based on *redundant storage representation*; that is, we assume that the shared memory cells of the PRAM are distributed among the modules of the DMM using some number a of hash functions $h_1,\dots,h_a : U \to \{0,\dots,n-1\}$, so that copies of cell $u \in U$ are stored in the modules $M_{h_1(u)},\dots,M_{h_a(u)}$. All such simulations assume that $h_1,\dots,h_a$ are randomly chosen from a *high performance universal class* of hash functions as presented, e.g., by Siegel [29] or Karp, Luby, and Meyer auf der Heide [19]. A function randomly chosen from such a class of hash functions behaves almost like a random function but can be stored using little ($\mathcal{O}(\sqrt{n})$) space and can be evaluated in constant time. Actually, any $(2,\log^2 n)$-universal class of hash functions (cf. [6]) would be sufficient for our purposes. Upfal and Wigderson [32] observe how to use redundant storage representation in order to speed up shared memory simulations. They introduce the majority technique which utilizes the observation that accessing more than half of the a copies of a requested shared memory cell is sufficient for reading as well as for writing: If processor P wants to write to cell u, it updates more than half of the copies of u and adds a time-stamp indicating the PRAM time. If processor P wants to read cell u it reads at least half of the copies of u and takes the copy with the latest time-stamp.

Let us refer to the task of accessing b out of the a copies of each of the n requested shared memory cells as the *"b out of a" task.* A method for performing a "b out of a" task is called a *protocol.* For the analysis it is more convenient to consider a "1 out of c" task. It is possible to reduce a "b out of a" task to $\binom{a}{b-1}$ "1 out of $a-b+1$" tasks, each with a different subset of $a-b+1$ hash functions. For constant a this yields only a constant factor in the delay. In this paper we present shared memory simulations based on executing the "2 out of 3" task. In fact, because of the reasons mentioned above, we only analyze protocols for the "1 out of 2" task. From now on we will assume that each memory cell u of the PRAM is stored in the modules $M_{h_1(u)}$ and $M_{h_2(u)}$.

Our protocols for the "1 out of 2" task are based on the model introduced by Karp, Luby, and Meyer auf der Heide [19]. Let ε be a constant, $0 < \varepsilon < 1$, that will be specified later. Consider a batch of εn requests, for which the "1 out of 2" task has to be executed. For such a batch, we define the labeled *access graph H* as follows. Its nodes correspond to the n modules of the DMM, and for each key u from the batch, H contains an edge labeled u between $M_{h_1(u)}$ and $M_{h_2(u)}$. Now a protocol for the "1 out of 2" task can be viewed as an *access game* on H. The access game is performed in rounds, and in each round each node of H (i.e., each module in the DMM) can remove one of its incident edges (i.e., one of the access requests directed to the module is processed). The access game is finished when all the edges have been removed from the graph.

2.1. Fast protocols for the "1 out of 2" task. The simple protocol for the "1 out of 2" task, as in [8, 12, 19, 25], corresponds to the access game in which, in each round, each node of H removes an arbitrary incident edge, i.e., processes one arbitrary request directed to it. MacKenzie, Plaxton, and Rajaraman [22] and Meyer auf der Heide, Scheideler, and Stemann [25] prove the tight lower bound of $\Omega(\log\log n)$ for the time needed by this protocol.

For the analysis of the access game or the protocol for the "1 out of 2" task, respectively, we also follow the idea of Karp, Luby, and Meyer auf der Heide [19], who analyze the structure of the access graph H. As h_1 and h_2 are almost random (i.e. randomly chosen from a high performance universal class), H is almost a random graph with n nodes and εn edges. In analogy to results on truly random graphs, these authors [19] show that H consists of connected components of size $\mathcal{O}(\log n)$, w.h.p., each of which is a tree with a constant number of additional edges, w.h.p. One can show (compare also [19]) that this property of H implies the existence of a schedule that works in constant time. The previously best algorithm to find this schedule takes $\mathcal{O}(\log\log n)$ time and yields the simulation from [8]. In contrast to formerly analyzed protocols for the access game, the heart of our simulations lies in finding fast protocols to execute the access game on H. In order to make the description of our protocols more intuitive, we assume that also the *module* may take part in the computation. Clearly this assumption is not critical, because we could let this work be carried out by the corresponding processor. Thus we assign the processors of the DMM both to the nodes and to the edges of H and allow the nodes (the modules) to decide which edges (if any) are to be removed. However, the knowledge about H is distributed among the processors at the beginning of the simulation, so that each processor knows just one edge. The only way the nodes may decide which edge to remove is to analyze their neighborhoods in H. There are two issues we have to deal with. First, each node has to explore its neighborhood quickly, and second, we have to find a rule to guide the nodes in their decision on which edge to remove on the basis of the structure of their neighborhood.

Exploring the neighborhood seems to be a complicated task. Even computing the degree of a node of H is complicated, because the edges in H are given in an arbitrary order and we do not have the adjacency list at hand. We show how each node v may explore certain properties of its k*-neighborhood*, i.e., the subgraph of H induced by the nodes at distance at most k from v, in time close to $\mathcal{O}(\log k)$. To succeed in this, we present a new precise analysis of the structure of H and apply extremely fast algorithms. We use probabilistic tools like the method of bounded differences [23] and a generalization of the Markov inequality to show that, w.h.p., for all i, the number of connected components of H of size at least i is bounded by $n/2^{bi}$ for some constant b.

Based on log-star techniques as developed in [3, 10, 14], we show how to use the structural properties of H mentioned above to compute certain properties of the k-neighborhood of all nodes of H in time $\mathcal{O}(\log k \log^* n)$. We then present a sequence of more and more involved protocols that use such information to design fast access protocols.

We can extend all our results to the simulation of the CRCW PRAM. It is well known that by applying integer sorting to the memory requests issued in one step of the CRCW PRAM, one can reduce the simulation of one step of the CRCW PRAM to that of one step of the EREW PRAM. We observe that instead of integer sorting one can use an algorithm for *strong semisorting* [3], which can be solved on the DMM much faster than sorting. This allows us to turn any simulation of one step of an n-processor EREW PRAM by an n-processor DMM with delay $\tau \geq \log^* n$ into a simulation of one step of an n-processor CRCW PRAM on an n-processor DMM with delay $\mathcal{O}(\tau)$.

2.2. Time-processor optimal simulations. We present a general method for making simulations based on hashing multiple copies of the PRAM memory cells time-processor optimal. Consider a simulation of one step of an n-processor EREW PRAM on an n-processor DMM with delay not exceeding τ, w.h.p., that works by solving the "b out of a" task with $b > a/2$ (cf. the discussion in section 2). Further, suppose that it solves the "b out of a" task by running $\binom{a}{b-1}$ "1 out of $a - b + 1$" tasks. We show that if a and b are constants, then the simulation can be made time-processor optimal with delay $\mathcal{O}(\tau)$ for $\tau \geq \log^* n$. This improves upon the $\mathcal{O}(\tau \log^* n)$ delay achieved in [19].

If we want to simulate one step of an $(n \cdot \tau)$-processor EREW PRAM on an n-processor DMM, each processor of the DMM simulates τ processors of the EREW PRAM. Hence, each processor of the DMM has a list of τ memory requests. We use $a + 2$ hash functions and perform the simulation by solving the "$b + 1$ out of $a + 2$" task. By our discussion in section 2, one can solve the task by performing a "1 out of $a - b + 2$" task for each subset of size $a - b + 2$ of the $a + 2$ hash functions. Therefore we focus on solving the "1 out of $a - b + 2$" task. Thus let us suppose that we have $a - b + 2$ hash functions. Our protocol consists of two phases. In the first phase, all but $\mathcal{O}(n)$ requests are satisfied in time $\mathcal{O}(\tau)$, using only one of the hash functions. We achieve this goal by analyzing a new routing problem, called all-but-linear routing. In the second phase we first evenly redistribute the remaining requests among the DMM processors such that each processor only gets a constant number of requests. Then we use the other $a - b + 1$ hash functions and perform a protocol for the "1 out of $a - b + 1$" task to satisfy the remaining requests. Since each processor has a constant number of requests to be sent, we may use the solution for the "1 out of $a - b + 1$" task to perform this step in time $\mathcal{O}(\tau)$, w.h.p.

2.3. Lower bound for topological simulations. Our solutions for the access game, as well as all previously known solutions to the "1 out of 2" task, are special cases of a *topological game* on the access graph. In this game, in order to remove all the edges from the graph, each node first analyzes its neighborhood and then removes its incident edges in a way depending on the topology of the neighborhood. We prove an $\Omega(\log \log \log n / \log \log \log \log n)$ lower bound for the topological game, which indicates that our $\mathcal{O}(\log \log \log n \log^* n)$-delay PRAM simulation is almost optimal in the class of algorithms based on the topological game.

3. Preliminaries. This section describes basic techniques used in the paper. All our simulations are based on universal hashing, i.e., the shared memory cells are distributed among the memory modules using three or more hash functions, randomly drawn from a universal class of hash functions. The analysis of the simulations requires high performance universal classes; i.e., if a function is chosen at random from the class, then its behavior will be close to that obtained by choosing a function at random from the class of all functions.

Since our simulations have $o(\log\log n)$ delay, we cannot afford to use deterministic and exact solutions for fundamental parallel problems, like sorting or prefix sums computation. Therefore, we use instead relaxed counterparts of these problems that can be solved by $\mathcal{O}(\log^* n)$-time randomized algorithms.

Section 3.1 briefly introduces the concept of hashing and universal hashing, section 3.2 states two tail estimates for dependent random variables, and section 3.3 presents results about fast randomized algorithms used in the paper.

3.1. Universal hashing. Our simulations require that the shared memory U of the PRAM is distributed among the memory modules of the DMM. For this we use hash functions that have properties similar to random functions. We use notation from Dietzfelbinger et al. [6], generalizing notions from Carter and Wegman [5].

For an integer k, define $[k] = \{0, \dots, k-1\}$. Let $U = [m]$, where $m > n$.

DEFINITION 1 (see [6]). *A family $\mathcal{H}_{m,n}$ of hash functions mapping U into $[n]$ is (μ, k)-universal, if for each $u_1 < \cdots < u_k \in U$, $l_1, \dots, l_k \in [n]$, and the hash function h drawn with uniform probability from $\mathcal{H}_{m,n}$, we have*

$$\Pr(h(u_1) = l_1, \dots, h(u_k) = l_k) \le \frac{\mu}{n^k} \ .$$

Such classes have been called sometimes μ strongly k universal or $((k)_\mu)$-independent. Notice that if $\mathcal{H}_{m,n}$ is (μ, k)-universal, then it is also (ν, j)-universal for all $\nu \ge \mu$ and $j \le k$.

For our purposes we require a $(2, \log^2 n)$-universal class of hash functions $\mathcal{H}_{m,n}$, such that a random $h \in \mathcal{H}_{m,n}$ can be constructed quickly, stored using little space ($\mathcal{O}(\sqrt{n})$ suffices), and evaluated in constant time. For example, we can use an $(1, n^\varepsilon)$-universal class of hash functions (where ε is a small positive constant) described by Siegel [29] for the case $n = m$, with the extension to $(2, n^\varepsilon)$-universal class for arbitrarily large $m > n$ from [19]. For a detailed description of this class see [19].

3.2. Tail estimates. In this paper we mainly deal with dependent random variables. To bound the deviation of the sum of dependent random variables from the expected value, we use two different tail estimates. The first one is also called the *method of bounded differences*, given in this form by McDiarmid [23].

LEMMA 3.1 (the method of bounded differences). *Let $x_1, \dots, x_n$ be independent random variables, where x_i takes values from a finite set A_i, for $i = 1, \dots, n$. Suppose that the function $f : \Pi_{i=1}^n A_i \to \mathbb{R}$ satisfies $|f(\bar{x}) - f(\bar{x}')| \le c_i$ whenever the vectors $\bar{x}$ and $\bar{x}'$ differ only in the ith coordinate. Let Y be the random variable $f(x_1, \dots, x_n)$. Then, for any $t > 0$,*

$$\Pr(Y \ge E(Y) + t) \le \exp\left(\frac{-2t^2}{\sum_{i=1}^n c_i^2}\right) \ .$$

Another tail estimate is a well-known generalization of the Markov inequality.

LEMMA 3.2 (the kth moment inequality). *Let Y be a random variable and $s > 0$. Then, for each $\alpha > 0$,*

$$\Pr(|Y| \ge \alpha) \le \frac{E(|Y|^s)}{\alpha^s} \ .$$

3.3. Log-star algorithms. In this section we outline main algorithmic tools used by our algorithms. Some of them are subsumed by others. Nevertheless, we state all of them to make it easier to refer to specific tasks. If an array of size $2 \cdot n$ contains at least n objects, we will call the array *padded-consecutive.*

DEFINITION 2. *The* strong semisorting *problem is the following: Given n integers $x_1, x_2, \ldots, x_n$, $x_i \in [n]$, store them in a padded-consecutive array, such that all variables with the same value occur in a padded-consecutive subarray.*

DEFINITION 3. *The* all nearest one *problem is the following: Given n bits $x_1, x_2, \ldots, x_n$, find for each bit x_i the nearest* 1*'s both to its left and to its right.*

DEFINITION 4. *The* approximate prefix sums *problem is the following: Given a sequence of nonnegative integers, $x_1, \ldots, x_n$, find a sequence $y_0 = 0$, $y_1, \ldots, y_n$, such that for $i \in [n]$, $y_i - y_{i-1} \geq x_i$, and*

$$\frac{1}{2}\sum_{j=1}^{i} x_j \leq y_i \leq 2\sum_{j=1}^{i} x_j.$$

The following lemma is a combination of several results.

LEMMA 3.3. *The following problems can be solved on the n-processor DMM in $\mathcal{O}(\log^* n)$-time with exponentially high probability, i.e., with probability $1 - 2^{-n^{\varepsilon}}$ for a constant $\varepsilon > 0$:*

(1) *strong semisorting,*

(2) *all nearest one, and*

(3) *approximate prefix sums.*

Proof. First, since an n-processor DMM can be simulated with a constant delay on an n-processor CRCW PRAM with $\Theta(n)$ shared memory, it is enough to prove the lemma for the n-processor CRCW PRAM that uses only $\mathcal{O}(n)$ space.

(1) Bast and Hagerup [3] show how to solve strong semisorting in $\mathcal{O}(\log^* n)$-time with the desired probability on a CRCW PRAM, provided the input is from $[n]$. For a general input, $\mathcal{O}(\log^* n)$-time perfect hashing [2, 10] reduces the problem to the solution of Bast and Hagerup. (See also [14, p. 275].)

(2) The all nearest one problem can be solved in $\mathcal{O}(\alpha(n)) = \mathcal{O}(\log^* n)$ time deterministically on a CRCW PRAM by an algorithm due to Ragde [26], and Berkman and Vishkin [4].

(3) Approximate prefix sums can be solved within the desired bound using an algorithm of Goodrich, Matias, and Vishkin [13]. □

4. The access graph. In this section we show how PRAM simulation can be modeled as the access game on the access graph. We also discuss basic properties of the access graph.

4.1. Definition and properties of the access graph. We start with the simulation of an EREW PRAM. The memory of the PRAM is hashed using three hash functions h_1, h_2, and h_3. That means each memory cell $u \in U$ of the PRAM is stored in the modules $M_{h_1(u)}, M_{h_2(u)}$, and $M_{h_3(u)}$ of the DMM. We call the representations of u in the $M_{h_i(u)}$'s the *copies* of u. We assume that all the hash functions used are drawn uniformly at random from a $(2, \log^2 n)$-universal class of hash functions $\mathcal{H}_{m,n}$ (see section 3.1). As mentioned in section 2, it is enough to analyze only protocols for the "1 out of 2" task, and therefore in our description we will use only two hash functions h_1 and h_2.

For technical reasons, we do not perform all n accesses to the shared memory simultaneously but split the requests into batches of size $n/2^{2c+6}$ for some constant $c \geq 1$ to be specified later. Since we only have a constant number of batches, this will slow down our algorithm only by a constant factor.

Let S denote such a batch. Let $G = ([n], E)$ be the labeled directed graph defined by two hash functions h_1, h_2 from $\mathcal{H}_{m,n}$ and the set of requests S. G has an edge $(h_1(u), h_2(u))$ labeled u for each $u \in S$. Note that parallel edges and self-loops are allowed in G, however all labels are distinct.

DEFINITION 5. *The* access graph H *is the labeled graph obtained from* G *by removing all directions from the edges. It consists of* n *nodes and* $n/2^{2c+6}$ *edges for some constant* $c \geq 1$.

The algorithms we present rely on the properties of the access graph. The following lemma is an extension of a result of Karp, Luby, and Meyer auf der Heide [19]. Define the size of a connected component C, denoted by $|C|$, to be the number of nodes it contains.

LEMMA 4.1. *For arbitrary positive constants* l *and* c *and for sufficiently large* n

(a) $\Pr$*(*H *has a connected component of size at least* $\frac{l}{c}\log n$*)* $\leq n^{-l}$*, and*

(b) *there is a constant* $w \geq 1$ *such that*

$$\Pr(H \textit{ has a connected component } C \textit{ with at least } |C| + w - 1 \textit{ edges}) \leq n^{-l}.$$

Proof. For the proof of the lemma it suffices that h_1 and h_2 are chosen uniformly at random from any $(2, \log^2 n)$-universal class of hash functions. The proof of the lemma relies on the following claim.

CLAIM 4.2. *Let* $k \geq 2, w \geq 0, k + w - 1 \leq \log^2 n$*. The probability that there is a connected subgraph* $G' \subseteq G$ *such that* G' *contains* k *vertices and at least* $k + w - 1$ *edges is at most*

$$n^{-w+1}k^{w-2}2^{-2c(k+w-1)} \quad .$$

Proof. Let $\mathcal{G}_{k,w}$ be the set of all directed connected graphs on node set $[n]$ with k vertices and $k + w - 1$ edges labeled by elements of S. Since any connected subgraph $G' \subseteq G$ with k vertices and at least $k + w - 1$ edges must contain a subgraph from $\mathcal{G}_{k,w}$, it is sufficient to show that G contains a subgraph from $\mathcal{G}_{k,w}$ with probability at most $n^{-w+1}k^{w-2}2^{-2c(k+w-1)}$.

An element of $\mathcal{G}_{k,w}$ can be generated by first choosing k vertices, then taking an undirected tree on these vertices and orienting the tree edges, then adding further w directed edges on the vertices of the tree, and finally assigning the labels to the chosen $k + w - 1$ edges. Therefore,[1]

$$\begin{aligned} |\mathcal{G}_{k,w}| &\leq \binom{n}{k} k^{k-2} 2^{k-1} k^{2w} 2^w \left(\frac{n}{2^{2c+6}}\right)^{k+w-1} \\ &\leq n^{2k+w-1} k^{2w-2} \left(\frac{e^2}{2^{2c+6}}\right)^{k+w-1} \quad . \end{aligned}$$

For fixed $G' \in \mathcal{G}_{k,w}$ and randomly chosen h_1 and h_2, G' is a subgraph of G only if the directions and labels with respect to h_1 and h_2 coincide with G'. Since $k +$

[1]Throughout the paper the inequalities $k^k/k! < e^k$ and $(k/l)^l \leq \binom{k}{l} \leq (ek/l)^l$ will be used without further comment.

$w-1 \le \log^2 n$ and h_1, h_2 are independently chosen from a $(2, \log^2 n)$-universal class, the probability that G' is a subgraph of G is at most $\left(2/n^{k+w-1}\right)^2$. Therefore, the probability that there is some $G' \in \mathcal{G}_{k,w}$ that appears as a subgraph of G is at most

$$n^{-w+1}k^{2w-2}\left(\frac{4e^2}{2^{2c+6}}\right)^{k+w-1} \le n^{-w+1}k^{2w-2}2^{-2c(k+w-1)} \ . \qquad \square$$

To prove part (a) of the lemma we use Claim 4.2 with $w=0$ and $k=\frac{l}{c}\log n$. This yields an upper bound on the probability of the existence of a connected component of size at least $k=\frac{l}{c}\log n$.

To prove part (b) notice first that from part (a) we obtain that H has a connected component of size at least $\frac{l}{2c}\log n$ with probability at most $n^{-l/2}$. Observe also that for any constant s, H contains a vertex with at least s self-loops with probability at most n^{1-s}. Hence we may use Claim 4.2 to obtain the upper bound for the probability that H has a connected component C with at least $|C|+w-1$ edges:

$$\begin{aligned} n^{-l/2}+n^{1-w}+\sum_{k=2}^{\frac{l}{2c}\log n} n^{-w+1}k^{w-2}2^{-2c(k+w-1)} &\le n^{-l/2}+n^{1-w}+\sum_{k=2}^{\frac{l}{2c}\log n}(k/n)^{w-1} \\ &\le n^{-l/2}+n^{1-w}+n^{2-w} \ . \end{aligned}$$

Thus if we set $w=\lceil 2+l/2\rceil$, then the probability is at most n^{-l}. $\square$

LEMMA 4.3. *Let l, b, and c be any positive constants with $c-1\ge b>2$, $l>2$, and let $1<k\le\frac{1}{2b}\log n$. Then, for n large enough,*

$$\Pr\left(H \textit{ has at least } \frac{n}{2^{bk}} \textit{ connected components of size at least } k\right) \le n^{-(l-1)} \ .$$

Proof. We first consider connected components of H of size exactly k, and then we extend our analysis to connected components of size at least k.

Let $\mathcal{W}_k$ be the set of all subsets of size k of the set of n nodes. With each $W\in\mathcal{W}_k$ we associate a binary random variable $\mathcal{I}_W$, such that $\mathcal{I}_W=1$ if the nodes from the set W form a connected component in H. Otherwise $\mathcal{I}_W=0$.

We want to bound the random variable

$$X_{(k)}=\sum_{W\in\mathcal{W}_k}\mathcal{I}_W,$$

which is the number of connected components of size k. As the random variables $\mathcal{I}_{W_1}$ and $\mathcal{I}_{W_2}$ are not independent for $W_1, W_2\in\mathcal{W}_k$, we use Lemma 3.2 to bound $X_{(k)}$. We compute the sth moment of $X_{(k)}$ with the following formula:

$$E(X_{(k)}^s)=E\left[\left(\sum_{W\in\mathcal{W}_k}\mathcal{I}_W\right)^s\right]=\sum_{(W_1,\ldots,W_s)\in(\mathcal{W}_k)^s}E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_s}) \ .$$

Consider a fixed tuple $(W_1,\ldots,W_s)$. If there exists a pair of sets $W_i, W_j\in\{W_1,\ldots,W_s\}$ that are not disjoint and not equal, i.e., $W_i\ne W_j$ and $W_i\cap W_j\ne\emptyset$, then $E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_s})=0$. This follows from the definition of the binary random variables because the two sets cannot both form a connected component of size exactly k. Hence, we only have to deal with the case that z sets, $1\le z\le s$, are disjoint and the other $s-z$ sets are equal to one of them. If we fix z, there are $\binom{s}{z}$ ways to choose the z sets that are disjoint and at most z^{s-z} ways to assign the remaining sets to one of them. Hence we can bound the sth moment in the following way:

$$E(X^s_{(k)}) \le \sum_{z=1}^{s} \binom{s}{z} z^{s-z} \sum_{\substack{(W_1,\ldots,W_z)\in(\mathcal{W}_k)^z \\ W_1,\ldots,W_z \text{ disjoint}}} E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \ .$$

Fix z disjoint sets $W_1,\ldots,W_z$ and assume that $s(k-1) \le \log^2 n$. For every i, $1 \le i \le z$, let $\mathcal{T}_i$ be the set of all directed spanning trees on nodes of W_i with edges labeled by the elements of S. Notice that

$$E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \le \sum_{T_1\in\mathcal{T}_1,\ldots,T_z\in\mathcal{T}_z} \Pr(\text{each of } T_1,\ldots,T_z \text{ is a subgraph of } H) \ .$$

Using arguments similar to those used in the proof of Claim 4.2, and by observing that the endpoints of the edges are chosen by hash functions from a $(2,\log^2 n)$-universal class, we obtain

$$E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \le \left(k^{k-2}\cdot 2^{k-1}\cdot\left(\frac{n}{2^{2c+6}}\right)^{k-1}\right)^z \cdot \left(\frac{2}{n^{k-1}}\right)^{2z} \ .$$

Now we can bound $E(X^s_{(k)})$:

$$\begin{aligned}
E(X^s_{(k)}) &\le \sum_{z=1}^{s}\binom{s}{z} z^{s-z}\binom{n}{k,k,\ldots,k,(n-zk)} k^{z(k-2)}\left(\frac{n}{2^{2c+5}}\right)^{z(k-1)}\left(\frac{4}{n^2}\right)^{z(k-1)} \\
&\le \sum_{z=1}^{s}\left(\frac{se}{z}\right)^z z^{s-z}\left(\frac{ne}{k}\right)^{zk} k^{z(k-2)}\left(\frac{1}{n2^{2c+3}}\right)^{z(k-1)} \\
&\le \sum_{z=1}^{s}\left(\frac{s}{z}\right)^z z^{s-z} n^z\left(\frac{e}{k^2}\cdot\frac{e^k}{2^{(2c+3)(k-1)}}\right)^z \\
&\le \sum_{z=1}^{s}\left(\frac{sn}{z^2 2^{ck}}\right)^z z^s \ .
\end{aligned}$$

Finally, Lemma 3.2 implies

$$\begin{aligned}
\Pr\left(X_{(k)} \ge \frac{n}{2^{bk+1}}\right) &\le \frac{E(X^s_{(k)})}{\left(\frac{n}{2^{bk+1}}\right)^s} \\
&\le \sum_{z=1}^{s}\left(\frac{sn}{z^2 2^{ck}}\right)^z\cdot\left(\frac{z2^{bk+1}}{n}\right)^s \\
&= 2^s\cdot\sum_{z=1}^{s}\left(\frac{s}{z2^{(c-b)k}}\right)^z\cdot\left(\frac{z2^{bk}}{n}\right)^{s-z} \\
&\le 2^s\cdot\sum_{z=1}^{s}\left(\frac{s}{z2^k}\right)^z\cdot\left(\frac{z}{\sqrt{n}}\right)^{s-z} \\
&= 2^s\cdot\sum_{z=1}^{s}\left(\frac{s\sqrt{n}}{z^2 2^k}\right)^z\cdot\left(\frac{z}{\sqrt{n}}\right)^{s} \\
&\le 2^s\cdot s\cdot\left(\frac{1}{2}\right)^{ks} .
\end{aligned}$$

In the fourth inequality we need $c \geq b+1$ and $k \leq \frac{1}{2b}\log n$. By setting $s = (l+2)\log n/(k-1)$ we obtain

$$\Pr\left(X_{(k)} \geq \frac{n}{2^{bk+1}}\right) \leq n^{-(l+1)} \ .$$

Now we extend our analysis to connected components of size at least k. For this we have only to observe

$$\Pr\left(\sum_{t \geq k} X_{(t)} \geq \frac{n}{2^{bk}}\right) \leq \Pr\left(\sum_{t\geq k} X_{(t)} \geq \sum_{t=k}^{n} \frac{n}{2^{bt+1}}\right)$$
$$\leq \sum_{t \geq k} \Pr\left(X_{(t)} \geq \frac{n}{2^{bt+1}}\right) \leq n^{-l} \ . \qquad \square$$

Lemma 4.1 and Lemma 4.3 are the basis of the proof of the following lemma which is essential for the analysis of our simulations.

LEMMA 4.4. *Let H be the access graph with n nodes and $\frac{n}{2^{2c+6}}$ edges for some constant $c \geq 1$ defined as in Lemma 4.3. For all constants β and l such that $c \geq 2l(\beta+2)$, with probability at least $1 - 2(\frac{1}{n})^{l-1}$,*

$$\sum_{\substack{\text{connected components } C \\ |C|>1}} |C| \cdot 2^{|C|\,\beta} \leq n \ .$$

Proof. Let $b = \beta + 2$. We split the sum into two parts:

$$\sum_{1<|C|\leq \frac{1}{2b}\log n} |C| \cdot 2^{|C|\,\beta} + \sum_{\frac{1}{2b}\log n < |C|} |C| \cdot 2^{|C|\,\beta} \ .$$

Observe that because $c \geq 2bl$, we get $\frac{1}{2b}\log n \geq \frac{l}{c}\log n$. Therefore, the right-hand part is zero w.h.p. by part (a) of Lemma 4.1. Hence, by Lemma 4.3, with probability at least $1 - 2\left(\frac{1}{n}\right)^{l-1}$, the above sum is bounded by

$$\sum_{k=2}^{\frac{1}{2b}\log n} \frac{n}{2^{bk}} k 2^{\beta k} \leq n \sum_{k=2}^{\frac{1}{2b}\log n} \left(\frac{1}{2^{b-\beta-1}}\right)^{k} \leq n \ .$$

The last inequality holds as $b = \beta + 2$. $\square$

4.2. The access game. In the following we view a PRAM simulation, or more precisely a protocol for the "1 out of 2" task, as the following process on the access graph H, which we call the *access game* on H. Each processor that wants to access a shared memory cell $u \in U$ asks in each step either $M_{h_1(u)}$ or $M_{h_2(u)}$. Consequently, if a module M_j answers the request for cell u, then the edge labeled u is removed from H. That is, every node in H removes one incident edge (if any). The simulation ends when all the edges from H are removed.

Note that initially all the processors are assigned to the edges in H and that the nodes in H do not know the adjacent edges. We want to view the simulation from the point of view of the nodes; i.e., a node removes an incident edge. For this we assign to each node a processor that computes which of the incoming edges will be removed. Then it sends a message to the processor assigned to this edge to send a request to the respective module.

Remark 1. Lemma 4.1 immediately implies the $\mathcal{O}(\log\log n)$ implementation of the "1 out of 2" task due to Karp, Luby, and Meyer auf de Heide [19]. In each round, each module removes an arbitrary incident edge. Lemma 4.1 ensures that the connected components have at most $\mathcal{O}(\log n)$ edges. Thus, because in each round the number of edges in each connected component is at least halved, $\mathcal{O}(\log\log n)$ rounds suffice to remove all edges.

5. A simulation with delay $O(\log\log n/\log\log\log n)$.

Throughout this section, H will denote the access graph, which is assumed to satisfy the conditions of Lemma 4.4. After the comments in section 4.2, the main challenge is how to remove the edges from the access graph. The first result is a randomized simulation of one step of an EREW PRAM with delay $\mathcal{O}(\log\log n/\log\log\log n)$, w.h.p. Its basic routine works with two hash functions and solves the "1 out of 2" task. The high level description of the protocol in terms of the processors and the modules is as follows. In each step, each module chooses among its incoming requests the one with the highest contention, i.e., for which the memory module storing the other copy gets most requests. For implementing the access protocol we use log-star techniques described in section 3.3 to get an $\mathcal{O}(\log^* n)$-time preprocessing algorithm for computing the number of requests directed to each module.

5.1. The neighbor-data-structure.

We introduce a data structure called *neighbor-data-structure*. The neighbor-data-structure supports the following two operations on the access graph: (i) remove a set of edges and (ii) assign to each nonisolated node its neighbor with the maximum degree.

The preprocessing phase generates an array A of length $4n$. It contains, for each node v, a subarray A_v of length $\tilde{d}(v)$, $d(v) \le \tilde{d}(v) \le 2d(v)$. A_v contains the $d(v)$ edges incident to v and gaps in the remaining positions. Further, an array B of length n contains a pointer to the leftmost cell of A_v for each node v. In addition, the preprocessing phase assigns $\tilde{d}(v)2^{\tilde{d}(v)}$ processors to each nonisolated node v. By Lemma 4.4, at most n processors are needed for this assignment.

For a subgraph H' of H, the neighbor-data-structure is derived from the one for H by simply removing the missing edges from the subarrays A_v. The assignment of processors and the lengths of the A_v's remain unchanged.

NDS PREPROCESSING.

- *Represent each (undirected) edge (i,j) by two ordered pairs $[i,j]$ and $[j,i]$.*
- *Perform strong semisorting with respect to the first coordinate.*
- *Compute for each node v the size $\tilde{d}(v)$ of the subarray A_v containing all edges adjacent to v.*
- *Allocate $\tilde{d}(v)\,2^{\tilde{d}(v)}$ processors to each node v.*

After strong semisorting, the second step of the preprocessing, all edges (the corresponding ordered pairs) adjacent to a node v of degree $d(v)$ appear in a subarray A_v of size $\tilde{d}(v) = \mathcal{O}(d(v))$. The algorithm for the all nearest one problem can be used to find the first and the last edge of each node and therefore to compute the size $\tilde{d}(v)$ of the subarray A_v. The following lemma bounds the time needed for the preprocessing.

LEMMA 5.1. NDS PREPROCESSING *builds up the neighbor-data-structure and can be performed in $\mathcal{O}(\log^* n)$ time on an n-processor DMM, w.h.p.*

Proof. The time bound follows from Lemma 3.3. Note that $d(v) \le \tilde{d}(v) \le 2d(v)$. Hence, by Lemma 4.4, the total size of the neighbor-data-structure is n. In addition, since the degree of v is bounded by the size of its connected component, w.h.p., the total number of allocated processors to each connected component is at most $2\,|C|^2\,2^{2\,|C|}$. Therefore, Lemma 4.4 ensures that we allocate only a linear number of processors, w.h.p. ∎

The following lemma shows how to dynamically manipulate the neighbor-data-structure.

LEMMA 5.2. *Once the neighbor-data-structure is built, one can update the neighbor-data-structure after deletion of any set of edges in constant time on an n-processor DMM.*

Proof. Since each edge e knows its positions in the arrays A_v, one can remove any edge by changing the corresponding positions into gaps. ∎

The next lemma describes the functionality of the neighbor-data-structure.

LEMMA 5.3. *Given the neighbor-data-structure for some subgraph of H, on an n-processor DMM, it is possible in constant time to assign to each nonisolated node a neighbor of maximum degree.*

Proof. Let us look at the array A_v as an array containing marked objects (corresponding to edges adjacent to v) and unmarked objects (corresponding to gaps). To compute the degree of v it is sufficient to compute the number of marked objects in A_v. Since there are only $2^{\tilde{d}(v)}$ combinations of marked and unmarked objects in A_v, one can compute this number in constant time, using $\tilde{d}(v)$ processors for each combination. Note that this amount of processors is available for each node because we allocated an exponential number of processors to each node. Now each node v can get the degrees of all its neighbors in constant time with $\tilde{d}(v)$ processors. To find a node with the maximum degree one can use the standard maximum finding algorithm that runs (deterministically) in constant time with $(\tilde{d}(v))^2$ processors [16, p. 72]. ∎

5.2. Schedule and analysis of the simulation. In section 4 we have shown that if every node always removes an arbitrary adjacent edge, then a simulation with delay $\Theta(\log\log n)$ results. We want to describe a more efficient protocol, SIMULATION 1, using the neighbor-data-structure.

SIMULATION 1.

```
• NDS PREPROCESSING: Build up the neighbor-data-structure.
• Repeat until no more edges are left:
    – In parallel, each node removes the edge
      pointing to a neighbor with highest degree.
    – Update the neighbor-data-structure.
```

We bound the number of iterations of the algorithm.

LEMMA 5.4. SIMULATION 1 *is finished after $\mathcal{O}(\frac{\log\log n}{\log\log\log n})$ iterations, w.h.p.*

Proof. We say an access graph H *survives* k iterations of SIMULATION 1, if at least one edge is left after performing k iterations. We want to construct structures that are embeddable in the access graph H, if it survives k iterations of SIMULATION 1. Let $k \geq 1$. A *k-fork* consists of k different nodes (called *leaves*) connected to one node (called the *center*) that is connected to another node called the *parent node* of the k-fork. A *k-witness* is defined recursively as follows. A 1-witness is a path of length 5 (i.e., with 6 nodes); the two nodes of degree one are the *leaves*. For $k > 1$, a k-witness is a $(k-1)$-witness each leaf of which is a parent node of a different k-fork. All leaves of the k-forks are called the leaves of the k-witness. Note that a k-witness has $2k!$ leaves. An example of a 4-witness is given in Figure 1. The proof of the lemma is based on the following claim.

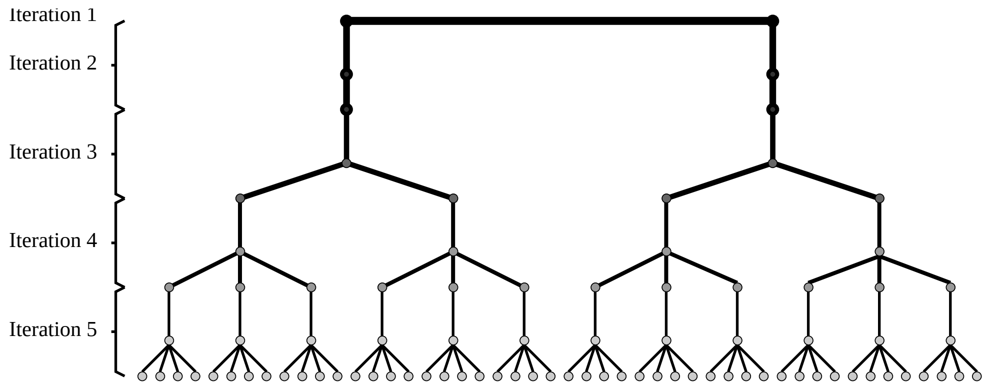


FIG. 1. *A 4-witness.*

CLAIM 5.5. *If H survives k iterations of* SIMULATION 1*, then there is an embedding of a k-witness W in H that maps edges that have a node in common to different edges of H.*

Proof. The proof of the claim is by induction on k. If H survives one iteration, then a path of length 5 must be embeddable in H. Assume now that the claim holds for k and let H survive $k+1$ iterations.

Perform the first iteration on the graph H. A subgraph $\tilde{H} \subseteq H$ which survives k iterations will be left. From the induction hypothesis, we know that a k-witness $\tilde{W}$ can be embedded in $\tilde{H}$ in the way described in the claim. For this structure to survive the last iteration we extend it to a witness-structure W that must be embeddable in H. Since in the first iteration no edge from $\tilde{H}$ has been removed, each node from $\tilde{H}$ must have removed an incident edge in $H - \tilde{H}$. As $\tilde{W}$ is a k-witness, it consists of a $(k-1)$-witness each leaf of which is a parent node of a k-fork. Because each center of a k-fork removes an edge from $H - \tilde{H}$ that points to a neighbor with the maximum degree, its degree in H is at least $k+2$. Thus every leaf v in $\tilde{W}$ must have a neighbor u (different than the center) of degree at least $k+2$. Hence v must be the parent node of a $(k+1)$-fork that consists of v, u and $k+1$ neighbors of u other than v and the center of the k-fork. This defines the required embedding of W in H. ◻

Assume that SIMULATION 1 needs more than k iterations on the graph H. Then by Claim 5.5 a k-witness W is embedded in a connected component C of H.

By Lemma 4.1, $|C| = \mathcal{O}(\log n)$, and C consists of a tree and a constant number ζ of additional edges, w.h.p. In the worst case, at each node of C, at most ζ edges from W can be embedded into these ζ edges of C. Therefore, there must exist a $(k-\zeta)$-witness $\tilde{W}$ that can be embedded one-to-one in C. As $|\tilde{W}| \geq 2(k-\zeta)!$, by the definition of a $(k-\zeta)$-witness, the embedding of W in C as described in Claim 5.5 uses $\Omega((k-\zeta)!)$ nodes of C. Since $|C| = \mathcal{O}(\log n)$, we can embed only a k-witness W for $k = \mathcal{O}(\frac{\log\log n}{\log\log\log n})$. ◻

This lemma together with Lemmas 5.1, 5.2, and 5.3 implies the following theorem.

THEOREM 5.6. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM with delay $\mathcal{O}(\log\log n/\log\log\log n)$, w.h.p.*

6. Faster simulations. In this section we extend the ideas from the previous section. We present an algorithm that not only computes information on the 2-neighborhood (degrees of neighbors) but explores a larger neighborhood of each node. Then, a second algorithm is presented that removes all edges from the access graph in constant time, assuming each node knows its whole connected component. As a first consequence we give a simple protocol that significantly improves the delay of the simulations presented before. Finally, we present the $\mathcal{O}(\log\log\log n \log^* n)$ delay simulation.

6.1. The path-access-structure. In this subsection we develop some algorithmic utilities for almost random graphs, especially methods for exploring a large neighborhood of a node, which is essential for our algorithms. Throughout this section, H will denote the access graph which fulfills the conditions of Lemma 4.4. The *k-neighborhood* of a node v is the subgraph of H containing the nodes and the edges that are reachable from v by a path of length at most k.

Our simulations use a data structure, called *path-access-structure* (PAS), that allows fast access to all simple paths of length at most k in H. We prove a simple lemma to bound the total length of all simple paths in H.

LEMMA 6.1. *Let C be a connected component that is a tree with a constant number ζ of additional edges and v be a node in C. Then the total number of simple paths starting at v is $\mathcal{O}(|C|)$.*

Proof. The total number of simple paths that do not use the additional edges is $|C|$. The other simple paths can use each additional edge only once. Therefore, the number of simple paths using additional edges can be bounded by $\zeta!2^{\zeta+1}|C| = \mathcal{O}(|C|)$. ∎

Note that, by Lemma 4.1, each connected component C of H has the properties stated in Lemma 6.1, w.h.p. As each simple path in C has length at most $|C|$, we can bound the total length of all simple paths in C by $\mathcal{O}(|C|^3)$, w.h.p. Hence, by the statement made in Lemma 4.4 about the distribution of the sizes of these components, the total length of all the simple paths is $\mathcal{O}(n)$, w.h.p.

The *path-access-structure for (simple paths of length at most) k* is a data structure that supports the following operations on H: (i) remove a set of edges, (ii) assign to each nonisolated node v the node with minimum identifier in the k-neighborhood of v, (iii) assign to each nonisolated node in H an incident edge that begins a simple path of length at least k (if there exists such an edge), and (iv) for all edges e and indices r, $1 \le r \le k$, give the number of simple paths of length exactly r that start with edge e.

Now we describe the implementation details of the path-access-structure for k. We store the simple paths of length at most k in an array S of length $\mathcal{O}(n)$, which consists of padded-consecutive subarrays S_v, one for each node v of H. Each S_v contains a representation of all paths starting in v, each one in consecutive cells. In order to access the paths we build up an array P which consists of padded-consecutive subarrays P_v, one for each node v of H. P_v contains a pointer to the header of each path starting at node v together with the length of this path.

We first give a high level description of how to apply the doubling technique to build up the path-access-structure for simple paths of length at most k.

PAS PREPROCESSING.

```
All nodes are active.
Initialize all the $P_v$ and $S_v$ as empty.
For $r = 0$ to $\log k$ perform the following iteration for all active nodes $v$ in parallel:
    • Using $P_v$, compute an approximate (within a factor of 2) number $\tilde{\delta}_r(v)$ of
      simple paths of length at most $2^r$ starting at $v$.
    • Using $S_v$, compute an approximate (within a factor of 2) total length $\tilde{\gamma}_r(v)$
      of all simple paths starting at $v$.
```

- *Update the sizes of S_v and P_v using $\tilde{\delta}_r$ and $\tilde{\gamma}_r$.*
- *Using the doubling technique, find all simple paths of length l, $2^r < l \leq 2^{r+1}$, starting at v and store them in S_v and P_v.*
- *Inactivate all nodes that are not beginnings of a simple path of length 2^{r+1}.*

LEMMA 6.2. *In time $\mathcal{O}(\log k \log^* n)$, with linear total work, the path-access-structure for all simple paths in H of length at most k can be built on an n-processor DMM, w.h.p., using the procedure* PAS PREPROCESSING.

Proof. The algorithm is based on the standard doubling technique (see, e.g., [16]). We perform $\log k + 1$ iterations and ensure the following invariant after r iterations, $0 \leq r \leq \log k$: Each node v has already found all simple paths of length at most 2^r that start at v, stored them in a padded-consecutive subarray S_v and stored the pointers to each path together with its length in a padded-consecutive subarray P_v. If a given node has already found all its simple paths, then it is *inactive*. Otherwise it is *active*. Note that in each iteration of the algorithm only active nodes participate.

When $r = 0$, then all the paths of length 1 are exactly the edges incident to v. Thus, this 0th iteration of PAS PREPROCESSING is covered by the construction of the neighbor-data-structure. The NDS PREPROCESSING and Lemma 5.1 can be used to build up the neighbor-data-structure. Within $\mathcal{O}(\log^* n)$ time and linear work, one can move all the edges incident to v to a padded-consecutive subarray at the beginning of A_v, w.h.p. (compare section 3.3). This creates the adjacency list of v. Thus, we can create the arrays S_v and P_v. Additionally, we inactivate all isolated nodes.

We perform the $(r+1)$st iteration, for $r \geq 0$, using the doubling technique. Let v be any active node and C_v be its connected component. Note that since v is active, we have $|C_v| \geq 2^r$. First, v computes how many simple paths of length at most 2^r start at v. Because it is hard to compute this value exactly, each node v computes a value $\tilde{\delta}_r(v)$ which is not smaller and at most 2 times larger than the number of paths that start at v. Since the subarray P_v containing the pointers to all simple paths starting at v is padded-consecutive, simply finding the first and the last such paths makes it possible to compute $\tilde{\delta}_r(v)$ after performing the algorithm for the all nearest one problem. Now we compute an approximation $\tilde{\gamma}_r(v)$ of the total length of all simple paths stored at S_v. Since the lengths of all such paths are stored at P_v, we compute $\tilde{\gamma}_r(v)$ using the approximate prefix sums algorithm for all v. The difference between the last cumulative path-lengths in two consecutive subarrays P_v and $P_{v'}$ gives us $\tilde{\gamma}_r(v)$.

Let (x, y) be the last edge of any simple path p of length 2^r which starts at v. To find all paths of length l, $2^r < l \leq 2^{r+1}$, starting with p, we must combine p with all paths of length at most 2^r starting at y. Then we remove the anomalies, i.e., the paths that create cycles.

Observe that, using the values $\tilde{\delta}_r(y)$ and $\tilde{\gamma}_r(y)$, we know how big the new arrays P_v and S_v have to be (for each path p, P_v has to be extended by $\tilde{\delta}_r(y)$ and S_v by $2^r \cdot \tilde{\delta}_r(y) + \tilde{\gamma}_r(y)$). This space allocation can be done using global approximate prefix sums. Observe that $\tilde{\delta}_r(v) = \mathcal{O}(|C_v|)$ and $\tilde{\gamma}_r(v) = \mathcal{O}(|C_v| \cdot 2^r) = \mathcal{O}(|C_v|^2)$, w.h.p., using Lemma 6.1. Hence the size of the new P_v is $\mathcal{O}(|C_v|^2)$, and the size of the new S_v is $\mathcal{O}(|C_v|^3)$, w.h.p.

It is easy to compute the length of each newly created path to maintain P_v. To update S_v we have only to copy the old paths from S_v and concatenate the paths from v to y with simple paths from y. Hence these operations can be performed in constant time with total work proportional to the sizes of the new P_v and S_v. This means that the total work for all nodes is $\sum_{|C| \geq 2^r} \mathcal{O}(|C|^4)$, and the total running time in each iteration is $\mathcal{O}(\log^* n)$, since it is dominated by the cost of computing $\tilde{\gamma}_r(v)$, $\tilde{\delta}_r(y)$ and of allocating the subarrays P_v and S_v.

Finally, we have to remove those obtained paths that are not simple. We identify each path in S_v with the position of the first node. Then we perform strong semisorting within all arrays S_v with respect to the pairs [path, a node on the path]. Now, if there is more than one pair $[p, y]$, which can be verified by applying the all nearest one algorithm, then the path p is not simple and we eliminate it. Strong semisorting within all arrays P_v can be used to remove nonsimple paths from these arrays. Using the lengths of the paths in the P_v's, approximate prefix sums makes it possible to remove all nonsimple paths in the arrays S_v. Hence we can maintain the P_v's and S_v's to be padded-consecutive. Now we inactivate a node v if the new P_v contains no path of length 2^{r+1}.

The running time of iteration r is $\mathcal{O}(\log^* n)$ with $\mathcal{O}(\sum_{|C|\geq 2^r} |C|^4)$ total work. Therefore, the total work of the algorithm is

$$\mathcal{O}\left(\sum_{r=0}^{\log k} \sum_{|C|\geq 2^r} |C|^4\right) = \mathcal{O}\left(\sum_{C} |C|^4 \log |C|\right),$$

and by Lemma 4.4, this is $\mathcal{O}(n)$. ∎

Now we show how to maintain the path-access-structure dynamically, that is, when we allow removing edges from the graph. The proof of the following lemma is in the spirit of the proof of Lemma 6.2.

LEMMA 6.3. *Assume that the path-access-structure has already been built. Then, after removing some edges from the graph, it can be updated in time $\mathcal{O}(\log^* n)$ on an n-processor DMM, w.h.p.*

Proof. First we perform the preprocessing step and run strong semisorting on all arrays S_v with respect to the names of the edges. (Of course, it causes no problem to find and look at the edges instead of the nodes.) The result is an array $\mathcal{R}$ of linear size, such that for each edge e in a padded-consecutive subarray of $\mathcal{R}$ all the occurrences on all simple paths together with the pointers to these paths in P_v's are stored.

Now, when an edge e is removed, we can easily remove from the arrays P_v and S_v all simple paths on which e occurs. To remove all the edges, one needs constant time and work proportional to the size of $\mathcal{R}$. If one requires that all the subarrays P_v and S_v still have to be padded-consecutive, then additional $\mathcal{O}(\log^* n)$ time is needed. ∎

LEMMA 6.4. *Assume that the path-access-structure for k has been built. Then, to each nonisolated node one can assign the node with the minimum identifier in its k-neighborhood in $\mathcal{O}(\log^* n)$ time on an n-processor DMM.*

Proof. All simple paths that start at v and are of length at most k are stored in a padded-consecutive subarray S_v. Thus, this subarray contains exactly the nodes from the k-neighborhood of v. Now each node v can find the node w with the minimal identifier in its k-neighborhood in $\mathcal{O}(\log^* n)$ time. Each node v has $|S_v| = \mathcal{O}(|C_v|^2)$ candidates, where C_v denotes the connected component v belongs to, and using $|S_v|^2 = \mathcal{O}(|C_v|^4)$ processors it can find the node with minimum identifier in constant time [16, p. 72]. $\mathcal{O}(\log^* n)$ time is needed for assigning processors to the nodes by using approximate prefix sums. ∎

LEMMA 6.5. *Assume that the path-access-structure for k is given. One can assign to each node its incident edge that begins a simple path of length at least k in constant time on an n-processor DMM.*

Proof. For each node v the array P_v immediately gives such an edge. ◻

The next lemma shows how to use the path-access-structure for k to count the number of all simple paths of length at most k.

LEMMA 6.6. *Suppose that the path-access-structure for k is given. Then, for all edges e and integers r, $1 \leq r \leq k$, the number of simple paths of length exactly r that start with edge e can be computed in $\mathcal{O}(\log^* n)$ on an n-processor DMM, w.h.p.*

Proof. For each simple path in all S_v's we consider the pairs [starting edge of the path, length of the path]. We perform strong semisorting with respect to these keys. In this way, all the simple paths (in fact their representatives) that start with the same edge e and are of the same length r are stored in a padded-consecutive subarray, which we call $X_{e,r}$. If $y_{e,r}$ denotes the number of such paths, then $y_{e,r} \leq |X_{e,r}| \leq 2y_{e,r}$. We allocate $|X_{e,r}| \cdot 2^{|X_{e,r}|}$ processors to the pair $[e, r]$ and compute $y_{e,r}$ in the same way as in the proof of Lemma 5.1 in constant time. Now we have to show that we use only n processors. Let C_e be the connected component e belongs to. By Lemma 4.1, $y_{e,r} = \alpha|C_v|$ and each connected component C has at most $\beta|C|$ edges, for some constants α and β. Hence

$$\sum_{e\in E}\sum_{r=0}^{k}|X_{e,r}|\cdot 2^{|X_{e,r}|} \leq \sum_{C}\sum_{e\in C}|C|\cdot(2y_{e,r})2^{2y_{e,r}} \leq \sum_{|C|>1} 2\alpha\beta|C|^3 2^{2\alpha|C|} \leq \sum_{|C|>1}|C|2^{b|C|}$$

for some constant b. Lemma 4.4 ensures that this is bounded by n. ◻

A *k-branch* of a node v in the access graph H is the set of all different simple paths of length at most k that start with the same edge incident to v. Clearly, for every node v the number of its k-branches equals its degree. Define LEVEL(r) of a k-branch of a node v to be the set of all simple paths of length r, $0 < r \leq k$, of this k-branch. The *weight* of a k-branch of a node v is the bit-vector $\overline{w} = (w_1, \dots, w_k)$, where w_r is 1 if and only if the number of simple paths in LEVEL(r) of the branch is at least 2^{r-1}. We order the weights with respect to the lexicographical ordering. Informally, a k-branch is lexicographically larger than another k-branch, if it is more similar to a complete binary tree with respect to the number of nodes in each level.

LEMMA 6.7. *Suppose that the path-access-structure for k is given. Then one can assign to each node in H an incident edge that begins a k-branch with maximum weight in $\mathcal{O}(\log^* n)$ time on an n-processor DMM.*

Proof. Using Lemma 6.6, each node can get the weights of all incident k-branches in $\mathcal{O}(\log^* n)$ time. Let C_v denote the connected component containing node v. Each node has $\mathcal{O}(|C_v|)$ k-branches, each of depth $\mathcal{O}(|C_v|)$. Hence it can find a k-branch with the maximum weight in constant time using $|C_v|^4$ processors using the standard algorithm (see, e.g., [16, p. 72]). Hence all nodes together can find the required k-branches in constant time on a p-processor DMM with $p = \sum_v |C_v|^4 = \sum_{|C|>1} |C|^5$. By Lemma 4.4, $p \leq n$. ◻

6.2. Cleaning up connected components.

The $\mathcal{O}(\log\log n/\log\log\log n)$-time simulation from section 5 is based on very local information. Each node looks only at its neighbors in the access graph and, based on their degrees, chooses one incident edge. The main idea leading to improvements of that result is to analyze the access graph in a nonlocal way and to try to use information on as many nodes and edges as possible.

The notion of the k-neighborhood plays the crucial role. Instead of looking only at the neighbors, now each node v will base the decision which incident edge to remove on the structure of its k-neighborhood. We will explore the k-neighborhood of each node in H. As we have seen in Lemma 6.2, essentially all information on the k-neighborhood can be computed in time $\mathcal{O}(\log k \log^* n)$. In this subsection we will describe a process that removes all edges from a connected component with diameter δ in time $\mathcal{O}(\log \delta \log^* n)$.

Cleaning up Connected Components.

```
• PAS Preprocessing: Build up the path-access-structure.
• Perform for each node in parallel:
    – Each node $v$ obtains the node $w$ with the minimal identifier in its component.
    – If $v \neq w$, then $v$ finds all nodes $u_1, u_2, \ldots, u_r$, such that $(v, u_i)$ is the
      first edge of a simple path from $v$ to $w$.
    – $v$ removes the edges $(v, u_i)$, $1 \leq i \leq r$.
```

The access protocols described in previous papers (e.g., in [19], see also Remark 1 in section 4.2) show how to remove all edges of a connected component C of H in time $\mathcal{O}(\log(|C|))$. The following lemma proves that Cleaning up Connected Components achieves time $\mathcal{O}(\log(\text{diameter of } C))$. Note that this does not give fast simulations on its own, because H has a connected component of diameter $\Omega(\log n / \log \log n)$, with constant probability (see Lemma 10.2).

Lemma 6.8. *Let δ be the maximum diameter of the connected components of H. Then* Cleaning up Connected Components *removes all edges in H in time $\mathcal{O}(\log \delta \log^* n)$, w.h.p.*

Proof. As it is shown in Lemma 6.2, one can build up the path-access-structure for δ in time $\mathcal{O}(\log \delta \log^* n)$ on an n-processor DMM. Because for each node its δ-neighborhood is equal to its connected component, Lemma 6.4 allows us to find the nodes w in Cleaning up Connected Components in $\mathcal{O}(\log^* n)$ time.

By Lemma 4.1 we have $r = \mathcal{O}(1)$, w.h.p. Now we may use the array S from the path-access-structure to find for each node v the nodes $u_1, \ldots, u_r$ in $\mathcal{O}(1)$ time. Notice also that each edge e of H must be of the form (v, u_i) for some v and i. Therefore, at the end of the algorithm all the edges are removed. Since $r = \mathcal{O}(1)$ w.h.p., the last step of the algorithm takes $\mathcal{O}(1)$ time, w.h.p. ∎

6.3. An $O(\sqrt{\log\log n}\log^* n)$-delay simulation. The $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ simulation consists of three phases and may be described on a high level in the following way. Let $k = 2^{\sqrt{\log\log n}}$.

Simulation 2.

```
• PAS Preprocessing: Build up the path-access-structure for $k$.
• Repeat $2 \cdot \log\log n / \log k$ times:
    – Each node removes an incident edge which is the beginning of a simple
      path of length at least $k$, if there exists such an edge.
    – Update the path-access-structure.
• Cleaning up connected components.
```

Since the clean-up phase removes all edges from H, the correctness of the algorithm is obvious. We show that after performing the loop in the second phase of Simulation 2, we have partitioned H so that the diameter of each connected component is at most $2k$. Therefore, the clean-up phase can be performed in time $\mathcal{O}(\log k \log^* n)$ on an n-processor DMM, w.h.p., as shown in Lemma 6.8.

We say that a connected component C of H *survives* t iterations of Simulation 2 if at least one edge of C is not removed at the end of the tth iteration in the second phase. The idea of the proof is to bound the size of C.

LEMMA 6.9. *After the second phase of* SIMULATION 2*, the diameter of each connected component is at most* $2k$*, w.h.p.*

Proof. Consider a connected component C of H that survives t iterations and any maximal connected subgraph C_t of C of size larger than 1 that is left after t iterations. Define inductively a sequence $C_t, C_{t-1}, \dots, C_0$ such that for every i, $0 \le i < t$, C_i is a maximal connected subgraph of C that survives the ith iteration and contains C_{i+1}.

If the diameter of C_t is greater than $2k$, then every node v on C_t is the beginning of a simple path of length k. Hence, according to the elimination rule of the first phase of SIMULATION 2, every node has removed in this iteration an incident edge not belonging to C_t which was the beginning of a simple path of length at least k. (Note that if the diameter of C_t is greater than $2k$, then this also holds for C_{t-i}, $1 \le i \le t$.)

The $|C_t| \cdot k$ edges of these paths are edges of the access graph H. Since each path is simple, edges of one path are embedded injectively. From Lemma 4.1 we know that each connected component C consists of $|C| + \mathcal{O}(1)$ edges. Hence, it has only a constant number ζ of cycles. If we embed edges of two of these paths to the same edge in H, then the sum of the two paths contains a cycle in H. Therefore, at least $|C_t| - \zeta$ of these simple paths of length k are disjoint from each other in the embedding, that is, C_{t-1} has $k \cdot (|C_t| - \zeta) \ge \frac{k}{2} \cdot |C_t|$ edges.

This recursion shows: if a connected component C survives t iterations in the second phase of SIMULATION 2, and if a connected part of C left at the end of the tth iteration has diameter greater than $2k$, then C must be of size $\Omega(k^t/2^t)$. Now assume that this is the case for $t = 2 \cdot \log\log n / \log k$. Then C must be of size $\Omega(\log^2 n)$, which contradicts Lemma 4.1. ∎

From Lemmas 6.2, 6.3, and 6.5, it follows that it is possible for each node to analyze in the first phase its k-neighborhood in time $\mathcal{O}(\log k \log^* n)$ and each iteration in the second phase can be executed in time $\mathcal{O}(\log^* n)$, w.h.p. Therefore, if we set $k = 2^{\sqrt{\log\log n}}$ and apply the majority technique we obtain the following theorem.

THEOREM 6.10. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM with $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ delay, w.h.p.*

6.4. An $O(\log\log\log n \log^* n)$-delay simulation. In this subsection we describe a simulation of an n-processor EREW-PRAM on an n-processor DMM that improves all previously known simulations exponentially. Essentially, we show how to partition each connected component using information about the $(\log\log n)$-neighborhood so that the diameter of the largest connected component is $\mathcal{O}((\log\log n)^2)$. Then we can apply the CLEANING UP CONNECTED COMPONENTS procedure to remove the remaining edges in H. To achieve an improvement compared to SIMULATION 2, we use the knowledge about the k-neighborhood in a more efficient way.

SIMULATION 3.
- PAS PREPROCESSING*: Build up the path-access-structure.*
- *Each node v removes the incident edge which is the beginning of a k-branch with maximal weight.*
- CLEANING UP CONNECTED COMPONENTS.

Lemmas 6.2 and 6.7 make sure that the first two phases of SIMULATION 3 can be performed in time $\mathcal{O}(\log k \log^* n)$, w.h.p. Now we prove that at the beginning of the clean-up phase the maximum diameter of each connected component in H is at most $\mathcal{O}(k^2)$, w.h.p., for $k \ge \log\log n$.

LEMMA 6.11. *Let $k \ge \log\log n$, and ζ denote the number of cycles in the access graph H. After the second phase of* SIMULATION 3*, w.h.p., H does not contain any simple path of length more than $(\zeta + 1) \cdot (2k + 1)^2$.*

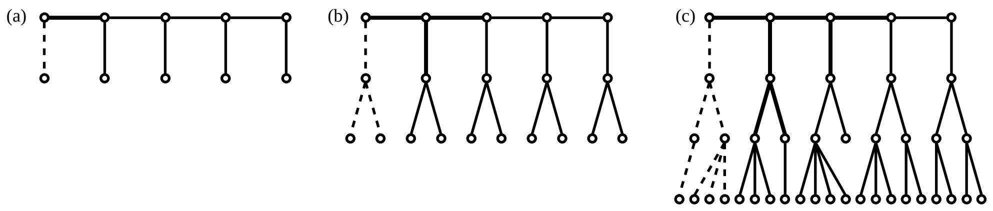


FIG. 2. (a) *the path $\tilde{p}$ with the edges that are removed by the nodes of $\tilde{p}$;* (b) *the path $\tilde{p}$ with the edges required by the elimination rule on* LEVEL(2)*;* (c) *the path $\tilde{p}$ with the edges required by the elimination rule on* LEVEL(3)*.*

Proof. Assume, to the contrary, that at the beginning of the clean-up phase a simple path p of length $(\zeta+1)\cdot(2k+1)^2$ survives. Let us call a branch that starts with an edge from p the *path-branch*, and let us call any k-branch chosen in the second phase of SIMULATION 3 by a node from p the *side-branch* of this node. Since no edge of the path p has been removed in the first step, for each node of p, the side-branch differs from the path-branch. Lemma 4.1 ensures that $\zeta = \mathcal{O}(1)$. Hence there exists a subpath $\tilde{p} = (v_0, v_1, \dots, v_{2k})$ of p, such that all vertices of the side-branches of nodes from $\tilde{p}$ are not contained in any cycle of length smaller than $2k$.

If the weight of a k-branch satisfies $w_1 = w_2 = \cdots = w_r = 1$, then we call it *$r$-complete*. We show that for each node from $\tilde{p}$ the side-branch must be r-complete; for all nodes v_i, $0 \le i \le 2k-r$, the right path-branch (starting from the edge (v_i, v_{i+1})) is r-complete; and for all nodes v_i, $r \le i \le 2k$, the left path-branch (starting from the edge (v_i, v_{i-1})) is also r-complete.

We prove the desired properties by induction on the levels (defined in section 6.1). For an example see Figure 2.

LEVEL(1). Because no edge of a simple path $\tilde{p}$ of length $2k$ was removed in the first step of the algorithm, each node from $\tilde{p}$ had to remove an incident edge not belonging to $\tilde{p}$. Therefore for each node from $\tilde{p}$ the side-branch is 1-complete. Similarly, since for each node from $\tilde{p}$ each path-branch has $w_1 = 1$, for each node v_i, $0 \le i \le 2k-1$, the right path-branch is 1-complete, and for each node v_i, $1 \le i \le 2k$, the left path-branch is 1-complete.

LEVEL(r). Now assume that $r > 1$ and for each node of $\tilde{p}$ the side-branch and the respective path-branches are $(r-1)$-complete. Consider a node v_i, $0 \le i \le 2k-r$, and the edge (v_i, v_{i+1}). Since the side-branch and the right path-branch of v_{i+1} are both $(r-1)$-complete and disjoint and have no cycle of length smaller than or equal to k, the right path-branch of v_i also must be r-complete. The nodes v_i, $r \le i \le 2k$, can be treated in a similar way.

This implies that we need a connected structure containing at least $2k\cdot 2^{k-1}$ nodes in H for a simple path of length $(\zeta+1)\cdot(2k+1)^2$ to survive the second phase of SIMULATION 3. For $k \ge \log\log n$ this contradicts Lemma 4.1. ☐

This yields the following result.

THEOREM 6.12. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM in $\mathcal{O}(\log\log\log n \log^* n)$ time, w.h.p.*

7. Reduction from CRCW PRAM to EREW PRAM.

In this section we show how to reduce the problem of simulating a CRCW PRAM on a DMM to the simulation of an EREW PRAM. It is standard to build simulations of CRCW PRAMs by combining sorting with simulations for EREW PRAMs. However, since we are interested in very fast simulations for which sorting is too slow, our reduction is based on strong semisorting. Suppose that processor P_i of the CRCW PRAM wants to access memory cell $u_i \in U$, for $1 \le i \le n$. In order to reduce the problem to the EREW PRAM we have to show how to deal with duplicate requests to the same memory cell. We consider only reading accesses; writing accesses can be performed similarly.

We first perform strong semisorting on pairs $(u_1, 1), \ldots, (u_n, n)$ with respect to the first coordinate. This results in the addresses with the same value being stored in a padded-consecutive subsequence of the output sequence. Using the algorithm for the all nearest one problem each u_i can test whether it is the first element in the padded-consecutive subsequence or not. If so, we call u_i a *leader*. Notice that if u_i is not a leader, then the solution to the all nearest one problem provides the pointer to its leader. Now we perform an EREW simulation on the DMM with requests only from leaders. Afterwards all leaders have accessed their cell u_i and can broadcast the value to the duplicates in constant time.

LEMMA 7.1. *If one step of an n-processor EREW PRAM can be simulated on an n-processor DMM with delay t, w.h.p., then one step of an n-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(t + \log^* n)$, w.h.p.*

Coupled with Theorem 6.12, this yields the following theorem.

THEOREM 7.2. *One step of an n-processor CRCW PRAM can be simulated on an n-processor DMM with $\mathcal{O}(\log^* n \log\log\log n)$ delay, w.h.p.*

8. All-but-linear routing.

In this section we analyze a relaxed version of the routing problem, the *all-but-linear routing* problem. Unlike in the general routing problem, we do not require that all messages are delivered to their destinations, but we have to route a large fraction of the messages. The motivation for our study is the problem of transforming simulations of n-processor PRAMs by n-processor DMMs into time-processor optimal simulations. As we will show in section 9, a fast solution for all-but-linear routing can be directly applied to obtain efficient time-processor optimal PRAM simulations.

DEFINITION 6. *In the* all-but-linear routing *problem on an n-processor DMM each processor Q_i has k keys $u_{i,1}, u_{i,2}, \ldots, u_{i,k}$, each to be sent to a destination from $M_0, \ldots, M_{n-1}$. The task is to deliver all but $\mathcal{O}(n)$ of the $n \cdot k$ keys.*

In our application we will need the destinations of the keys to be chosen almost randomly. The actual input for the PRAM simulation consists of *kn distinct* integers $x_{1,1}, \ldots, x_{n,k}$ from $[m]$, and we let $u_{i,j} = h(x_{i,j})$, for $i \in [n], j \in [k]$, where h is chosen uniformly at random from a $(2, \log^2 n)$-universal class $\mathcal{H}_{m,n}$ of hash functions (see section 3.1). We call such an instance of all-but-linear routing *quasi-random*. In fact, we could give a simpler solution if h were a truly random function (see [30]).

Our main result in this section is the following theorem.

THEOREM 8.1. *The quasi-random all-but-linear routing problem can be solved in time $\mathcal{O}(k + \log^* n)$, w.h.p., for $k \le \sqrt[4]{\log n}$.*

Our solution for quasi-random all-but-linear routing is based on an algorithm for the $k - k$ relation routing problem by Goldberg et al. [11].

DEFINITION 7. *In the $k - k$ relation routing problem, each processor wants to send at most k messages to other processors which are assumed to be distinct. The destinations can be arbitrary except that each processor is the destination of at most k messages.*

8.1. Reduction to $k - k$ relation routing.

We will reduce quasi-random all-but-linear routing to $k-k$ relation routing by achieving the following two goals. First, we will ensure that each processor has only keys with distinct destinations. Second, we will restrict the problem to processors that are destinations of $\mathcal{O}(k)$ messages.

The first problem can be solved using an element distinctness algorithm for each processor. That is, each processor Q_i sequentially runs through its keys $u_{i,1}, \ldots, u_{i,k}$ and finds all duplicate destinations. This can be done by using, for example, bucket sorting, in $\mathcal{O}(k)$ time and with $\mathcal{O}(n)$ space for each processor. Let us call a processor *bad* if it has keys that have the same destination.

LEMMA 8.2. *If $k = \mathcal{O}(\sqrt[4]{\log n})$, then $\mathcal{O}(\log^2 n)$ processors are bad, w.h.p.*

Proof. Let X_i, $i = 1, \ldots, n$, denote the binary random variable indicating if processor P_i is bad or not. Processor P_i is bad with probability at most $\binom{k}{2}\frac{2}{n} \le \frac{k^2}{n}$, that is, $P(X_i = 1) \le k^2/n$. Let $X = \sum X_i$. Since the keys have destinations chosen by a function h taken at random from a $(2, \log^2 n)$-universal class of hash functions, the choices of any $ek^2 \log n$ processors are almost independent (cf. Definition 1). In particular, we can give the following bound for $X = \sum_{i=1}^{n} X_i$:

$$\Pr(X \ge ek^2 \log n) \le \binom{n}{ek^2 \log n} 2 \left(\frac{k^2}{n}\right)^{ek^2 \log n} \le 2 \left(\frac{1}{\log n}\right)^{ek^2 \log n} \le \frac{1}{n^{\log \log n}}. \qquad \square$$

Hence, for all-but-linear routing, we can leave all keys of bad processors unprocessed, subsuming them in the $\mathcal{O}(n)$ remaining keys. To reduce the remaining all-but-linear routing to $k - k$ relation routing, we need the following lemma that bounds the number of keys that have their destination at modules with load at least $\Omega(k)$.

LEMMA 8.3. *Let h be chosen uniformly at random from a $(2, \log^2 n)$-universal class of hash functions with range $[n]$, and $1 \le k \le \frac{\log n}{\log \log n}$. If $n \cdot k$ keys are distributed among n locations using h, then at most $\frac{n}{2^k}$ keys are in locations with load at least $8k$, w.h.p.*

Proof. Denote the nk keys by $b_0, \ldots, b_{nk-1}$. A location is *heavy* if its load is at least $8k$. We associate with each key b_i a binary random variable $\mathcal{E}_i$ with $\mathcal{E}_i = 1$ if b_i is in a heavy location; otherwise $\mathcal{E}_i = 0$.

We want to bound the random variable $X = \sum_{i=0}^{kn-1} \mathcal{E}_i$. Again we use the kth moment inequality (Lemma 3.2) and therefore compute $E(X^s)$ for $s = \Theta(\log n)$.

$$E(X^s) = \sum_{(j_1,\ldots,j_s)\in[kn]^s} E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s}) = \sum_{z=1}^{s} \sum_{\substack{(j_1,\ldots,j_s)\in[kn]^s \\ z \text{ of the } j_i \text{ are different}}} E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s}) \ .$$

First we bound each term $E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s})$. Fix keys $(j_1, \ldots, j_s) \in [kn]^s$ such that z of the j_i are different. $E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s})$ is equal to the probability that each of the z keys has its destination in a heavy location. This probability is bounded by the sum over all possible locations of the z keys of the probabilities that the z keys have destinations in given locations and that the locations are heavy. Since h is chosen from a $(2, \log^2 n)$-universal class of hash functions and $z < \log^2 n$, the probability that the z keys have destinations in given r locations, $1 \le r \le z$, is bounded by $r^z \cdot 2/n^z$. For $r \le z/8k$, we trivially estimate the probability that r locations are heavy by 1. If $z/8k < r \le z$, then we observe that in order to r locations be heavy there must be other $8kr - z$ keys with destinations in the r locations. Therefore in that case we bound the probability that the r locations are heavy by $\binom{nk}{8kr-z} \cdot 2 \cdot (r/n)^{8kr-z}$.

Hence, we obtain

$$\begin{aligned} E(\mathcal{E}_{j_1}\cdots\mathcal{E}_{j_s}) &\le \sum_{1\le r\le \frac{z}{8k}} \binom{n}{r} 2\left(\frac{r}{n}\right)^z + \sum_{\frac{z}{8k}<r\le z} \binom{n}{r} 2\left(\frac{r}{n}\right)^z \binom{nk}{8kr-z} 2\left(\frac{r}{n}\right)^{8kr-z} \\ &\le 2\sum_{r\le\frac{z}{8k}} \left(\frac{ne}{r}\right)^r \left(\frac{r}{n}\right)^z + 4\sum_{\frac{z}{8k}<r\le z} \left(\frac{ne}{r}\right)^r \left(\frac{r}{n}\right)^z \left(\frac{nker}{(8kr-z)n}\right)^{8kr-z} \\ &\le 4\left(\frac{8ekn}{z}\right)^{\frac{z}{8k}} \left(\frac{z}{8kn}\right)^z + 8\left(\frac{en}{z}\right)^z \left(\frac{z}{n}\right)^z \left(\frac{kez}{(8k-1)z}\right)^{(8k-1)z} \\ &\le 4\left(\frac{z}{\sqrt{n}}\right)^z + \left(\frac{1}{2}\right)^{8(k-1)z} \\ &\le 5\left(\frac{1}{2}\right)^{8(k-1)z} . \end{aligned}$$

The third inequality holds for $k = \mathcal{O}(\log n/\log\log n)$ and $z = o(n)$, which holds as $z \le s = \Theta(\log n)$. In this case it is easily seen that the terms in each sum at least double when r increases. Hence we can bound each sum by two times the largest term. Now we can bound $E(X^s)$:

$$\begin{aligned} E(X^s) &\le \sum_{z=1}^{s} \sum_{\substack{(j_1,\dots,j_s)\in[kn]^s \\ z \text{ of the } j_i \text{ are different}}} 5\left(\frac{1}{2}\right)^{8(k-1)z} \\ &\le \sum_{z=1}^{s} \binom{s}{z} (kn)^z z^{s-z} 5\left(\frac{1}{2}\right)^{8(k-1)z} \\ &\le 5\sum_{z=1}^{s} z^s \left(\frac{sekn}{z^2 2^{8(k-1)}}\right)^z \\ &\le 10 s^s \left(\frac{ekn}{s 2^{8(k-1)}}\right)^s \\ &\le \left(\frac{n}{2^{2k}}\right)^s . \end{aligned}$$

Again, each term of the sum in the third line at least doubles for large enough n if z increases. Hence, we have bounded the sum by two times the largest term.

As the $\mathcal{E}_i$ and therefore also X are nonnegative random variables, we finally get

$$\Pr(X \ge \alpha) \le \frac{E(X^s)}{\alpha^s} \le \left(\frac{n}{2^{2k}\alpha}\right)^s .$$

For $\alpha = \frac{n}{2^k}$ we get

$$\Pr\left(X \ge \frac{n}{2^k}\right) \le \left(\frac{1}{2^k}\right)^s .$$

For any positive l, if $s \ge l \log n$, then this probability is bounded by n^{-l}. ☐

8.2. $k-k$ relation routing. It remains to show that we can perform the $k-k$ relation routing in time $\mathcal{O}(k)$. As was observed by Anderson and Miller [1], one can solve the off-line version of the problem in k steps. However, we are interested in this problem when each processor has only the information about the messages it has to send, and it can learn about the messages of the other processors only by sending and receiving messages.

It is easy to see that any $k-k$ relation routing can be performed in time k^2 on a DMM, but our aim is to do this faster. Anderson and Miller [1] considered the $k-k$ relation routing problem on a model equivalent to a 1-collision DMM and showed how to solve $\log n - \log n$ relation routing in $\mathcal{O}(\log n)$ expected time. Valiant [33] considered the more general problem. He described a $\Theta(\log n + k)$ expected time algorithm on the same model. This result was improved by Goldberg et al. [11], to an algorithm for the $k-k$ relation routing problem running in $\mathcal{O}(\log\log n + k)$ time, w.h.p. We describe a faster algorithm on an DMM.

THEOREM 8.4. *The $k-k$ relation routing problem can be solved on an n-processor DMM in $\mathcal{O}(\min\{\log^* n + k, k^2\})$ communication steps for any $k \le n^{1/11}$.*

The proof of this theorem uses known methods and techniques from [1, 11]. In fact it is very similar in spirit to the results presented there. Nevertheless we present the full proof in order to make our description of the all-but-linear routing algorithm self-contained.

To prove the theorem we first describe the randomized routing protocol. It consists of $\mathcal{O}(\log k)$ rounds. In round i the problem of a $k/2^{i-1} - k/2^{i-1}$ relation routing will be reduced to a $k/2^i - k/2^i$ relation routing problem. After the last round all but n/k of the $n \cdot k$ keys will be delivered. Then we redistribute the remaining keys among the processors and deliver them in $\mathcal{O}(k)$ steps, w.h.p.

Before presenting more details we introduce some notation. We call a processor *overloaded* at the beginning of round i if the number of keys it has not already delivered is greater than $k/2^{i-1}$. Similarly we call a module *overloaded* at the beginning of phase i if the number of keys that it still has to receive is greater than $k/2^{i-1}$. A module or a processor, respectively, becomes overloaded in round i if it was not overloaded at the beginning of round i but it is at the beginning of round $i+1$. The idea of the algorithm is that in each round only nonoverloaded processors *participate*, i.e., only those keys participate that are not sent by an overloaded processor. Additionally we consider only keys sent to a nonoverloaded module. The main problem is to bound the number of overloaded processors and modules and hence the number of keys that are left after the $\log k$ rounds. Let c be a positive constant, $c \ge 80$.

$k-k$ RELATION ROUTING.

```
(1) For $1 \le i \le \log(\frac{k}{5\log k}) + 1$, perform the following round $i$:
    Only processors that have fewer than $k/2^{i-1}$ keys left to send participate.
      • Repeat $c \cdot k/2^i$ times:
        Each participating processor chooses a random number $r$ from $[\frac{k}{2^{i-1}}]$ and
        tries to send the $r$th undelivered message, if it exists.
(2) All processors that have fewer than $5\log k/2$ nondelivered keys send the keys
    one by one, each key $5\log k/2$ times.
(3) Redistribute evenly the remaining keys using approximate parallel prefix sums
    applied to the numbers of nondelivered keys belonging to each processor.
(4) Deliver the remaining keys.
```

The proof of the theorem is based on the following lemma which bounds the number of keys not participating.

LEMMA 8.5. *Let $k \le n^{1/11}$, and let i be an arbitrary round in the first step of the $k-k$ relation routing algorithm above. Then at most n/k^4 modules and at most n/k^4 processors become overloaded in round i, w.h.p.*

Proof. Consider an arbitrary round i of the algorithm, $1 \leq i \leq (\frac{k}{5\log k}) + 1$. Let x_j denote the sequence of integers randomly chosen by processor P_j during round i.

We first prove that the number of overloaded modules at the beginning of round $i+1$ can be bounded by n/k^4, w.h.p. Let Y_M be the number of modules that become overloaded during round i. Let M be a module that is not overloaded at the beginning of round i, and let m_j denote the number of participating keys that are destined for M at the jth step of round i. Notice that M cannot become overloaded in round i if m_j is less than $k/2^i$. Therefore, in that case, the probability that M does not receive a message in the jth step is

$$\left(1 - \frac{1}{k/2^{i-1}}\right)^{m_j} \leq \left(1 - \frac{1}{k/2^{i-1}}\right)^{k/2^i} \leq e^{-1/2} \ .$$

Observe now that if M is overloaded at the end of round i, then in at least $ck/2^i - k/2^i$ steps of this round no message is received by M. Therefore, we can bound the probability that M is overloaded at the end of round i by

$$\binom{c \cdot k/2^i}{k/2^i} \cdot (e^{-1/2})^{c\cdot k/2^i - k/2^i} \leq (ec)^{k/2^i} \cdot (e^{-1/2})^{(c-1)\cdot k/2^i} = \left(c \cdot e^{3/2 - c/2}\right)^{k/2^i} \ .$$

Since we have assumed that $i \leq \log(\frac{k}{5\log k}) + 1$ and $c \geq 80$, we may bound the probability by $1/2k^4$. This immediately implies that $E(Y_M)$, the expected number of modules that become overloaded in round i, is at most $n/2k^4$. Now we apply the method of bounded differences (Lemma 3.1) to bound the probability that $Y_M \geq n/k^4$. For this we view Y_M as the image of a function f_M:

$$Y_M = f_M(\{x_j | P_j \text{ is participating at the beginning of round } i\}) \ .$$

Observe that any change of the choice of P_j in a single step from key α to β may change only the load of the two modules being destinations of α and β. Since all choices of P_j are independent, a change of the value of x_j for any j may change the number of modules overloaded in round i by at most $2ck/2^i \leq ck$. Hence we apply Lemma 3.1 to get

$$\Pr(Y_M \geq n/k^4) \leq \Pr(Y_M \geq n/2k^4 + E(Y_M)) \leq \exp\left(-\frac{2(n/k^4)}{n(ck)^2}\right) = \exp\left(-\frac{2n}{c^2k^{10}}\right).$$

This probability is exponentially small for $k \leq n^{1/11}$.

Now we want to show that, w.h.p., at most n/k^4 processors become overloaded during round i. Let Y_P be the number of processors that become overloaded during round i. Let P be any participating processor in round i and let p_j denote the number of keys that P still has to send in the jth step of round i.

Let $d_{l,j}$ denote the number of participating keys in the jth step that have the same destination as the lth key that P has to send. We consider only those P that do not send to overloaded modules. Therefore each $d_{l,j}$ is less than or equal to $k/2^{i-1}$. Note that P cannot become overloaded in round i if p_j is ever less than $k/2^i$. Thus we consider only the case $p_j \geq k/2^i$ for all j. In that case the probability that P sends a key successfully in the jth step is at least

$$\sum_{l=1}^{p_j} 2^{i-1}/k \cdot \left(1 - 2^{i-1}/k\right)^{d_{l,j}-1} \geq \sum_{l=1}^{p_j} 2^{i-1}/k \cdot \left(1 - 2^{i-1}/k\right)^{k/2^{i-1}} \geq 1/2e^2 \ .$$

Therefore, using arguments similar to those above, we obtain that the probability that P stops participating in round i is at most

$$\binom{ck/2^i}{k/2^i} \cdot (1-1/2e^2)^{(c-1)k/2^i} \leq \left(e \cdot c \cdot (1-1/2e^2)^{c-1}\right)^{k/2^i} .$$

Since $i \leq \log(\frac{k}{5\log k}) + 1$ and we set $c \geq 80$, this probability is bounded by $1/2k^4$. As in the proof of the first part, we can conclude that the expected number of processors that become overloaded during round i is at most $\frac{n}{2k^4}$. One can verify that if the value of x_j changes for any j, then Y_P changes by at most ck. Therefore, by the method of bounded differences (Lemma 3.1), the probability that Y_P is greater than n/k^4 is at most

$$\Pr\left(Y_P \geq \frac{n}{k^4}\right) \leq \Pr\left(Y_P \geq \frac{n}{2k^4} + E(Y_P)\right) \leq \exp\left(-\frac{2(n/k^4)^2}{n \cdot (ck)^2}\right) = \exp\left(-\frac{2n}{c^2k^{10}}\right) .$$

This probability is exponentially small for $k \leq n^{1/11}$. □

Now we proceed with the proof of Theorem 8.4.

Proof of Theorem 8.4. After the first step of $k-k$ RELATION ROUTING a key may be left if either its processor has fewer than $5\log k/2$ nondelivered keys, or its processor is overloaded in some round and stops participating, or the module where the key should be sent is overloaded in some round. If the key is in the processor with fewer than $5\log k/2$ nondelivered keys, then after sending it $5\log k/2$ times in the second step either it will be delivered to the destination, or there are more than $5\log k/2$ keys sending to the destination module. The latter means that the module is overloaded at some round in the first step. Therefore, after the second step, we are left only with such keys for which either the processor to which they belong or the corresponding destination module is overloaded at some round in the first step. Because an overloaded processor affects at most k keys and an overloaded module affects at most k^2 keys, the number of keys that are left at the end of the $\log k$ rounds can be bounded using Lemma 8.5 by

$$\log k\left(k^2\frac{n}{k^4} + k\frac{n}{k^4}\right) \leq \frac{n}{k} .$$

Using approximate parallel prefix computation (Lemma 3.3) applied to the number of nondelivered keys belonging to each processor, we can redistribute the n/k remaining keys among the modules in $\mathcal{O}(\log^* n)$ time, such that each processor gets at most a constant number, w.h.p. Finally, each processor can now deliver its keys in $\mathcal{O}(k)$ steps.

Hence, altogether we need $\mathcal{O}(\log^2 k + k + \log^* n)$ time. □

At the beginning of $k-k$ relation routing a module is called *overloaded* if it gets $\omega(k)$ requests.

Lemma 8.3 bounds the number of keys affected by the modules getting $\omega(k)$ requests by $kn/2^k$. These keys and the keys from the bad processors have to be added to the other keys that will not be processed during the $\mathcal{O}(\log k)$ rounds of $k-k$ relation routing. Therefore, the total number of keys not processed during the $\mathcal{O}(\log k)$ rounds of $k-k$ relation routing remains linear. This finishes the description of quasi-random all-but-linear routing and completes the proof of Theorem 8.1.

9. Time-processor optimal EREW PRAM simulations. In this section we want to present time-processor optimal simulations of EREW PRAMs on DMMs, that is, simulations of $(n \cdot t)$-processor EREW PRAMs on n-processor DMMs with delay not exceeding $\mathcal{O}(t)$, w.h.p. For this we first consider the problem of accessing 1 out of c copies of the $(n \cdot t)$ requested distinct memory cells on an n-processor DMM. The copies are distributed using c independently and randomly chosen functions from a $(2, \log^2 n)$-universal class of hash functions. We show that such a problem can be reduced in time $\mathcal{O}(t + \log^* n)$, w.h.p., to a constant number of independent "1 out of $c - 1$" tasks, provided that $t \leq \sqrt[4]{\log n}$. Next we apply this reduction to obtain our time-processor optimal simulations of an $(n \cdot t)$-processor EREW PRAM on an n-processor DMM.

Let us consider the problem of accessing 1 out of the c copies of the $(n \cdot t)$ requested distinct memory cells, where the copies are distributed using independent hash functions $h_1, \ldots, h_c$. We assume that each processor of the DMM has a list of t keys (memory requests). We consider only the situation where $t \leq \sqrt[4]{\log n}$. We first use all-but-linear routing to satisfy all but $\mathcal{O}(n)$ of the access requests, by accessing the copies of the keys stored in memory modules given by h_1. Theorem 8.1 ensures that this phase can be performed in time $\mathcal{O}(t + \log^* n)$, w.h.p. Next, the remaining $\mathcal{O}(n)$ access requests are redistributed evenly among the processors of the DMM such that each processor gets $\mathcal{O}(1)$ of them. For that we use approximate prefix sums with respect to the number of nondelivered keys of each processor and then accordingly redistribute the keys. By Lemma 3.3, approximate prefix sums require $\mathcal{O}(\log^* n)$ time, w.h.p. Given computed approximate prefix sums, the distribution phase can be done in time $\mathcal{O}(t)$. Let us observe that, since the hash functions $h_2, h_3, \ldots, h_c$ are independent of h_1, the remaining $\mathcal{O}(n)$ requests are independent of $h_2, h_3, \ldots, h_c$. Therefore, we can use them as the input for a constant number of "1 out of $c - 1$" tasks on the basis of hash functions $h_2, h_3, \ldots, h_c$. Thus the above algorithm reduces the initial problem to a constant number of "1 out of $c-1$" tasks in time $\mathcal{O}(t+\log^* n)$, w.h.p.

By our discussion in section 2, a simulation of one step of an $(n \cdot t)$-processor EREW PRAM on an n-processor DMM can be reduced to solving independently $\binom{a}{b-1}$ times (for $b > a/2$) the problem of accessing 1 out of the $a - b + 1$ copies of the $(n \cdot t)$ requested distinct memory cells. Therefore, the algorithm above implies the following theorem.

THEOREM 9.1. *If one can execute a "1 out of $a - b$" task ($b > a/2$) in time bounded by $t \leq \sqrt[4]{\log n}$, w.h.p., then one step of an $(n \cdot t)$-processor EREW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\binom{a}{b-1} \cdot (t + \log^* n))$, w.h.p.*

Hence we can combine Theorem 9.1 with the algorithm for the "1 out of 2" task described in section 6.4 to obtain the following theorem.

THEOREM 9.2. *One step of an $(n \log\log\log n \log^* n)$-processor EREW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\log\log\log n \log^* n)$, w.h.p.*

We finally note that Karp, Luby, and Meyer auf de Heide [19] gave a slightly weaker theorem than Theorem 9.1 for CRCW PRAM simulations. We may apply it to get the following.

THEOREM 9.3. *One step of an $(n \log\log\log n \log^* n)$-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\log\log\log n (\log^* n)^2)$, w.h.p.*

10. A lower bound for the topological game. In this section we pinpoint the limits of our approach based on the "1 out of 2" task. We show that all previously known solutions based on the "1 out of 2" task as well as our algorithms are special cases of a game on the access graph, called the *topological game*. Then we prove a lower bound for the topological game.

All simulations based on the “1 out of 2” task, or equivalently on the analysis of the access game, have the following common scheme. In each step i each node v of H either analyzes its $k_{v,i}$-neighborhood or, on the basis of the structure of its $k_{v,i}$-neighborhood, decides which edge to remove, where $k_{v,i}$ is some integer. We can describe this scheme in a slightly more general way. Let k be the maximum of $k_{v,i}$ taken over all nodes v of H and all steps i of the algorithm. In the first superstep each node finds and analyzes its k-neighborhood. Then, in the remaining rounds it decides which edge to remove on the basis of the current k-neighborhood. Here we assume that each node will get the information on all removed edges in its k-neighborhood for free. It is easy to see that this extension makes the algorithm at least as powerful as the original one.

Let H be the access graph as defined in section 4 with n nodes and n/c edges, for $c < \log\log n$. Further assume that H is composed using two independent random hash functions. We define the *topological game* on the access graph H as follows. k is a parameter for the game.

Topological Game.

(1) Each node of H finds and analyzes its k-neighborhood; we charge for this $\log k$ steps.
(2) Repeat the following round until all the edges are removed:
    (a) Each node of degree one removes all edges in its k-neighborhood.
    (b) Each other node removes one incident edge. The choice of this edge solely depends on the topology of the current topology of its k-neighborhood.

We stress here the following features of this scheme.

Explanation and discussion of step 10. Let us observe that in all previous simulations [8, 12, 19, 22, 25] only a 1-neighborhood was analyzed. On the other hand, we notice that the time needed for computing any information about the k-neighborhood of a node v that involves the knowledge on any node that is in distance k from v seems to require $\Omega(\log k)$ steps. We remark that our data structures presented in section 6 allow each processor to obtain useful information about its k-neighborhood in $\mathcal{O}(\log k \cdot \log^* n)$ time, w.h.p.

Explanation and discussion of step 0a. This step is motivated by the fact that using a slight modification of the algorithm Cleaning Up Connected Components, each node that is close to the border of its connected component (that is, a node whose k-neighborhood contains a simple path of length less than k that cannot be extended by any edge to a simple path) can be removed in constant time using only the information about the k-neighborhood.

Explanation and discussion of step 0b. Here we make a key assumption concerning the topological game. A node bases its decision of which edge to choose on the topologies of the $(k-1)$-neighborhoods of its (direct) neighbors. The names of the nodes and the labels of the edges are not allowed to be used. In particular, in the case of all $(k-1)$-neighborhoods being disjoint and isomorphic, the node can only choose a random incident edge to remove. We further assume that removing further edges during the game does not increase the number of rounds required. All strategies known so far fit into this model. Especially, no simulation strategy is known that takes advantage of using the labels of nodes and edges to break ties, i.e., to choose among neighbors with disjoint isomorphic $(k-1)$-neighborhoods. On the other hand, it is challenging to find simulations that take advantage of using the labels, or to extend our lower bound.

The main idea of the lower bound for the topological game is to focus only on fully symmetric structures in H. We show that there exists a node that has a (topologically) symmetric k-neighborhood, possibly after removing some edges, and does not contain nodes of degree one in its k-neighborhood. Then we assume that it randomly makes the decision which outgoing edge to remove. We show that, after performing these random decisions, a smaller symmetric subgraph will be left in the next round, with sufficiently high probability. The bound for decreasing the size of the symmetric subgraph will yield the lower bound.

For $0 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$, define the values of δ_i and d_i as follows:

$$\delta_i = \frac{\log\log n}{(\log\log\log n)^{4(i+1)}} \quad \text{and} \quad d_i = \frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}} \ .$$

We will use the following inequalities for values δ_i and d_i.

LEMMA 10.1. *For every $1 \le k \le \sqrt{\log\log n}$, $0 \le i < \frac{\log\log\log n}{8\log\log\log\log n} - 1$, and sufficiently large n the following inequalities hold:*

(i) $2e^{-\frac{2\cdot\delta_i^{d_i}}{2^{2(\delta_i-\delta_{i+1})}}} \le \frac{1}{\log n}$ *and*

(ii) $\frac{3\cdot\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}} \le \delta_i^{d_i-d_{i+1}-2k}$.

Proof.

(i) It is enough to show that

$$d_i\log\delta_i - 2\delta_i + 2\delta_{i+1} \ge \log\ln(2\log n).$$

If we substitute the terms that define δ_i and d_i on the left-hand side and use the assumption about the range of i, we get the following inequality:

$$\begin{aligned}
&\frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}}(\log\log\log n - 4(i+1)\log\log\log\log n)\\
&-2\left(\frac{\log\log n}{(\log\log\log n)^{4(i+1)}} + \frac{\log\log n}{(\log\log\log n)^{4(i+2)}}\right)\\
&\ge \frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}}\\
&\ge \sqrt{\log\log n}\,(\log\log\log n)^2\\
&\ge \log\ln(2\log n).
\end{aligned}$$

(ii) It is enough to show that

$$\delta_i - \delta_{i+1} - 3 - d_{i+1}\log\delta_i - 2k\log\delta_i \ge 0.$$

It is easily checked that $2k \le d_{i+1}$ for all i in the indicated range. This yields the following inequalities if we substitute the terms that define δ_i and d_i on the left-hand side and use the assumption about the range of i and k:

$$\begin{aligned}
&\delta_i - \delta_{i+1} - 3 - d_{i+1}\log\delta_i - 2k\log\delta_i\\
&\ge \delta_i - \delta_{i+1} - 3 - 2d_{i+1}\log\delta_i\\
&= \frac{\log\log n}{(\log\log\log n)^{4(i+1)}} - \frac{\log\log n}{(\log\log\log n)^{4(i+2)}} - 3\\
&\quad -2\frac{\log\log n}{(\log\log\log n)^{4(i+2)-2}}\log\left(\frac{\log\log n}{(\log\log\log n)^{4(i+1)}}\right)
\end{aligned}$$

$$\geq \frac{\log\log n}{2(\log\log\log n)^{4(i+1)}} - 3 - 2\frac{\log\log n}{(\log\log\log n)^{4(i+2)-3}}$$
$$\geq \frac{\log\log n}{4(\log\log\log n)^{4(i+1)}} \geq 0 \ . \qquad \square$$

DEFINITION 8. *A (δ_i, d_i)-tree is an undirected, unlabeled, and acyclic connected graph that contains a vertex r of degree δ_i such that all vertices in distance (exactly) d_i from r are of degree 1 and all other vertices are of degree δ_i.*

Fix an algorithm A that performs the topological game. We say a labeled directed graph A contains a copy of an unlabeled undirected graph B if, after removal of the labels and the edge orientation in A, the obtained graph contains a subgraph isomorphic to B. We want to maintain the condition that the access graph remaining after performing i rounds of the algorithm contains a copy of a (δ_i, d_i)-tree with sufficiently high probability. The proof of this invariant is done by induction, which is based on the following two lemmas.

LEMMA 10.2. *Let G be the directed access graph with n labeled nodes and n/c labeled edges (cf. section 4.1). Let $c < \log\log n$, and let $\mathfrak{T}_q$ be a fixed unlabeled and undirected tree with q nodes. For $q \leq \frac{\log n}{10\log\log n}$ and sufficiently large n, the probability that G contains a copy of $\mathfrak{T}_q$ is at least $1/2$.*

The proof of the lemma is in the spirit of the proof of a similar lemma for balanced graphs due to Erdős and Rényi [9].

Proof. Since the lemma trivially holds for $q = 1$, we assume that $q \geq 2$. Let $\mathcal{T}_q^{(n)}$ be the set of all subgraphs of the complete directed graph on n nodes and n/c edges labeled by the elements of S (cf. section 4.1) that, after removal of the labels and the edge orientations, are isomorphic to $\mathfrak{T}_q$. With each $T \in \mathcal{T}_q^{(n)}$ we associate a binary random variable $\mathcal{E}(T)$, such that $\mathcal{E}(T) = 1$ or $\mathcal{E}(T) = 0$ according to whether T is a subgraph of G or not. Our aim is to bound the probability that $\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T) = 0$. We estimate that probability by using Chebyshev's inequality. For that purpose we now provide bounds for the expectation and the variance of the random variable $\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)$.

First consider the expected value $E(\sum \mathcal{E}(T))$. The probability that a fixed tree with $q-1$ edges is a subgraph of G is $(1/n^2)^{q-1}$. In order to obtain a lower bound on the number of trees in $\mathcal{T}_q^{(n)}$ we observe that we may obtain a subset of $\mathcal{T}_q^{(n)}$ by selecting trees with given q nodes and given labels assigned to the edges of the tree. Hence we obtain the following formula:

$$E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) = \sum_{T\in\mathcal{T}_q^{(n)}} E(\mathcal{E}(T))$$
$$= |\mathcal{T}_q^{(n)}| \cdot \left(\frac{1}{n^2}\right)^{q-1} \tag{1}$$
$$\geq \binom{n}{q} \frac{\left(\frac{n}{c}\right)!}{\left(\frac{n}{c} - (q-1)\right)!} \left(\frac{1}{n^2}\right)^{q-1}$$
$$\geq \left(\frac{n}{q}\right)^q \left(\frac{n}{2c}\right)^{q-1} n^{-2(q-1)}$$
$$\geq \frac{n}{(2cq)^q} \ . \tag{2}$$

Let T_1 and T_2 be two elements from $\mathcal{T}_q^{(n)}$. If T_1 and T_2 have no edges in common, then

$$E(\mathcal{E}(T_1)\mathcal{E}(T_2)) = \left(\frac{1}{n^2}\right)^{2(q-1)} .$$

Therefore

$$\begin{aligned} E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right) &= |\{T_1,T_2\in\mathcal{T}_q^{(n)} : T_1,T_2\,\text{edge disjoint}\}|\cdot\left(\frac{1}{n^2}\right)^{2(q-1)} \\ &\le (|\mathcal{T}_q^{(n)}|)^2\cdot\left(\frac{1}{n^2}\right)^{2(q-1)} \\ &\overset{(1)}{\le} \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right)\right)^2 . \end{aligned} \tag{3}$$

If T_1 and T_2 have exactly r edges in common ($1 \le r \le q-1$), then we get

$$E(\mathcal{E}(T_1)\mathcal{E}(T_2)) = \left(\frac{1}{n^2}\right)^{2(q-1)-r} .$$

Because T_1 and T_2 are trees, they have at least $r+1$ nodes in common. Hence, we can give an upper bound for the number of such pairs T_1, T_2 of subgraphs:

$$\begin{aligned} &(2^q\cdot q^{q-2})^2\sum_{j=r+1}^{q}\binom{n}{q}\binom{q}{j}\binom{n-q}{q-j}\left(\frac{n}{c}\right)^{2(q-1)-r} \\ &\le q^{4q-4}\sum_{j=r+1}^{q}\frac{n!}{q!(n-q)!}\frac{q!}{j!(q-j)!}\frac{(n-q)!}{(q-j)!(n-2q+j)!}\left(\frac{n}{c}\right)^{2(q-1)-r} \\ &\le q^{4q-4}\sum_{j=r+1}^{q}\frac{n^{2q-j+2(q-1)-r}}{j!((q-j)!)^2c^{2(q-1)-r}} \\ &\le q^{4q-4}qn^{4q-2r-3} . \end{aligned}$$

Hence we obtain

$$E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{not edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right) \le \sum_{r=1}^{q-1} q^{4q-3}n^{4q-2r-3}\left(\frac{1}{n^2}\right)^{2(q-1)-r} \le nq^{4q-2} . \tag{4}$$

Now we may bound $E((\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T))^2)$:

$$E\left(\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right)^2\right)$$

$$= E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) + E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right)$$

$$+ E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{not edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right)$$

$$\overset{(3)(4)}{\le} E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) + \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right)\right)^2 + nq^{4q-2} \ . \tag{5}$$

Finally we provide an upper bound for the variance:

$$\operatorname{Var}\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) = E\left(\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right)^2\right) - \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right)\right)^2$$

$$\overset{(5)}{\le} E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) + nq^{4q-2} \ . \tag{6}$$

Using the Chebyshev inequality we can bound the probability that no $T \in \mathcal{T}_q^{(n)}$ is a subgraph of G:

$$\begin{aligned}
\Pr\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T) = 0\right) &\le \frac{\operatorname{Var}\left(\sum \mathcal{E}(T)\right)}{\left(E\left(\sum \mathcal{E}(T)\right)\right)^2} \\
&\overset{(6)}{\le} \frac{\sum E(\mathcal{E}(T)) + nq^{4q-2}}{\left(E\left(\sum \mathcal{E}(T)\right)\right)^2} \\
&\overset{(2)}{\le} \frac{(2cq)^q}{n} + \frac{nq^{4q-2}(2cq)^{2q}}{n^2} \\
&= \frac{(2cq)^q + (2c)^{2q}q^{6q-2}}{n} \\
&\le \frac{q^{10q}}{n} \qquad \text{for} \quad c \le q \\
&\le \frac{1}{2} \qquad \text{for} \quad q \le \frac{\log n}{10\log\log n} \ .
\end{aligned}$$

Therefore, the probability that G contains a copy of $\mathfrak{T}_q$ is at least $1/2$ for $q \le \frac{\log n}{10\log\log n}$. ◻

LEMMA 10.3. *Consider a (δ_i, d_i)-tree for $0 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$, and let $k \le \sqrt{\log\log n}$. If every node randomly removes an incident edge, then, with probability at least $1 - 1/\log n$, at most $\delta_i^{d_i - d_{i+1} - 2k}$ of the nodes have degree smaller than δ_{i+1}.*

Proof. Consider a fixed node v with degree δ_i. Each edge incident to v will be removed by an incident node with the same probability $1/\delta_i$. Define for $1 \le j \le \delta_i$ the binary random variable X_j which is one if and only if the jth incident edge of v will be removed by an incident node. We know that $\Pr(X_j = 1) = 1/\delta_i$. Let $X = \sum X_j$. Note that $E(X) = 1$. We want to bound the deviation from the expected value. Because the decisions of the adjacent nodes are independent, we can use the Chernoff bound [15]:

$$\Pr(X \ge \alpha) = \Pr(X \ge \alpha E(X)) \le \frac{1}{2^\alpha} \text{ for } \alpha \ge 5 \enspace .$$

In particular, this implies that

$$\Pr(\text{a node has degree smaller than } \delta_{i+1}) \le \left(\frac{1}{2}\right)^{\delta_i - \delta_{i+1} - 1} \enspace .$$

Let x_l, $1 \le l \le \frac{\delta_i^{d_i+1}-1}{d_i-1}$, be the sequence of independent random decisions made by the $\frac{\delta_i^{d_i+1}-1}{d_i-1}$ nodes of the (δ_i, d_i)-tree. The random variable

$$Y = f\left(\left\{x_l : 1 \le l \le \frac{\delta_i^{d_i+1}-1}{d_i-1}\right\}\right)$$

denotes the number of nodes that have degree smaller than δ_{i+1}. If we change the value of one x_l, Y can be changed by at most two. The expected number of nodes having degree less than δ_{i+1} can be bounded by

$$\frac{\frac{\delta_i^{d_i+1}-1}{d_i-1}}{2^{\delta_i-\delta_{i+1}-1}} \le 2\frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}} \enspace .$$

Using the method of bounded differences (Lemma 3.1), we get

$$\begin{aligned}\Pr\left(Y \ge 3\frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}}\right) &\le \Pr\left(Y \ge E(Y) + \frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}}\right) \\ &\le 2e^{-2\frac{\left(\delta_i^{d_i}\right)^2}{2^{2(\delta_i-\delta_{i+1}-1)}}\cdot\frac{d_i-1}{4\left(\delta_i^{d_i+1}-1\right)}} \\ &\le 2e^{-\frac{\delta_i^{d_i}}{2^{2(\delta_i-\delta_{i+1})}}} \enspace .\end{aligned}$$

Substituting the values of δ_i and d_i into this formula and using Lemma 10.1 we get that for $k \le \sqrt{\log\log n}$ the number of nodes with small degree can be bounded by $\delta_i^{d_i-d_{i+1}-2k}$ with probability at least $1 - 1/\log n$. ◻

Using these two lemmas we can show the following.

LEMMA 10.4. *After performing i rounds of the algorithm A, a (δ_i, d_i)-tree is a subgraph of the remaining access graph with probability at least $(1 - 1/\log n)^i$ for $1 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$ and $k \le \sqrt{\log\log n}$.*

Proof. The proof is done by induction on i. For $i = 0$ we use Lemma 10.2. Assume that the lemma holds for some i, where $i < \frac{\log\log\log n}{8\log\log\log\log n}$. From the induction hypothesis we know that a (δ_i, d_i)-tree is a subgraph of the access graph at the beginning of round $i+1$. We consider only the edges from this tree and remove all other edges. Because of the definition of the topological game, we remove all nodes that are within distance at most k from a leaf of the tree. Each decision based on the topology of the k-neighborhood made by the remaining nodes is random and is independent of decisions of other nodes.

Hence, using Lemma 10.3, at most $\delta_i^{d_i - d_{i+1} - 2k}$ nodes have degree smaller than δ_{i+1}, and a (δ_i, d_i)-tree is a subgraph of the remaining graph. □

The invariant of Lemma 10.4 holds for k small enough in every round i, $1 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$, even if we start only with $\frac{n}{\log\log\log n}$ edges. This implies the following theorem.

THEOREM 10.5. *The expected number of rounds in any topological game until all edges of the access graph H will be removed is* $\Omega(\frac{\log\log\log n}{\log\log\log\log n})$.

Proof. After $\frac{\log\log\log n}{8\ \log\log\log\log n}$ rounds it is only possible to pick a k-neighborhood for

$$k \le 2^{\frac{\log\log\log n}{8\ \log\log\log\log n}} \le \sqrt{\log\log n}\ .$$

Therefore, when we use Lemma 10.4, after $i \le \frac{\log\log\log n}{8\log\log\log\log n}$ rounds some edges will be left. The probability for this event can be bounded by

$$\left(1 - \frac{1}{\log n}\right)^{\frac{\log\log\log n}{8\log\log\log\log n}} \ge 1/e\ . \qquad □$$

REFERENCES

[1] R. J. ANDERSON AND G. L. MILLER, *Optical Communication for Pointer Based Algorithms*, Tech. Report CRI 88-14, University of Southern California, Los Angeles, CA, 1988.

[2] H. BAST AND T. HAGERUP, *Fast and reliable parallel hashing*, in Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1991, pp. 50–61.

[3] H. BAST AND T. HAGERUP, *Fast parallel space allocation, estimation and integer sorting*, Inform. and Comput., 123 (1995), pp. 72–110.

[4] O. BERKMAN AND U. VISHKIN, *Recursive star-tree parallel data structure*, SIAM J. Comput., 22 (1993), pp. 221–242.

[5] J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[6] M. DIETZFELBINGER, A. KARLIN, K. MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT, AND R. E. TARJAN, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[7] M. DIETZFELBINGER AND F. MEYER AUF DER HEIDE, *How to distribute a dictionary in a complete network*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1990, pp. 117–127.

[8] M. DIETZFELBINGER AND F. MEYER AUF DER HEIDE, *Simple, efficient shared memory simulations*, in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1993, pp. 110–119.

[9] P. ERDŐS AND A. RÉNYI, *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl., 5 (1960), pp. 17–61.

[10] J. GIL, Y. MATIAS, AND U. VISHKIN, *Towards a theory of nearly constant time parallel algorithms*, in Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1991, pp. 698–710.

[11] L. A. GOLDBERG, M. JERRUM, T. LEIGHTON, AND S. RAO, *Doubly logarithmic communication algorithms for optical communication parallel computers*, SIAM J. Comput., 26 (1997), pp. 1100–1119.

[12] L. A. GOLDBERG, Y. MATIAS, AND S. RAO, *An optical simulation of shared memory*, in Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1994, pp. 257–267.

[13] M. T. Goodrich, Y. Matias, and U. Vishkin, *Optimal parallel approximation algorithms for prefix sums and integer sorting*, in Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, New York, NY, 1994, pp. 241–250.

[14] T. Hagerup, *The log-star revolution*, in Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 577, Springer-Verlag, Heidelberg, 1992, pp. 259–278.

[15] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1989/90), pp. 305–308.

[16] J. JáJá, *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA, 1992.

[17] A. Karlin and E. Upfal, *Parallel hashing—an efficient implementation of shared memory*, in Proceedings of the 18th Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1986, pp. 160–168.

[18] R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 869–941.

[19] R. M. Karp, M. Luby, and F. Meyer auf der Heide, *Efficient PRAM simulation on a distributed memory machine*, Algorithmica, 16 (1996), pp. 517–542.

[20] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[21] T. Leighton, *Methods for message routing in parallel machines*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1992, pp. 77–96.

[22] P. D. MacKenzie, C. G. Plaxton, and R. Rajaraman, *On contention resolution protocols and associated probabilistic phenomena*, J. ACM, 45 (1998), pp. 324–378.

[23] C. McDiarmid, *On the method of bounded differences*, in Surveys in Combinatorics, J. Siemons, ed., London Math. Soc. Lecture Note Ser. 141, Cambridge University Press, Cambridge, UK, 1989, pp. 148–188.

[24] K. Mehlhorn and U. Vishkin, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories*, Acta Inform., 21 (1984), pp. 339–374.

[25] F. Meyer auf der Heide, C. Scheideler, and V. Stemann, *Exploiting storage redundancy to speed up randomized shared memory simulations*, Theoret. Comput. Sci., 162 (1996), pp. 245–281.

[26] P. Ragde, *The parallel simplicity of compaction and chaining*, J. Algorithms, 14 (1993), pp. 371–380.

[27] A. G. Ranade, *How to emulate shared memory*, J. Comput. System Sci., 42 (1991), pp. 307–326.

[28] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[29] A. Siegel, *On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications*, in Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1989, pp. 20–25.

[30] V. Stemann, *Contention Resolution in Hashing Based Shared Memory Simulations*, Ph.D. thesis, Fachbereich 17, University of Paderborn, Paderborn, Germany, 1995.

[31] E. Upfal, *Efficient schemes for parallel communication*, J. ACM, 31 (1984), pp. 507–517.

[32] E. Upfal and A. Wigderson, *How to share memory in a distributed system*, J. ACM, 34 (1987), pp. 116–127.

[33] L. G. Valiant, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 943–971.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 5, pp. 1740–1759

ALL-PAIRS ALMOST SHORTEST PATHS*

DORIT DOR†, SHAY HALPERIN†, AND URI ZWICK†

Abstract. Let $G = (V, E)$ be an unweighted undirected graph on n vertices. A simple argument shows that computing all distances in G with an additive one-sided error of at most 1 is as hard as Boolean matrix multiplication. Building on recent work of Aingworth et al. [*SIAM J. Comput.*, 28 (1999), pp. 1167–1181], we describe an $\tilde{O}(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$-time algorithm $\mathbf{APASP}_2$ for computing all distances in G with an additive one-sided error of at most 2. Algorithm $\mathbf{APASP}_2$ is simple, easy to implement, and faster than the fastest known matrix-multiplication algorithm. Furthermore, for every even $k > 2$, we describe an $\tilde{O}(\min\{n^{2-2/(k+2)}m^{2/(k+2)}, n^{2+2/(3k-2)}\})$-time algorithm $\mathbf{APASP}_k$ for computing all distances in G with an additive one-sided error of at most k. We also give an $\tilde{O}(n^2)$-time algorithm $\mathbf{APASP}_\infty$ for producing stretch 3 estimated distances in an unweighted and undirected graph on n vertices. No constant stretch factor was previously achieved in $\tilde{O}(n^2)$ time.

We say that a weighted graph $F = (V, E')$ *k-emulates* an unweighted graph $G = (V, E)$ if for every $u, v \in V$ we have $\delta_G(u, v) \le \delta_F(u, v) \le \delta_G(u, v) + k$. We show that every unweighted graph on n vertices has a 2-emulator with $\tilde{O}(n^{3/2})$ edges and a 4-emulator with $\tilde{O}(n^{4/3})$ edges. These results are asymptotically tight.

Finally, we show that any *weighted* undirected graph on n vertices has a 3-spanner with $\tilde{O}(n^{3/2})$ edges and that such a 3-spanner can be built in $\tilde{O}(mn^{1/2})$ time. We also describe an $\tilde{O}(n(m^{2/3}+n))$-time algorithm for estimating all distances in a *weighted* undirected graph on n vertices with a stretch factor of at most 3.

Key words. graph algorithms, shortest paths, approximation algorithms, spanners, emulators

AMS subject classifications. 05C85, 68Q25, 68R10, 05C38

PII. S0097539797327908

1. Introduction.

The all-pairs shortest paths (APSP) problem is one of the most fundamental algorithmic graph problems. The complexity of the fastest known algorithm for solving the problem for weighted directed graphs is $O(mn + n^2 \log n)$, where n and m are the number of vertices and edges in the graph (Johnson [22]; see also [14]). Algorithms for the APSP problem which work on directed graphs with nonnegative edge weights and whose running times are $O(m^*n + n^2 \log n)$, where m^* is the number of edges participating in shortest paths, were obtained by Karger, Koller, and Phillips [23] and by McGeoch [26]. Karger, Koller, and Phillips [23] also obtain an $\Omega(mn)$ lower bound on any *path comparison*-based algorithm for the APSP problem. Takaoka [29], slightly improving a result of Fredman [17], obtained an algorithm for the APSP problem whose running time is $O(n^3((\log\log n)/\log n)^{1/2})$. These algorithms work again on directed graphs with nonnegative edge weights.

The special case of the APSP problem in which the input graph is unweighted is closely related to matrix multiplication. It is fairly easy to see that solving the APSP problem exactly, even on unweighted graphs, is at least as hard as Boolean matrix multiplication. Recent work by Alon, Galil, and Margalit [3], Alon and Naor [4], Galil and Margalit [19], [20], [21], and Seidel [28] have shown that if matrix multiplication can be performed in $O(M(n))$ time, then the APSP problem for unweighted directed

*Received by the editors September 29, 1997; accepted for publication (in revised form) April 9, 1999; published electronically March 17, 2000. A preliminary version of this paper appeared in the Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 452–461.
http://www.siam.org/journals/sicomp/29-5/32790.html

†Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel (ddorit@math.tau.ac.il, hshay@math.tau.ac.il, zwick@math.tau.ac.il).

graphs can be solved in $\tilde{O}(\sqrt{n^3 M(n)})$ time and the APSP problem for unweighted undirected graphs can be solved in $\tilde{O}(M(n))$ time ($\tilde{O}(f)$ means $O(f \operatorname{polylog} n)$). The $\tilde{O}(\sqrt{n^3 M(n)})$ bound for directed graphs was very recently improved by Zwick [33]. The current best upper bound on matrix multiplication is $M(n) = O(n^{2.376})$ (Coppersmith and Winograd [13]).

While the above results are extremely interesting from the theoretical point of view, they are of little use in practice because the fast matrix multiplication algorithms are better than the naive $O(n^3)$-time algorithm only for very large values of n. There is interest therefore in obtaining fast algorithms for the APSP problem that do *not* use fast matrix multiplication. The current best *combinatorial* algorithm for the unweighted APSP problem is an $O(n^3/\log n)$-time algorithm obtained by Feder and Motwani [16] (see also [8]). This offers only a marginal improvement over the naive $O(n^3)$-time algorithm.

Because an algorithm for the APSP problem would yield an algorithm with a similar time bound for Boolean matrix multiplication, obtaining a combinatorial $O(n^{3-\epsilon})$-time algorithm for the APSP problem would be a major breakthrough. Here we obtain such combinatorial algorithms for the all-pairs *almost* shortest paths (APASP) problem.

Awerbuch et al. [7] and Cohen [11] considered the problem of finding stretch t all-pairs paths, where t is some fixed constant and a path is of stretch t if its length is at most t times the distance between its endpoints. Cohen [11], improving the results of Awerbuch et al. [7], obtains, for example, an $\tilde{O}(n^{5/2})$-time algorithm for finding stretch $4+\epsilon$ paths and distances in weighted undirected graphs for any $\epsilon > 0$ (all weights from now on are assumed to be positive). She also exhibits a trade-off between the running time of the algorithm and the obtained stretch factor. For any even t, stretch $t+\epsilon$ paths between all pairs of vertices can be found in $\tilde{O}(n^{2+2/t})$ time. The works of Awerbuch et al. [7] and Cohen [11] are based on the construction of sparse spanners (Awerbuch [6], Peleg and Schäffer [27]). A t-spanner of a graph $G = (V, E)$ is a subgraph $G' = (V, E')$ of G such that for every $u, v \in V$ we have $\delta_{G'}(u,v) \le t \cdot \delta_G(u,v)$, where $\delta_G(u,v)$ is the distance between the vertices u and v in the (possibly weighted) graph G.

A different approach altogether was employed recently by Aingworth et al. [2]. They describe a simple and elegant $\tilde{O}(n^{5/2})$-time algorithm for finding all distances in unweighted and undirected graphs with an *additive* one-sided error of at most 2. They also make the very important observation that the small distances, and not the long distances, are the hardest to approximate. Based on the ideas of Aingworth et al. [2], James B. Orlin (unpublished) obtained an $\tilde{O}(n^{7/3})$-time algorithm for finding all distances with an additive one-sided error of at most 4.

In this work we improve and extend the result of Aingworth et al. [2], and of Orlin, and obtain an $\tilde{O}(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$-time algorithm, called $\mathbf{APASP}_2$, for finding all distances in unweighted and undirected graphs with an additive one-sided error of at most 2. Algorithm $\mathbf{APASP}_2$ is just the first in a sequence of algorithms $\mathbf{APASP}_k$ for even $k \ge 2$ that exhibit a trade-off between running time and accuracy. For any even $k > 2$, algorithm $\mathbf{APASP}_k$ runs in $\tilde{O}(\min\{n^{2-\frac{2}{k+2}} m^{\frac{2}{k+2}}, n^{2+\frac{2}{3k-2}}\})$ time and has a one-sided error of at most k. Algorithm $\mathbf{APASP}_4$, for example, runs in $\tilde{O}(n^{5/3}m^{1/3}, n^{11/5})$ time. All algorithms described in this paper can be easily adapted to find almost shortest paths whose lengths are equal to the estimated distances.

In addition, we show that for any $k \ge 2$, the stretch of the estimates produced by algorithm $\mathbf{APASP}_k$ is at most 3. As k increases, the running time of algorithm

$\mathbf{APASP}_k$ decreases. For $k = \Theta(\log n)$, the running time becomes $\tilde{O}(n^2)$. We let $\mathbf{APASP}_\infty$ be the algorithm $\mathbf{APASP}_k$ with $k = 2\lfloor \log n \rfloor$. Algorithm $\mathbf{APASP}_\infty$ produces stretch 3 distances in unweighted undirected graphs in $\tilde{O}(n^2)$ time. As mentioned, no fixed stretch factor was previously achieved in $\tilde{O}(n^2)$ time.

The errors in the estimated distances produced by $\mathbf{APASP}_k$ are *one sided*. If $\delta(u,v)$ denotes the distance between two vertices u and v in $G = (V,E)$ and $\hat{\delta}(u,v)$ denotes the estimated distance between u and v produced by the algorithm, then for every $u,v \in V$ we have $\delta(u,v) \le \hat{\delta}(u,v) \le \delta(u,v) + k$. The fact that $\delta(u,v) \le \hat{\delta}(u,v)$ follows immediately from the fact that each estimated distance $\hat{\delta}(u,v)$ is the length of a path from u to v in the graph. Note that by subtracting $\lfloor k/2 \rfloor$ from all the estimated distances produced by $\mathbf{APASP}_k$ we can obtain estimates $\tilde{\delta}(u,v)$ satisfying $|\delta(u,v) - \tilde{\delta}(u,v)| \le \lceil k/2 \rceil$ for every $u,v \in V$.

We next introduce the notion of *emulators*. We say that a weighted graph $F = (V,E')$ *k-emulates* an unweighted graph $G = (V,E)$ if for every $u,v \in V$ we have $\delta_G(u,v) \le \delta_F(u,v) \le \delta_G(u,v)+k$. Emulators may be seen as the additive counterparts of spanners. We show that any graph on n vertices has a 2-emulator with $\tilde{O}(n^{3/2})$ edges and a 4-emulator with $\tilde{O}(n^{4/3})$ edges. These can be constructed in $\tilde{O}(n^{5/2})$ and in $\tilde{O}(n^{7/3})$ time, respectively. We are not able to obtain sparser emulators. We are able, however, to construct 6-emulators of size $\tilde{O}(n^{4/3})$ in $\tilde{O}(n^2)$ time. The bounds on the number of edges in 2-emulators and 4-emulators are asymptotically tight. For any $\epsilon > 0$, there are graphs on n vertices that cannot be 2-emulated by graphs with $n^{3/2-\epsilon}$ edges, and there are graphs on n vertices that cannot be 4-emulated by graphs with $n^{4/3-\epsilon}$ edges.

We are also able to obtain some results for *weighted* undirected graphs. We show that any weighted graph on n vertices has a 3-spanner with $\tilde{O}(n^{3/2})$ edges and that such a 3-spanner can be found in $\tilde{O}(mn^{1/2})$ time. Finally, we describe an $\tilde{O}(n(m^{2/3}+n))$-time algorithm for finding stretch 3 distances in a weighted undirected graph on n vertices. Extended and improved results for weighted graphs, including an $\tilde{O}(n^2)$-time algorithm for finding stretch 3 distances and an $\tilde{O}(n^{3/2}m^{1/2})$-time algorithm for finding stretch 2 distances, appear in Cohen and Zwick [12].

2. Preliminaries.

The work of Aingworth et al. [2] is based on the following simple observation: there is a small set of vertices that dominates all the high-degree vertices of a graph. A set of vertices D is said to *dominate* a set U if every vertex in U has a neighbor in D. This observation is also central to our work.

Aingworth et al. [2] show that there is always a set of size $O(n \log n/s)$ that dominates all the vertices of degree at least s in a graph on n vertices and that such a set can be found deterministically in $O(m + ns)$ time, where m is the number of edges in the graph. Here, we show that such a set can be found deterministically in $O(m+n)$ time.

LEMMA 2.1. *Let $G = (V,E)$ be an undirected graph with n vertices and m edges. Let $1 \le s \le n$. A set D of size $O((n \log n)/s)$ that dominates all the vertices of degree at least s in the graph can be found deterministically in $O(m+n)$ time.*

Proof. Before describing a deterministic algorithm we note that picking each vertex of V independently at random with probability $(c \log n)/s$, for some large enough $c > 0$, yields a desired dominating set of size $O((n \log n)/s)$ with high probability.

The deterministic algorithm uses the greedy heuristic. Start with an empty set D. At each stage, add to D a vertex of V that dominates the largest number of vertices of degree at least s that are not already dominated by vertices of D. Lovász [25] and Chvátal [9] show that the greedy heuristic produces a dominating set whose size

```
Algorithm decompose$(G, \langle s_1, s_2, \ldots, s_{k-1}\rangle)$:

input:  (i) An undirected graph $G = (V, E)$.
        (ii) A decreasing sequence $s_1, s_2, \ldots, s_{k-1}$ of degree thresholds.

output: (i) A sequence $E_1, E_2, \ldots, E_k, E^*$ of edge sets.
        (ii) A sequence $D_1, D_2, \ldots, D_k$ of vertex sets.

For $i \leftarrow 2$ to $k$ do $E_i \leftarrow \{e = (u, v) \in E \mid \deg(u) \le s_{i-1}$ or $\deg(v) \le s_{i-1}\}$
For $i \leftarrow 1$ to $k-1$ do $(D_i, E_i^*) \leftarrow$ dominate$(G, s_i)$
$E_1 \leftarrow E$ ; $D_k \leftarrow V$ ; $E^* \leftarrow \bigcup_{1 \le i < k} E_i^*$
```

FIG. 1. *A decomposition algorithm.*

is at most $\ln n$ times the size of the smallest fractional dominating set. A fractional dominating set of size n/s is easily obtained by giving each vertex of the graph a weight of $1/s$. The size of the dominating set produced by the greedy heuristic is therefore at most $(n \ln n)/s$, as required.

It is easy to see that the greedy heuristic can be implemented in $O(m + n)$ time. For each vertex v of G we maintain a linked list of the edges that connect v to vertices of G of degree at least s (in the original graph) that are not yet dominated by the vertices of D. We use a simple array of length n to implement a priority queue that holds the vertices of G. The key of each vertex is the length of its linked list. When a vertex is added to D, we have to update the lists and the keys of some of the vertices in the graph. If a vertex u now becomes dominated, then we have to update the lists and the keys of all its neighbors in G. This takes, however, only $O(\deg(u))$ time, where $\deg(u)$ is the degree of u in G. The total running time is therefore $O(m + n)$. ☐

In what follows we use an algorithm, called **dominate**(G, s), that receives an undirected graph $G = (V, E)$ and a *degree threshold* s. The algorithm outputs a pair (D, E^*), where D is a set of size $O((n \log n)/s)$ that dominates the set of vertices of degree at least s in G. The set $E^* \subseteq E$ is a set of edges of G of size $O(n)$ such that for every vertex $u \in V$ with degree at least s, there is an edge $(u, w) \in E^*$ such that $w \in D$. Once a dominating set D is obtained, the set E^* can be easily obtained by including in it a single edge for each vertex of the graph of degree at least s.

Almost all the algorithms presented in this paper use the decomposition algorithm given in Figure 1. The input to the algorithm is an undirected graph $G = (V, E)$ and a decreasing sequence $s_1 > s_2 > \cdots > s_{k-1}$ of degree thresholds.

Algorithm **decompose** produces a decreasing sequence of edge sets $E_1 \supseteq E_2 \supseteq \cdots \supseteq E_k$, where $E_1 = E$ and E_i for $1 < i \le k$ includes the edges that touch vertices of degree at most s_{i-1}. As the number of vertices of degree at most s_{i-1} is at most n, and as each such vertex contributes at most s_{i-1} edges to E_i, the number of edges in E_i for $1 < i \le k$ is at most ns_{i-1}. Throughout the paper, we let $\deg(v)$ denote the degree of a vertex v.

Algorithm **decompose** also produces a set of dominating sets $D_1, D_2, \ldots, D_k$ and an edge set E^*. The set D_i for $1 \le i < k$ dominates all the vertices in G of degree greater than s_i. In what follows we denote the set of vertices of degree greater than s_i by V_i. The set D_k is simply V, the set of all vertices of the graph. The set

$E^* \subseteq E$ is a set of edges of the graph such that if $u \in V_i$, i.e., if $\deg(u) > s_i$, then there exists a vertex $w \in D_i$ such that $(u, w) \in E^*$. The size of E^* is at most kn. As the running time of **dominate** is $O(m + n)$, the running time of **decompose** is clearly $O(k(m + n))$.

Another ingredient used by our algorithm is the classical Dijkstra's algorithm.

LEMMA 2.2 (Dijkstra's algorithm). *Let $G = (V, E)$ be a weighted directed graph with n vertices and m edges. Let $s \in V$. Dijkstra's algorithm runs in $O(m + n \log n)$ time and finds distances and a tree of shortest paths from s to all the vertices of V. Furthermore, if the weights of the edges are nonnegative integers, then the distances from s to all the vertices that are at distance at most m from s can be found in $O(m + n)$ time.*

Dijkstra's algorithm appeared originally in [15], though the running time of the version described there is $O(n^2)$. For a more modern description of Dijkstra's algorithm see [14]. The running time of $O(m + n \log n)$ is obtained by using *Fibonacci heaps* [18]. If all the weights in the graph G are nonnegative integers and only the distances that are most m are to be computed, then a simple array of length m can be used as a priority queue, resulting in an $O(m + n)$ running time. Thorup [30] showed recently, using more sophisticated techniques, that Dijkstra's algorithm can be implemented to run in $O(m + n)$ time even if the distances of the vertices from s are not bounded, provided that bit operations on the weights are allowed. Our algorithms do not rely on his results.

In all the algorithms described in this paper, except those of section 7, we start with an *unweighted* undirected graph $G = (V, E)$. We then build many auxiliary *weighted* graphs and run Dijkstra's algorithm on each one of them. The weights of the edges in these auxiliary graphs will always be integers in the range $\{1, 2, \ldots, n-1\}$. As we will only be interested in distances that are at most $n - 1$ we can, if we want to, use the simple $O(m + n)$-time implementation of Dijkstra's algorithm. Using the $O(m + n \log n)$-time implementation of Dijkstra's algorithm will not result in a loss of efficiency, however, as all our algorithms are faster than the exact $O(mn)$-time algorithm that simply runs breadth-first search (BFS) from each vertex only when $m \geq n \log n$.

As mentioned above, we can easily solve the APSP problem in unweighted graphs in $O(mn)$ time. Our goal in this paper is to reduce the running time of APSP algorithms to as close to $\tilde{O}(n^2)$ as possible. To achieve this goal we are willing to settle for *almost* shortest paths instead of genuine shortest paths.

Our algorithms involve many runs of Dijkstra's algorithm. Most of these runs are performed, however, on graphs with substantially fewer edges than in the original input graph. It is interesting to note that we use Dijkstra's algorithm, which works on *weighted* graphs, even though the problem that we are trying to solve is unweighted. A typical step in our algorithms is to choose a vertex $u \in V$, choose a set of edges F, and then run Dijkstra's algorithm from u on the graph $H = (V, F)$. The set of edges F is *not* necessarily a subset of the edge set E of the input graph. Furthermore, the set F *varies* from step to step. The weight of an edge $(u, v) \in F$ is taken to be the current best upper bound on the distance between u and v in the input graph G. Bounds obtained in a run of Dijkstra's algorithm are used, therefore, in some of the subsequent runs.

In our algorithms, we use a *symmetric* $n \times n$ matrix, denoted $\{\hat{\delta}(u, v)\}_{u,v}$, to hold the current best upper bounds on distances between all pairs of vertices in the input graph $G = (V, E)$. Initially $\hat{\delta}(u, v) = 1$ if $(u, v) \in E$ and $\hat{\delta}(u, v) = +\infty$ otherwise. (If we assume that the input graph G is connected, we can replace each $+\infty$ by $n-1$.) By **dijkstra**$((V,F),\hat{\delta},u)$ we denote an invocation of Dijkstra's algorithm from the vertex u on the graph (V,F), where the weight of an edge $(u,v)\in F$ is taken to be $\hat{\delta}(u,v)$. As we are interested only in distances that are at most $n-1$, the running time of such a call to Dijkstra's algorithm is $O(|F|+n)$. The edges of F are considered to be *undirected*. As mentioned, an edge of F is not necessarily an edge of E. If $(u,v)\in F$ is an edge of the original graph, then its weight is 1, otherwise, its weight is greater than 1. A call to **dijkstra**$((V,F),\hat{\delta},u)$ updates the row and the column belonging to u in the matrix $\hat{\delta}$ with the distances found during this run if they are smaller than the previous estimates. Note that the matrix $\{\hat{\delta}(u,v)\}_{u,v}$ serves as both input and output of **dijkstra**.

If the graph (V,F) is a subgraph of the input graph $G=(V,E)$, then a call to **dijkstra**$((V,F),\hat{\delta},u)$ amounts to running a BFS on (V,F) from u. When we want to stress this fact, we denote such a call by **bfs**$((V,F),\hat{\delta},u)$. In such a call, the matrix $\{\hat{\delta}(u,v)\}_{u,v}$ is only used for output because the weight of each edge in the input graph is assumed to be 1.

It should be clear from the above discussion that at any time during the run of our algorithms and for any $u,v\in V$ we have $\delta(u,v)\le\hat{\delta}(u,v)$, where $\delta(u,v)$ is the distance between u and v in the input graph G.

In the next section we describe a sequence of algorithms $\mathbf{apasp}_k$ for $k\ge 2$. Algorithm $\mathbf{apasp}_k$ runs in $\tilde{O}(n^{2-1/k}m^{1/k})$ time and produces estimates with surplus $2(k-1)$. In section 4 we describe a sequence of algorithms $\overline{\mathbf{apasp}}_k$ for $k\ge 2$. Algorithm $\overline{\mathbf{apasp}}_k$ runs in $\tilde{O}(n^{2+1/k})$ time and produces estimates with surplus $2(\lfloor k/3\rfloor+1)$. Algorithm $\mathbf{APASP}_k$, mentioned in the abstract and in the introduction, is a combination of a suitable algorithm from the sequence $\mathbf{apasp}_k$ with a suitable algorithm from the sequence $\overline{\mathbf{apasp}}_k$, as explained after the proof of Theorem 4.2 in section 4.

3. Algorithm $\mathbf{apasp}_k$.

Aingworth et al. [2] obtained an $\tilde{O}(n^{5/2})$-time algorithm for approximating all distances in an undirected and unweighted graph with a one-sided additive error of at most 2. We describe a family of algorithms $\mathbf{apasp}_k$, where $k\ge 2$ is an integer. Algorithm $\mathbf{apasp}_k$ runs in $\tilde{O}(n^{2-1/k}m^{1/k})$ time and produces estimated distances with an additive error of at most $2(k-1)$. Note that for $k=2$, we get a running time of $\tilde{O}(n^{3/2}m^{1/2})$ and an additive error of at most 2. At the other extreme, if we take $k=\lceil\log n\rceil$, we get a running time of $\tilde{O}(n^2)$ and an additive error of $O(\log n)$.

A description of the algorithm $\mathbf{apasp}_k$ is given in Figure 2. The algorithm is extremely simple. It starts by decomposing the graph G by invoking algorithm **decompose** of section 2 with the thresholds $s_i=(m/n)^{1-i/k}(\log n)^{i/k}$ for $1\le i<k$. As a result of this decomposition, we get a decreasing sequence of edge sets $E=E_1\supseteq E_2\supseteq\cdots\supseteq E_k$, a sequence of dominating sets $D_1,D_2,\ldots,D_{k-1}$, and $D_k=V$, and an edge set E^*. We will also refer to a decreasing sequence of vertex sets $V=V_1\supseteq V_2\supseteq\cdots\supseteq V_k$, where V_i for $1\le i<k$ is the set of all vertices of G of degree greater than s_i and $V_k=V$. The set E_i for $1\le i\le k$ is then the set of edges that touch a vertex that does *not* belong to V_{i-1}, and the set D_i is a set that dominates V_i through edges of E^*.

Next, algorithm $\mathbf{apasp}_k$ initializes a matrix $\{\hat{\delta}(u,v)\}$ of upper bounds on the distances of the graph by letting $\hat{\delta}(u,v)=1$ if $(u,v)\in E$ and $\hat{\delta}(u,v)=+\infty$ otherwise. Throughout the operation of the algorithm we have $\delta(u,v)\le\hat{\delta}(u,v)$ for every $u,v\in V$,

```
Algorithm apasp_k:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

For $i \leftarrow 1$ to $k - 1$ let $s_i \leftarrow (m/n)^{1-i/k}(\log n)^{i/k}$

$(\langle E_1, E_2, \ldots, E_k, E^*\rangle, \langle D_1, D_2, \ldots, D_k\rangle) \leftarrow$ decompose$(G, \langle s_1, s_2, \ldots, s_{k-1}\rangle)$

For every $u, v \in V$ do
    if $(u, v) \in E$ then $\hat{\delta}(u, v) \leftarrow 1$ else $\hat{\delta}(u, v) \leftarrow +\infty$

For $i \leftarrow 1$ to $k$ do
    For every $u \in D_i$ run dijkstra$(\,(V, E_i \cup E^* \cup (\{u\} \times V))\,, \hat{\delta}, u)$
```

FIG. 2. *An $\tilde{O}(n^{2-1/k}m^{1/k})$-time algorithm for computing surplus $2(k-1)$ distances.*

where $\delta(u, v)$ denotes the distance from u to v in the graph G.

Finally, the main part of **apasp**$_k$ is composed of successive calls to Dijkstra's algorithm. At first, Dijkstra's algorithm is run from every vertex of the dominating set D_1, then it is run from every vertex of the dominating set D_2, and so on. Most of these runs are performed, however, on graphs that are much sparser than the original input graph G. The run of Dijkstra's algorithm from a vertex $u \in D_i$ is performed on a graph $G_i(u) = (V, E_i(u))$, where $E_i(u) = E_i \cup E^* \cup (\{u\} \times V)$. The edges of $E_i \cup E^*$ are original edges of the graph and they are attached a weight of 1. The edges of $\{u\} \times V$ are not necessarily edges of the input graph. The weight attached to such an edge (u, v) is $\hat{\delta}(u, v)$, the best upper bound available at the time on the distance from u to v in the graph. Note that a slightly different graph is used in each invocation of Dijkstra's algorithm, even from vertices that belong to the same dominating set.

THEOREM 3.1. *For every $k \geq 2$, the running time of algorithm* **apasp**$_k$ *is $O(kn^{2-1/k}m^{1/k}(\log n)^{1-1/k})$, where n is the number of vertices and m is the number of edges in the input graph $G = (V, E)$, and for every $u, v \in V$ we have $\delta(u, v) \leq \hat{\delta}(u, v) \leq \delta(u, v) + 2(k - 1)$.*

Proof. We start with the complexity analysis of the algorithm. The call to **decompose** takes $O(kn^2)$ time. The initialization of $\hat{\delta}(u, v)$ takes $O(n^2)$ time. Most of the running time of the algorithm is spent in the calls to Dijkstra's algorithm. It is not difficult to see that the total running time of these calls is

$$O\left(\sum_{i=1}^{k} |D_i| \cdot (|E_i| + n)\right) = O\left(\frac{n \log n}{s_1} \cdot m + \sum_{i=2}^{k-2} \frac{n \log n}{s_i} \cdot ns_{i-1} + n \cdot ns_{k-1}\right) .$$

This follows from the fact that $|D_i| = O(n \log n/s_i)$ for $1 \leq i < k$, $|D_k| = |V| = n$, and from the fact that $|E_1| = |E| = m$ and $|E_i| \leq ns_{i-1}$ for $2 \leq i \leq k$. With the choice $s_i = (m/n)^{1-i/k}(\log n)^{i/k}$, each one of the k terms in this expression is equal to $n^{2-1/k}m^{1/k}(\log n)^{1-1/k}$ and the total running time of the algorithm is $O(kn^{2-1/k}m^{1/k}(\log n)^{1-1/k})$, as promised. The running time of the algorithm can also be expressed as $O(kn^2 \log n \cdot ((m \log n)/n)^{1/k})$.

We now examine the accuracy of the algorithm. For every $1 \leq i \leq k$ and $u, v \in V$, let $\delta_i(u, v)$ be the value of $\hat{\delta}(u, v)$ after running **dijkstra** from all the vertices of

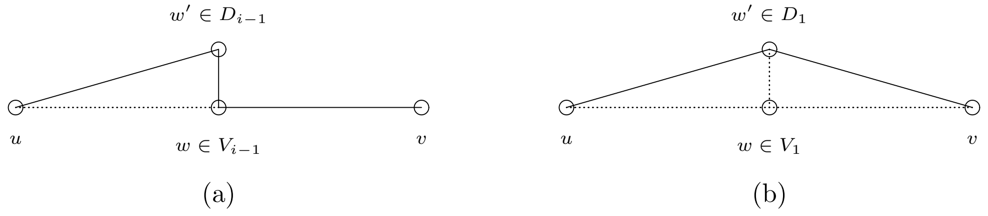


FIG. 3. (a) *Case* 2 *in the proof of Theorem* 3.1. (b) *Case* 1 *in the proof of Theorem* 4.1.

D_i. We now prove by induction on i that if $u \in D_i$ and $v \in V$, then $\delta(u,v) \le \delta_i(u,v) \le \delta(u,v) + 2(i-1)$. Recall that $D_k = V$. For every $u, v \in V$, we get that $\hat{\delta}(u,v) = \delta_k(u,v) \le \delta(u,v) + 2(k-1)$, as required.

Let $G_i(u) = (V, E_i(u))$ be the graph on which Dijkstra's algorithm is run from $u \in D_i$. For $i = 1$, the claim is clear, because for every $u \in D_1$ we have $G_1(u) = G$, and therefore $\delta_1(u,v) = \delta(u,v)$ for every $u \in D_1$ and $v \in V$. Suppose therefore that $i > 1$ and that the claim holds for $i-1$. Let $u \in D_i$ and let $v \in V$. Consider a shortest path p from u to v. We consider two cases.

Case 1. All the edges of p belong to E_i.

Because all the edges of p belong to E_i, they also belong to the graph $G_i(u)$ on which Dijkstra's algorithm is run from u. We get, therefore, that $\delta_i(u,v) = \delta(u,v)$ and we are done.

Case 2. Not all the edges of p belong to E_i.

Because the edges of E_i are edges that touch vertices that are not in V_{i-1}, there must be a vertex from V_{i-1} on the path p. (There must, in fact, be two consecutive vertices from V_{i-1} on the path p but we do not use this fact here.) Let w be the last vertex from V_{i-1} on the path p. Let p' be the subpath of p that connects w and v. Because all the vertices on p', except w, do not belong to V_{i-1}, all the edges of p' belong to E_i. Let $w' \in D_{i-1}$ be such that $(w, w') \in E^*$. The graph $G_i(u)$ contains the edge (w, w') and a weighted edge (u, w') whose weight is $\delta_{i-1}(u, w') = \delta_{i-1}(w', u)$ (see Figure 3(a)). By the induction hypothesis, $\delta_{i-1}(w', u) \le \delta(w', u) + 2(i-2) \le \delta(u,w) + (2i-3)$. As a consequence, we get that

$$\begin{aligned}\delta_i(u,v) &\le \delta_{i-1}(u,w') + \delta(w',w) + \delta(w,v)\\ &\le (\delta(u,w) + (2i-3)) + 1 + \delta(w,v)\\ &\le \delta(u,v) + 2(i-1)\ .\end{aligned}$$

This completes the proof of the theorem. ∎

Although the additive error of the estimated distances produced by $\mathbf{apasp}_k$ increases as k increases, we can show that the *stretch* of the estimated distances produced is always at most 3.

THEOREM 3.2. *For every* $2 \le k = O(\log n)$, *the running time of algorithm* $\mathbf{apasp}_k$ *is* $O(kn^{2-1/k}m^{1/k}(\log n)^{1-1/k})$, *and for every* $u, v \in V$ *we have* $\delta(u,v) \le \hat{\delta}(u,v) \le \min\{\,\delta(u,v) + 2(k-1)\,,\, 3\delta(u,v) - 2\,\}$.

Proof. We only have to show that for every $u, v \in V$ we have $\hat{\delta}(u,v) \le 3\delta(u,v) - 2$. All the rest follows from Theorem 3.1. We start with the following lemma.

LEMMA 3.3. *Let* p *be a path of length* ℓ *between* u *and* v, *where* $u \in D_i$. *If* $\delta_i(u,v) > \ell + 2r$, *then there are at least* $r+1$ *edges on the path* p *that do not belong to* $E_i \cup E^*$.

We prove this Lemma by induction on i. If $i = 1$, then $\delta_1(u, v) = \delta(u, v) \leq \ell$ and there is nothing to prove.

Suppose that the lemma is true for every $j < i$. Let p be a path of length ℓ between $u \in D_i$ and v, and suppose that $\delta_i(u, v) > \ell + 2r$. It follows that not all the edges of p belong to $E_i(u)$. This completes the proof if $r = 0$. Suppose therefore that $r > 0$. Let $e = (w_1, w_2)$ be the last edge on p that does not belong to $E_i(u)$. Let $1 \leq j < i$ be such that $w_2 \in V_j \setminus V_{j-1}$ (where $V_0 = \phi$). Since $w_2 \notin V_{j-1}$, we get that $e \in E_j$. Let $w_2' \in D_j$ such that $(w_2, w_2') \in E^*$. Let ℓ_1 and ℓ_2 be the distances from u to w_2 and from w_2 to v on p. Because $\delta_i(u, v) > \ell + 2r$ on the one hand, and $\delta_i(u, v) \leq \delta_j(w_2', u) + 1 + \ell_2$ on the other, we get that $\delta_j(w_2', u) > (\ell_1 + 1) + 2(r - 1)$. Let p' be the path from w_2' to u composed of the edge (w_2', w_2) and the portion of p from w_2 to u. According to the induction hypothesis, there must be at least r edges on p' that do not belong to $E_j \cup E^*$. Since $(w_2', w_2) \in E^*$, we get that all these r edges must be edges of p. Because $e \in E_j$, we get that e is not one of these edges. Since $e \notin E_i \cup E^*$, we get that there must be at least $r + 1$ edges on p that do not belong to $E_i \cup E^*$, as required. This completes the proof of the lemma.

Proof of Theorem 3.2. A path p of length ℓ may contain at most ℓ edges that do not belong to $E_k \cup E^*$. Thus, if p is a path of length ℓ connecting $u \in D_k$ and v, then it follows from Lemma 3.3 that $\hat{\delta}(u, v) \leq 3\ell$. Note that $D_k = V$. This is almost what we wanted to show, but not quite.

Consider again a path p of length ℓ between two vertices u and v. If at least one of the edges of p belongs to $E_k \cup E^*$, then Lemma 3.3 implies that $\hat{\delta}(u, v) \leq 3\ell - 2$. If the path p is of length one, i.e., $\ell = 1$, then $\hat{\delta}(u, v) = \delta(u, v) = 1 = 3\ell - 2$. Otherwise, we know that $e_1 = (u, u')$ and $e_2 = (v', v)$, the first and last edges on the path p, do not belong to $E_k \cup E^*$. As a consequence, we get that $u, v \in V_{k-1}$. Let $1 \leq j_1 < k$ and $1 \leq j_2 < k$ be such that $u \in V_{j_1} \setminus V_{j_1-1}$ and $v \in V_{j_2} \setminus V_{j_2-1}$ (where again $V_0 = \phi$). Assume, without loss of generality, that $j_2 \leq j_1$. For brevity, let $j = j_2$. Because $u, v \notin V_{j-1}$, we get that $e_1, e_2 \in E_j$. Let $w \in D_j$ such that $(v, w) \in E^*$. Let p' be the path from w to u composed of the edge (w, v) and the path p in reversed order. Since the number of edges on the path p' that do not belong to $E_j \cup E^*$ is at most $\ell - 2$, we get, by Lemma 3.3, that $\delta_j(w, u) \leq (\ell + 1) + 2(\ell - 2) = 3\ell - 3$. Thus, $\delta_k(u, v) \leq \delta_j(w, u) + 1 \leq 3\ell - 2$, as required. ☐

As an immediate corollary, we get that the estimated distances produced by $\mathbf{apasp}_k$ satisfy $\delta(u, v) \leq \hat{\delta}(u, v) \leq (3 - \frac{2}{k})\delta(u, v)$ for every $u, v \in V$. For $k = 2$, the distances are stretched by a factor of at most 2. For $k = 3$, the distances are stretched by a factor of at most 7/3. For any k, the distances are stretched by a factor of at most 3.

By taking $k = \Theta(\log n)$, we get an $\tilde{O}(n^2)$-time algorithm for finding stretch 3 approximate distances. An extension of this algorithm for weighted graphs is presented in [12].

4. Algorithm $\overline{\mathbf{apasp}}_{\boldsymbol{k}}$. In this section we present another family of algorithms, called $\overline{\mathbf{apasp}}_k$, for estimating all the distances in an unweighted and undirected graph with a small additive error. Algorithms from the family $\overline{\mathbf{apasp}}_k$ perform better than algorithms from the family $\mathbf{apasp}_k$ when the input graphs are sufficiently dense. For example, algorithms $\mathbf{apasp}_2$ and $\overline{\mathbf{apasp}}_3$ both produce estimated distances with an additive error of at most 2. Algorithm $\mathbf{apasp}_2$ runs in time $\tilde{O}(n^{3/2}m^{1/2})$ while algorithm $\overline{\mathbf{apasp}}_3$ runs in time $\tilde{O}(n^{7/3})$. Algorithm $\overline{\mathbf{apasp}}_3$ is therefore faster when $m \geq n^{5/3}$. We will show later that algorithm $\overline{\mathbf{apasp}}_k$ for $k > 3$ runs in $\tilde{O}(n^{2+1/k})$ time and produces estimates with an additive error of at most $2(\lfloor k/3 \rfloor + 1)$. Thus, algorithm

```
Algorithm $\overline{\mathbf{apasp}}_k$:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

For $i \leftarrow 1$ to $k - 1$ let $s_i \leftarrow n^{1-i/k}(\log n)^{i/k}$

$(\langle E_1, E_2, \ldots, E_k, E^*\rangle, \langle D_1, D_2, \ldots, D_k\rangle) \leftarrow \mathbf{decompose}(G, \langle s_1, s_2, \ldots, s_{k-1}\rangle)$

For every $u, v \in V$ do
    if $(u, v) \in E$ then $\hat{\delta}(u, v) \leftarrow 1$ else $\hat{\delta}(u, v) \leftarrow +\infty$

For $i \leftarrow 1$ to $k$ do
    For every $u \in D_i$ do
        run $\mathbf{dijkstra}(\,(V, E_i \cup E^* \cup (\{u\} \times V) \cup (\cup_{i+j_1+j_2 \le 2k+1} D_{j_1} \times D_{j_2}))\,, \hat{\delta}, u)$
```

FIG. 4. *An $\tilde{O}(n^{2+1/k})$-time algorithm for computing surplus $O(k)$ distances.*

$\overline{\mathbf{apasp}}_5$, for example, runs in $\tilde{O}(n^{11/5})$ time and provides estimated distances with a surplus of at most 4.

Algorithm $\overline{\mathbf{apasp}}_k$, given in Figure 4, is similar to algorithm $\mathbf{apasp}_k$. There are two main differences, however. The first is that the sequence of degree thresholds is this time $s_i = n^{1-i/k}(\log n)^{i/k}$ instead of $s_i = (m/n)^{1-i/k}(\log n)^{i/k}$. The second and more important one is that the edge set $E_i(u)$ of the graph $G_i(u) = (V, E_i(u))$ on which Dijkstra's algorithm is run from u during the ith iteration of the algorithm is now "reacher" than the corresponding set $E_i(u)$ used in $\mathbf{apasp}_k$. The set $E_i(u)$ is now composed of the following components:

$$E_i(u) \;=\; E_i \;\cup\; E^* \;\cup\; (\{u\} \times V) \;\cup\; \left(\bigcup_{i+j_1+j_2 \le 2k+1} D_{j_1} \times D_{j_2} \right).$$

Thus, in addition to edges used by $\mathbf{apasp}_k$, $\overline{\mathbf{apasp}}_k$ also uses edges from the set $D_{j_1} \times D_{j_2}$ for every $1 \le j_1, j_2 \le k$ such that $i + j_1 + j_2 \le 2k + 1$. Note, in particular, that for every $1 \le i \le k$ and every $u \in D_i$, we have

$$(V \times D_1) \cup (D_1 \times V) \subseteq E_i(u)\ .$$

(Recall that $D_k = V$.) As a further example, note that in $\overline{\mathbf{apasp}}_3$ we have

$$E_3(u) \;=\; E_i \;\cup\; E^* \;\cup\; (\{u\} \times V) \;\cup\; (D_1 \times D_3) \;\cup\; (D_2 \times D_2) \;\cup\; (D_3 \times D_1)\ .$$

We begin with the relatively simple analysis of $\overline{\mathbf{apasp}}_3$. The more complicated analysis of $\overline{\mathbf{apasp}}_k$ for $k > 3$ will then follow.

THEOREM 4.1. *Algorithm $\overline{\mathbf{apasp}}_3$ runs in $O(n^{7/3} \log^{2/3} n)$ time, where n is the number of vertices in the input graph $G = (V, E)$, and for every $u, v \in V$ we have $\delta(u, v) \le \hat{\delta}(u, v) \le \delta(u, v) + 2$.*

Proof. The fact that algorithm $\overline{\mathbf{apasp}}_3$ runs in $O(n^{7/3} \log^{2/3} n)$ time will follow from the analysis in the proof of Theorem 4.2. We show here that $\delta(u, v) \le \hat{\delta}(u, v) \le \delta(u, v) + 2$ for every $u, v \in V$.

It is again clear that $\delta(u,v) \le \hat{\delta}(u,v)$ for every $u,v \in V$. It remains to show therefore that $\hat{\delta}(u,v) \le \delta(u,v)+2$ for every $u,v \in V$. Let u and v be two vertices in G. Let p be a shortest path from u to v in G. We consider the following three cases.

Case 1. The shortest path p contains a vertex w from V_1.

Let $w' \in D_1$ such that $(w,w') \in E$ (see Figure 3(b)). (Note that we do not require $(w,w') \in E^*$, although we could.) Since $(V \times D_1) \cup (D_1 \times V) \subseteq E_3(u)$, the edges (u,w') and (w',v) belong to the graph on which Dijkstra's algorithm is run from u. The weights of these edges are $\delta(u,w')$ and $\delta(w',v)$, the distances found by running Dijkstra's algorithm on the original graph G from $w' \in D_1$. Note that $\delta(u,w') \le \delta(u,w)+1$ and $\delta(w',v) \le 1+\delta(w,v)$. By running Dijkstra's algorithm from u, we find therefore that

$$\hat{\delta}(u,v) \le \delta(u,w') + \delta(w',v) \le \delta(u,v)+2.$$

Case 2. The shortest path p contains vertices from V_2 but not from V_1.

This case is very similar to Case 1 in the proof of Theorem 3.1. Let w be the *last* vertex on the path that belongs to V_2. All the edges on the path from w to v touch vertices that do not belong to V_2 and therefore belong to the set E_3 and to the graph $G_3(u)$ on which Dijkstra's algorithm is run from u. Let $w' \in D_2$ be such that $(w,w') \in E^*$. The graph $G_3(u)$ contains weighted edges connecting u to all the vertices of V. It contains in particular a weighted edge (u,w'). The weight of this edge is the distance $\delta_2(u,w')$ between u and w' in the graph $G_2(w') = (V, E_2(w'))$, found by running Dijkstra's algorithm from $w' \in D_2$ on $G_2(w')$. Because all the edges on the path from u to w as well as (w,w') belong to $E_2(w')$, we get that $\delta_2(u,w') \le \delta(u,w)+1$. By running Dijkstra's algorithm from u we find therefore that

$$\hat{\delta}(u,v) \le \delta_2(u,w') + \delta(w',w) + \delta(w,v) \le \delta(u,v)+2.$$

Case 3. The shortest path p does not contain any vertex from V_2.

This shortest path is then contained in (V,E_3), and therefore $\hat{\delta}(u,v) = \delta(u,v)$. ◻

As mentioned above, the edge set $E_3(u)$ for every $u \in V$ contains the edge set $D_2 \times D_2$. A closer look at the proof of Theorem 4.1 reveals that we have not used this fact in the proof. The set $D_2 \times D_2$ plays, however, a central role in the proof of Theorem 6.3.

We now turn to the more complicated analysis of algorithm $\overline{\mathbf{apasp}}_k$ for $k \ge 3$. (The analysis below applies also for $k=2$, but for $k=2$, the bound in Theorem 4.1 is tighter.)

THEOREM 4.2. *For every $k \ge 2$, the running time of algorithm $\overline{\mathbf{apasp}}_k$ is $O(k^2 n^{2+1/k}(\log n)^{1-1/k})$, where n is the number of vertices in the input graph $G = (V,E)$, and for every $u,v \in V$ we have $\delta(u,v) \le \hat{\delta}(u,v) \le \delta(u,v) + 2(\lfloor k/3 \rfloor + 1)$.*

Proof. We start with the complexity analysis. It is easy to see that all the preparatory steps of the algorithm take $\tilde{O}(n^2)$ time. For every $1 \le i \le k$ and for every $u \in D_i$, we then run Dijkstra's algorithm from u on the graph $G_i(u) = (V, E_i(u))$. It is easy to verify that $|E_1| = O(n^2)$, $|E_i(u)| = O(kns_{i-1})$ for $2 \le i \le k$, and that $|D_i| = O(n \log n / s_i)$ for $1 \le i < k$, and $|D_k| = n$. A simple calculation, similar to the one in the beginning of the proof of Theorem 3.1, shows that the total running time of $\overline{\mathbf{apasp}}_k$ is $O(k^2 n^{2+1/k}(\log n)^{1-1/k})$. The running time of the algorithm can also be expressed as $O((kn)^2 \log n \cdot (n/\log n)^{1/k})$.

We now study the accuracy of the distance estimates produced by $\overline{\mathbf{apasp}}_k$. For every $1 \le i \le k$, we let $\delta_i(u,v)$ be the value of $\hat{\delta}(u,v)$ after running **dijkstra** from all the vertices of D_i. We now define recursively the sequence

$$e_{i_1,j,i_2} = \left\{ \begin{array}{cl} 0 & \text{if } \; i_1 \le j \\ 2 & \text{if } \; i_1 + j + i_2 \le 2k+1 \\ e_{i_1-1,j,i_1} + 2 & \text{otherwise} \end{array} \right\}$$

and prove by induction the following claim.

Claim. If $u \in D_{i_1}$ and $v \in D_{i_2}$ are connected by a path p of length ℓ in which the vertex of highest degree belongs to V_j, then $\delta_{i_1}(u,v) \le \ell + e_{i_1,j,i_2}$.

To prove the claim we use essentially the same arguments used in the proofs of Theorems 3.1 and 4.1.

If $i_1 \le j$, then all the edges on the path p are contained in $E_{i_1}(u)$ and therefore $\delta_{i_1}(u,v) \le \ell$, as required.

Suppose now that $i_1 + j + i_2 \le 2k+1$ and that $j < i_1$. This means that $D_j \times D_{i_2} \subseteq E_{i_1}(u)$. Let w be a vertex on p that belongs to V_j. Let ℓ_1 be the distance from u to w on the path p. Let ℓ_2 be the distance from w to v on the path p. Clearly $\ell = \ell_1 + \ell_2$. Let $w' \in D_j$ such that $(w,w') \in E^*$. It is easy to see that $\delta_j(u,w') \le \ell_1 + 1$ and $\delta_j(w',v) \le \ell_2 + 1$. Because $(u,w'),(w',v) \in E_{i_1}(u)$, we get that $\delta_{i_1}(u,v) \le \ell + 2$, as required.

Finally, suppose that $i_1 + j + i_2 > 2k+1$ and that $j < i_1$. Since $j < i_1$, we get that $V_j \subseteq V_{i_1-1}$. Let w be the *last* vertex on p that belongs to V_{i_1-1}. Let $w' \in D_{i_1-1}$ such that $(w,w') \in E^*$. Let ℓ_1 and ℓ_2 be, as before, the distances from u to w and from w to v on p. Consider the path p' composed of the edge (w',w) and the portion of the path p from w to u. The path p' starts at $w' \in D_{i_1-1}$ and ends at $u \in D_{i_1}$, and the vertex of highest degree on it belongs to V_j. The length of p' is $\ell_1 + 1$. By applying the claim inductively to p', we get that $\delta_{i_1-1}(w',u) \le \ell_1 + 1 + e_{i_1-1,j,i_1}$. Because w is the last vertex from v_{i_1-1} on p, all the edges on the portion of p from w to v belong to $E_{i_1}(u)$. The set $E_{i_1}(u)$ also contains the edge $(w',w) \in E^*$ and a weighted edge (u,w') of weight $\delta_{i_1-1}(w',u) = \delta_{i_1-1}(u,w')$. After running **dijkstra** from u on $(V, E_i(u))$, we get $\delta_{i_1}(u,v) \le (\ell_1 + 1 + e_{i_1-1,j,i_1}) + (1 + \ell_2) = \ell + (e_{i_1-1,j,i_1} + 2) = \ell + e_{i_1,j,i_2}$, as required. This completes the proof of the claim.

Now let $e_{i,j} = e_{i,j,i+1}$. It is easy to verify that

$$e_{i,j} = \left\{ \begin{array}{cl} 0 & \text{if } \; i \le j \\ 2 & \text{if } \; 2i + j \le 2k \\ e_{i-1,j} + 2 & \text{otherwise} \end{array} \right\} .$$

It is not difficult to unwind this recursion and get that

$$e_{i,j} = \left\{ \begin{array}{cl} 0 & \text{if } \; i \le j \\ 2 & \text{if } \; 2i + j \le 2k \\ \min\{\, 2(i-j)\, ,\, 2(i - k + \lceil \frac{j}{2} \rceil + 1)\,\} & \text{otherwise} \end{array} \right\} .$$

Finally, we get

$$e_{k,j,k} = \left\{ \begin{array}{cl} 0 & \text{if } \; j = k \\ 2 & \text{if } \; j = 1 \\ e_{k-1,j} + 2 & \text{otherwise} \end{array} \right\} = \left\{ \begin{array}{cl} 0 & \text{if } \; j = k \\ 2 & \text{if } \; j = 1 \\ \min\{\, 2(k-j)\, ,\, 2\lceil \frac{j}{2} \rceil + 2\,\} & \text{otherwise} \end{array} \right\} .$$

It is now not difficult to verify that for every $1 \le j \le k$, we have $e_{k,j,k} \le 2(\lfloor k/3 \rfloor + 1)$. Because $D_k = V$, this completes the proof of the theorem. $\square$

To get an additive error of at most k, where $k > 2$ is even, we can use either algorithm $\mathbf{apasp}_{k/2+1}$, whose running time is $\tilde{O}(n^{2-2/(k+2)}m^{2/(k+2)})$, or algorithm $\overline{\mathbf{apasp}}_{(3k-2)/2}$, whose running time is $\tilde{O}(n^{2+2/(3k-2)})$. The combination of these two algorithms is the algorithm $\mathbf{APASP}_k$ mentioned in the abstract.

We can easily get a randomized version of algorithm $\overline{\mathbf{apasp}}_3$ which has the property that *all* reported distances greater than $n^{2/3}$ are, with high probability, correct. Similarly, we can get a randomized version of $\mathbf{apasp}_2$ for which all reported distances greater than $n^{3/2}/m^{1/2}$ are, with high probability, correct. We use the following simple observation (a similar idea is used by Ullman and Yannakakis [31]).

THEOREM 4.3. *Let $G = (V, E)$ be a weighted directed graph with n vertices and m edges. Let $1 \le s \le n$. There is an $O(n^3 \log n/s)$-time randomized algorithm that finds, with high probability, the exact distance between any pair of vertices connected by a shortest path that uses at least s edges.*

Proof. Let D be a random set of vertices obtained by picking each vertex independently with probability $(c \log n)/s$ for some large enough $c > 0$. The expected size of D is $O(n \log n/s)$. Run Dijkstra's algorithm from each vertex of D in both G and the graph obtained from G by reversing the edges. The complexity of this step is $O(nm \log n/s)$. For every $u \in D$ and $v \in V$, we now know $\delta(u, v)$ and $\delta(v, u)$ exactly. For every pair of vertices $u, v \in V$, let $\hat{\delta}(u, v) = \min_{w \in D}\{\delta(u, w) + \delta(w, v)\}$. The complexity of this step is $O(n^3 \log n/s)$. It is easy to see that $\hat{\delta}(u, v) = \delta(u, v)$ if and only if there is a shortest path between u and v that passes through a vertex of D. If there is a shortest path between u and v of length s, then with high probability, at least one of the vertices on the path will belong to D. Since there are $O(n^2)$ pairs of vertices connected by shortest paths that use at most s edges, and since we can focus on one such path for each pair, we get that, with high probability, each one of these $O(n^2)$ paths will pass through a vertex of D, and the exact distances between all these pairs will be found. ∎

It follows that if the set D_1 in algorithm $\overline{\mathbf{apasp}}_3$ is chosen at random by picking each element with probability $cn^{-2/3} \log n$, then, with high probability, if $\delta(u, v) \ge n^{2/3}$, then $\hat{\delta}(u, v)$ is the exact distance between u and v. Long distances are therefore easier to compute.

5. Boolean matrix multiplication.

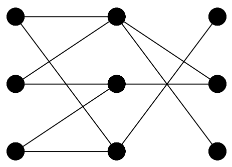
Let A and B be two Boolean $n \times n$ matrices. Construct a graph $G_{A,B} = (V, E)$ with

$$V = \{u_1, \dots, u_n\} \cup \{v_1, \dots, v_n\} \cup \{w_1, \dots, w_n\} ,$$
$$E = \{(u_i, v_k) \mid a_{ik} = 1\} \cup \{(v_k, w_j) \mid b_{kj} = 1\} .$$

The graph corresponding to two 3×3 matrices is depicted in Figure 5. Let $C = A \times B$ (Boolean matrix multiplication). Clearly, $c_{ij} = 1$ if and only if $\delta_G(u_i, w_j) = 2$. Furthermore, because the graph $G_{A,B}$ is bipartite, $c_{ij} = 1$ if and only if $\delta_G(u_i, w_j) \le 3$. As a consequence we get the following result.

THEOREM 5.1. *If all the distances in an undirected n-vertex graph can be approximated with a one-sided additive error of at most 1 in $O(A(n))$ time, then Boolean matrix multiplication can also be performed in $O(A(n))$ time.*

By adding a disjoint path of length $k - 2$ ending at each u_i, we get that, for any *fixed* $k \ge 2$, distinguishing between distance k and $k + 2$ in graphs with n vertices, i.e., deciding for each pair of vertices u, v whether $\delta(u, v) \le k$ or $\delta(u, v) \ge k + 2$ (if $\delta(u, v) = k + 1$, then either decision is fine), is at least as hard as multiplying two Boolean matrices of size $n \times n$. Note, in contrast, that if $k \ge n^{2/3}$, then by



$$\begin{pmatrix}1 & 0 & 1\\ 1 & 1 & 0\\ 0 & 1 & 1\end{pmatrix} \times \begin{pmatrix}0 & 1 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{pmatrix}$$

FIG. 5. *Boolean matrix multiplication.*

Theorem 4.3 we can distinguish, with high probability, between distances k and $k+2$ in graphs with n vertices in $\tilde{O}(n^{7/3})$ time, i.e., faster than the fastest known matrix multiplication algorithm.

Similarly, because any approximation algorithm that finds approximated distances of stretch strictly less than 2 can distinguish between distance 2 and distance 4, getting approximate distances of stretch *less* than 2 between all pairs of vertices is also as hard as Boolean matrix multiplication.

By turning the graph $G_{A,B}$ into a directed graph, where edges are directed to the right, we get that $c_{ij} = 1$ if and only if $\delta_G(u_i, w_j) < \infty$. Approximating distances in *directed* graphs to within any *multiplicative* factor, not necessarily bounded, is therefore as hard as Boolean matrix multiplication. Note that such an approximation is equivalent to the computation of the *transitive closure* of the graph. It is proved in [1] (see Theorems 5.6 and 5.7) that the computation of the transitive closure of a directed graph is equivalent to Boolean matrix multiplication. We end this section with another simple observation.

THEOREM 5.2. *If two $n \times n$ Boolean matrices could be multiplied in $O(M(n))$ time, then for any fixed $\epsilon > 0$, all the distances in an unweighted directed graph on n vertices can be estimated with stretch $1+\epsilon$ in $\tilde{O}(M(n))$ time.*

Proof. Let $G = (V, E)$ be an unweighted directed graph on n vertices. Let A be the adjacency matrix of the graph with self-loops added to all the vertices. Note that $\delta_G(u, v) \le d$ if and only if $(A^d)_{u,v} = 1$. Let $r_i = \lfloor (1+\epsilon)^i \rfloor$ for $1 \le i \le k = \lceil \log_{1+\epsilon} n \rceil$. We compute A^{r_i} for $1 \le i \le k$. We let $\hat{\delta}(u, v) = r_{i+1}$ if and only if $(A^{r_i})_{u,v} = 0$ but $(A^{r_{i+1}})_{u,v} = 1$. It is easy to verify that for every $u, v \in V$ we have $\delta(u, v) \le \hat{\delta}(u, v) \le (1+\epsilon) \cdot \delta(u, v)$, and that the running time of this algorithm is $\tilde{O}(M(n))$. $\square$

6. Distance emulators.

6. Distance emulators. Closely related to the algorithms of sections 3 and 4 is the notion of emulators.

DEFINITION 6.1 (emulators). *Let $G = (V, E)$ be an unweighted undirected graph. A weighted graph $H = (V, F)$ is said to be a k-*emulator *of G if and only if for every $u, v \in V$ we have $\delta_G(u, v) \le \delta_H(u, v) \le \delta_G(u, v) + k$.*

There is a fundamental difference, however, between emulators and the auxiliary graphs used in the algorithms of sections 3 and 4. There, we constructed for each vertex u an auxiliary graph $G_k(u)$ that supplied good approximations to the distances from u to all the vertices of the graph. Here we want a *single* graph that will supply good approximations of *all* distances. Constructing a sparse k-emulator is therefore harder than computing surplus k distances.

The definition of k-emulators is related to the definition of *k-spanners* (Awerbuch [6], Peleg and Schäffer [27]). Let $G = (V, E)$ be a weighted undirected graph. A subgraph $G' = (V, E')$ of G is said to be a k-spanner of G if and only if for every

$u, v \in V$ we have $\delta_{G'}(u,v) \leq k \cdot \delta_G(u,v)$. Since G' is a subgraph of G, we always have $\delta_G(u,v) \leq \delta_{G'}(u,v)$. This definition differs from the definition of emulators in three respects. We require additive error, not multiplicative error. We do not insist on getting a subgraph of the original graph and we allow weighted edges. Althöfer et al. [5] also consider *Steiner spanners* in which vertices and edges may be added to the graph. Steiner spanners are more closely related to emulators. Liestman and Shermer [24] consider *additive spanners*. They are able, however, to obtain sparse additive spanners only for specific graphs such as pyramids, grids, and hypercubes. Additive spanners, unlike emulators, must be subgraphs of the original graph. Emulators may be described as weighted additive Steiner spanners. The definition of k-emulators is also related to the definition of *hop sets* (Cohen [10]).

Implicit in the work of Aingworth et al. [2] is an $\tilde{O}(n^{5/2})$-time algorithm for constructing 2-emulators with $\tilde{O}(n^{3/2})$ edges. We can get the following slightly stronger result.

THEOREM 6.2. *Every unweighted undirected graph $G = (V, E)$ on n vertices can be 2-emulated by a* subgraph *$G' = (V, F)$ with $O(n^{3/2}(\log n)^{1/2})$ edges. Such a subgraph can be constructed in $O(n^2 \log^2 n)$ time.*

Proof. We start by proving the existence of such an emulator. Split the vertices of G into two classes: $V_1 = \{v \in V \mid \deg(v) \geq (n \log n)^{1/2}\}$ and $V_2 = \{v \in V \mid \deg(v) < (n \log n)^{1/2}\}$. Find a set D of size $O((n \log n)^{1/2})$ that dominates V_1 and a set E^* of at most n edges such that for every $u \in V_1$ there exists $v \in D$ such that $(u, v) \in E^*$. From every $v \in D$ perform a BFS and find its distances to all the vertices of the graph. A 2-emulator of G of size $O(n^{3/2}(\log n)^{1/2})$ is then obtained by taking all edges that touch vertices of V_2 and weighted edges between any vertex of D and any vertex of V. Instead of adding these weighted edges, we can simply take a tree of shortest paths rooted at each vertex of D. The total number of edges is still $O(n^{3/2}(\log n)^{1/2})$. It is easy to check that the resulting subgraph is a 2-emulator. The proof is similar to the proofs of Theorems 3.1 and 4.1.

The above construction can be carried out in $O(n^{5/2}(\log n)^{1/2})$ time. The most time-consuming task is running BFS from all the vertices of D. To reduce the running time to $\tilde{O}(n^2)$, we split the vertices into $O(\log n)$ classes instead of just two. (A similar idea is used in [2].) The resulting algorithm $\mathbf{emul}_2$ is given in Figure 6. Note that s_{k-1}, the last degree threshold in $\mathbf{emul}_2$, is about $(n \log n)^{1/2}$.

It is easy to see that the running time of $\mathbf{emul}_2$ is $O(\sum_{i=1}^{k-1} |D_i| \cdot |E_i|) = O(n^2 \log n \cdot \sum_{i=1}^{k-1} s_{i-1}/s_i) = O(n^2 \log^2 n)$, and that the number of edges in the set F produced by $\mathbf{emul}_2$ is $O(|E_k| + n \cdot \sum_{i=1}^{k} |D_i|) = O(n^{3/2}(\log n)^{1/2})$.

All that remains is to show that $G' = (V, F)$ is indeed a 2-emulator of G. Note that $G' = (V, F)$ is a subgraph of $G = (V, E)$. This immediately implies that $\delta_G(u,v) \leq \delta_{G'}(u,v)$ for every $u, v \in V$. We have to show that $\delta_G(u,v) \leq \delta_{G'}(u,v) \leq \delta_G(u,v) + 2$ for every $u, v \in V$. As usual, we consider two different cases.

Case 1. There is a shortest path between u and v in G all of whose edges are contained in E_k.

In this case, $\delta_{G'}(u,v) = \delta_G(u,v)$.

Case 2. Every shortest path between u and v in G contains edges that are not contained in E_k.

Consider a shortest path p between u and v in G. Let w be a vertex of highest degree on p. Let $1 \leq i < k$ be such that $w \in V_i \setminus V_{i-1}$ (where $V_0 = \phi$). Let $w' \in D_i$ be such that $(w, w') \in E$. Because all the vertices on p do not belong to V_{i-1}, we get that all the edges on p as well as the edge (w, w') belong to E_i. The shortest

```
Algorithm emul_2:

input: An unweighted undirected graph $G = (V, E)$.
output: A subgraph 2-emulator $(V, F)$ of $G$.

Let $k \leftarrow \lceil \frac{1}{2} \log_2(n/\log_2 n) \rceil$
For $i \leftarrow 0$ to $k-1$ let $s_i \leftarrow n/2^i$

$(\langle E_1, E_2, \ldots, E_k, E^* \rangle, \langle D_1, D_2, \ldots, D_k \rangle) \leftarrow$ decompose$(G, \langle s_1, s_2, \ldots, s_{k-1} \rangle)$

For $i \leftarrow 1$ to $k-1$ do
For every $u \in D_i$ run bfs$(\,(V, E_i)\,, \hat{\delta}, u)$

The edge set $F$ is composed of $E_k$ and the edges
of all the shortest paths trees found in all the BFS runs.
```

FIG. 6. *An $\tilde{O}(n^2)$-time algorithm for generating a subgraph 2-emulator.*

paths tree constructed by running BFS on (V, E_i) from w' contains therefore a path from w' to u of length at most $\delta_G(u, w) + 1$ and a path from w' to v of length at most $\delta_G(w, v) + 1$. It follows that this shortest paths tree, and therefore $G' = (V, F)$, contains a path from u to v of length at most $\delta_G(u, v) + 2$.

This completes the proof of the theorem. $\square$

A subgraph 2-emulator is also an additive 2-spanner and a multiplicative 3-spanner. In section 7 (Theorem 7.3) we show that weighted graphs also have 3-spanners of size $\tilde{O}(n^{3/2})$. We present there an algorithm whose running time is $\tilde{O}(mn^{1/2})$ for finding such 3-spanners. Because there are bipartite graphs with $\Omega(n^{3/2})$ edges that do not contain cycles of length four [32], this result is tight up to polylogarithmic factors. We can also show the following result.

THEOREM 6.3. *Every unweighted undirected graph $G = (V, E)$ on n vertices has a 4-emulator with $\tilde{O}(n^{4/3})$ edges. Such a graph can be constructed in $\tilde{O}(n^{7/3})$ time.*

Proof. It is not difficult to check that the graph $G_3 = (V, E_3 \cup E^* \cup (D_1 \times V) \cup (D_2 \times D_2) \cup (V \times D_1))$, in the notations of algorithm $\overline{\textbf{apasp}}_3$, is a 4-emulator of $G = (V, E)$. $\square$

It is tempting to claim that k-emulators for $k > 4$ can be similarly obtained by running $\overline{\textbf{apasp}}_k$ with $k > 4$. Unfortunately, this is not true. The fact that all the edges that touch a vertex u are added to the graph on which distances are found from u seems to be crucial there. We cannot do the same with emulators because we are supposed to use the same graph for all sources.

Let e_k be the infimum of all numbers for which each graph on n vertices has a k-emulator with $\tilde{O}(n^{1+e_k})$ edges. We have shown that $e_2 \leq 1/2$ and $e_4 \leq 1/3$. Does $e_k \to 0$ as $k \to \infty$? This remains an intriguing open problem.

We are not able to construct emulators with $o(n^{4/3})$ edges. We can, however, construct 6-emulators with $\tilde{O}(n^{4/3})$ edges in $\tilde{O}(n^2)$ time.

THEOREM 6.4. *Let $G = (V, E)$ be an unweighted undirected graph of n vertices. A 6-emulator of G with $O(n^{4/3}(\log n)^{2/3})$ edges can be constructed in $O(n^2 \log^2 n)$ time.*

Proof. The required 6-emulator is generated by algorithm **emul**$_6$ given in Figure 7. Algorithm **emul**$_6$ is similar to algorithm **emul**$_2$. Note that s_k, the last degree

```
Algorithm emul_6:

input: An unweighted undirected graph $G = (V, E)$.
output: A 6-emulator $(V, F)$ of $G$.

Let $k \leftarrow \lceil \frac{2}{3} \log_2(n/\log_2 n) \rceil$
For $i \leftarrow 0$ to $k-1$ let $s_i \leftarrow n/2^i$

$(\langle E_1, E_2, \ldots, E_k, E^* \rangle, \langle D_1, D_2, \ldots, D_k \rangle) \leftarrow$ decompose$(G, \langle s_1, s_2, \ldots, s_{k-1} \rangle)$

For every $u, v \in V$ do
if $(u, v) \in E$ then $\hat{\delta}(u, v) \leftarrow 1$ else $\hat{\delta}(u, v) \leftarrow +\infty$

For $i \leftarrow 1$ to $k-1$ do
For every $u \in D_i$ run bfs$(\,(V, E_i \cup E^*)\,\hat{\delta}, u)$

Let $F \leftarrow E_k \cup E^* \cup_{i=1}^{k-1} D_i \times D_{k-1}$
The weight of an edge $(u, v) \in F$ is $\hat{\delta}(u, v)$
```

FIG. 7. *An $\tilde{O}(n^2)$-time algorithm for generating a 6-emulator.*

threshold in **emul**$_6$, is about $n^{1/3}(\log n)^{2/3}$.

It is straightforward to verify that **emul**$_6$ runs in $O(n^2 \log^2 n)$ time and that the edge set F is of size $O(n^{4/3}(\log n)^{2/3})$. All that remains is to show that $H = (V, F)$ is a 6-emulator of G. We have to show that for every $u, v \in V$ we have $\delta_G(u, v) \le \delta_H(u, v) \le \delta_G(u, v) + 6$. Let $u, v \in V$. The fact that $\delta_G(u, v) \le \delta_H(u, v)$ is obvious. Let p be a shortest path from u to v in G. We again consider two cases.

Case 1. All the edges of p are contained in E_k.

In this case, $\delta_H(u, v) = \delta_G(u, v)$.

Case 2. The path p contains edges that do not belong to E_k.

The path p must pass through at least two vertices from V_{k-1}. Let w_1 and w_3 be the first and last such vertices on the path (the vertex w_1 may be u and the vertex w_3 may be v). Let w_2 be a vertex with the maximum degree on the path (the vertex w_2 may be one of w_1 and w_3). Let $1 \le i < k$ be such that $w_2 \in V_i \setminus V_{i-1}$. Let $w'_1, w'_3 \in D_{k-1}$ be neighbors of w_1 and w_3 such that $(w_1, w'_1) \in E^*$ and $(w_3, w'_3) \in E^*$. Let $w'_2 \in D_i$ be a neighbor of w_2 such that $(w_2, w'_2) \in E^*$.

Because w_1 and w_3 are the first and last vertices from V_{k-1} on the path, and because $(w_1, w'_1), (w_3, w'_3) \in E^*$, we get that $\delta_H(u, w'_1) \le \delta_G(u, w_1) + 1$ and $\delta_H(w'_3, v) \le \delta_G(w_3, v) + 1$. Since w_2 is a vertex with maximum degree on the shortest path from u to v, the shortest path p and the edges $(w_1, w'_1), (w_2, w'_2), (w_3, w'_3)$ are contained in the graph $G_i = (V, E_i \cup E^*)$. We get, therefore, that $\delta_G(w_1, w_2) = \delta_{G_i}(w_1, w_2)$ and $\delta_G(w_2, w_3) = \delta_{G_i}(w_2, w_3)$ and thus $\delta_{G_i}(w'_1, w'_2) \le \delta_G(w_1, w_2) + 2$ and $\delta_{G_i}(w'_2, w'_3) \le \delta_G(w_2, w_3) + 2$. For every $x \in D_i$ and $y \in D_k$, we have added to F an edge (x, y) whose weight is at most $\delta_{G_i}(x, y)$. We get therefore that $\delta_H(w'_1, w'_2) \le \delta_G(w_1, w_2) + 2$ and $\delta_H(w'_2, w'_3) \le \delta_G(w_2, w_3) + 2$. Combining these bounds we get

$$\begin{aligned}\delta_H(u, v) &\le \delta_H(u, w'_1) + \delta_H(w'_1, w'_2) + \delta_H(w'_2, w'_3) + \delta_H(w'_3, v) \\ &\le (\delta_G(u, w_1) + 1) + (\delta_G(w_1, w_2) + 2) + (\delta_G(w_2, w_3) + 2) + (\delta_G(w_3, v) + 1) \\ &= \delta_G(u, v) + 6.\end{aligned}$$

This completes the proof of the theorem. □

It is easy to see that k-emulators are Steiner $(k{+}1)$-spanners. It follows easily from the arguments of Althöfer et al. [5] and the constructions of Wenger [32] that there are unweighted undirected graphs on n vertices for which every Steiner 3-spanner, and therefore any 2-emulator, must have $\tilde{\Omega}(n^{3/2})$ edges, and there are graphs for which every Steiner 5-spanner, and therefore any 4-emulator, must have $\tilde{\Omega}(n^{4/3})$ edges (where $\tilde{\Omega}(f) = \Omega(f\,/\,\mathrm{polylog}\,n)$).

7. Stretched paths and distances.

In this section we describe algorithms for finding stretched paths in *weighted* graphs. We use the following result which is part of the folklore.

LEMMA 7.1 (truncated Dijkstra). *Let $G = (V, E)$ be a weighted graph on n vertices. Suppose that the adjacency lists of the vertices of G are sorted according to weight. Let $v \in V$ be a vertex of G and let $1 \le s \le n$. Shortest paths from v to s vertices closest to v can be found in $O(s(s + \log n))$ time.*

The set of s vertices returned by the truncated Dijkstra algorithm running from v is not uniquely defined, since there may be many vertices at the same distance from v. All that we require is that if S is the set of vertices returned by the algorithm then for every $u \in S$ and $w \in V \setminus S$ we have $\delta(v, u) \le \delta(v, w)$.

THEOREM 7.2. *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. We can preprocess the graph in $O((m \log n)^{2/3} n)$ time so that given any two vertices $u, v \in V$, we can in $O(1)$ time output an estimated distance $\hat{\delta}(u, v)$ satisfying $\delta(u, v) \le \hat{\delta}(u, v) \le 3{\cdot}\delta(u, v)$.*

Proof. Let $s \ge \log n$ be a parameter to be chosen later. We run the truncated Dijkstra algorithm from every vertex $v \in V$ and find a set $N(v)$ of s vertices closest to v. The time required for finding these sets is $O(ns^2)$. Next, we find a set D of size $d = O(n \log n/s)$ so that for every $v \in V$ there is $u \in D$ such that $u \in N(v)$. Such a set can be found in $O(ns)$ time. For every vertex $v \in V$, we keep a pointer to a vertex $u = P(v)$ such that $u \in D \cap N(v)$. We now run the full Dijkstra algorithm from all the vertices of D. The time required is $O(nm \log n/s)$. We keep a $d \times n$ matrix with the distances from the vertices of D to all the other vertices of the graph. The time used so far is $O(ns^2 + nm \log n/s)$. This is minimized by taking $s = (m \log n)^{1/3}$. The total time is then $O((m \log n)^{2/3} n)$.

Given a pair of vertices u and v, we first check whether $v \in N(u)$. If so, we output the exact distance $\delta(u, v)$ computed during the truncated Dijkstra from u. Otherwise, we let $w = P(u) \in D \cap N(u)$ and we output the estimated distance $\hat{\delta}(u, v) = \delta(u, w) + \delta(w, v)$. The distance $\delta(u, w)$ was found during the truncated Dijkstra from u. The distance $\delta(w, v)$ was found during the full Dijkstra from w.

Clearly $\delta(u, v) \le \hat{\delta}(u, v)$. If $v \in N(u)$, then $\hat{\delta}(u, v) = \delta(u, v)$. If $v \notin N(u)$, then $\delta(u, w) \le \delta(u, v)$ and the estimate $\hat{\delta}(u, v)$ satisfies

$$\begin{aligned}\hat{\delta}(u, v) &= \delta(u, w) + \delta(w, v)\\ &\le \delta(u, w) + (\delta(w, u) + \delta(u, v))\\ &\le 2\delta(u, w) + \delta(u, v) \;\le\; 3\delta(u, v),\end{aligned}$$

as required. □

Using essentially the same algorithm we can get the following result.

THEOREM 7.3. *Every weighted undirected graph $G = (V, E)$ on n vertices has a 3-spanner with $O(n^{3/2}(\log n)^{1/2})$ edges. Such a 3-spanner can be constructed in $\tilde{O}(m(n \log n)^{1/2})$ time.*

Proof. Let s be a parameter to be chosen later. Run the truncated Dijkstra algorithm from every vertex and find for every vertex $v \in V$ a set $N(v)$ of s vertices closest to v. Find a set D of size $O(n \log n/s)$ such that for every $v \in V$, there is a $u \in D \cap N(v)$. We then run a full Dijkstra from every vertex of D. The 3-spanner will be composed of the shortest paths trees found in all the truncated and full runs of Dijkstra's algorithm. The total number of edges will therefore be $O(ns + n^2 \log n/s)$. We choose $s = (n \log n)^{1/2}$. The number of edges in the 3-spanner is then $O(n^{3/2}(\log n)^{1/2})$ and the total running time is $O(ns^2 + nm \log n/s) = O(n^2 \log n + m(n \log n)^{1/2})$. Note that if $m(n \log n)^{1/2} \le n^2 \log n$, then $m \le n^{3/2}(\log n)^{1/2}$ and the original graph is the required 3-spanner. ▯

Cohen and Zwick [12] extend the techniques presented here and obtain, among other things, an $\tilde{O}(n^2)$-time algorithm for finding stretch 3 distances, and an algorithm whose running time is $\tilde{O}(n^{3/2}m^{1/2})$ for finding stretch 2 distances in weighted undirected graphs with n vertices and m edges.

8. Concluding remarks and open problems.

We have shown that surplus 2 estimates of all distances in an unweighted undirected graph on n vertices can be computed in $\tilde{O}(n^{7/3})$ time, i.e., faster than the fastest known matrix multiplication algorithm. Many open problems still remain. We end by mentioning some of them:

1. Is it possible to find surplus 2 estimated distances between all pairs of vertices in a graph on n vertices in $O(n^{7/3-\epsilon})$ time for some $\epsilon > 0$?
2. Is it possible to find surplus k estimated distances between all pairs of vertices in a graph on n vertices, for some fixed constant $k \ge 2$, in $\tilde{O}(n^2)$ time?
3. Do there exist fixed constants $k \ge 2$ and $\epsilon > 0$ such that every graph on n vertices has a k-emulator with $O(n^{4/3-\epsilon})$ edges?
4. Is it possible to find the *exact* distances between all pairs of vertices in an unweighted *directed* graph on n vertices in $\tilde{O}(M(n))$ time, where $M(n)$ is the time needed to multiply two $n \times n$ matrices?

Acknowledgments. We would like to thank Howard Karloff for some helpful discussions, Edith Cohen and David Peleg for some clarifications regarding their papers, and the two anonymous referees whose comments helped improve the presentation of the paper.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[2] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani, *Fast estimation of diameter and shortest paths (without matrix multiplication)*, SIAM J. Comput., 28 (1999), pp. 1167–1181.

[3] N. Alon, Z. Galil, and O. Margalit, *On the exponent of the all pairs shortest path problem*, J. Comput. System Sci., 54 (1997), pp. 255–262.

[4] N. Alon and M. Naor, *Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions*, Algorithmica, 16 (1996), pp. 434–449.

[5] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares, *On sparse spanners of weighted graphs*, Discrete Comput. Geom., 9 (1993), pp. 81–100.

[6] B. Awerbuch, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[7] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg, *Near-linear time construction of sparse neighborhood covers*, SIAM J. Comput., 28 (1999), pp. 263–277.

[8] J. Basch, S. Khanna, and R. Motwani, *On Diameter Verification and Boolean Matrix Multiplication*, Tech. Report STAN-CS-95-1544, Department of Computer Science, Stanford University, Stanford, CA, 1995.

[9] V. Chvátal, *A greedy heuristic for the set-covering problem*, Math. Oper. Res., 4 (1979), pp. 233–235.

[10] E. Cohen, *Polylog-time and near-linear work approximation scheme for undirected shortest paths*, in Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montréal, Canada, 1994, pp. 16–26.

[11] E. Cohen, *Fast algorithms for constructing t-spanners and paths with stretch t*, SIAM J. Comput., 28 (1999), pp. 210–236.

[12] E. Cohen and U. Zwick, *All-pairs small-stretch paths*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, 1997, pp. 93–102.

[13] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[14] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[15] E. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[16] T. Feder and R. Motwani, *Clique partitions, graph compression and speeding-up algorithms*, J. Comput. System Sci., 51 (1995), pp. 261–272.

[17] M. L. Fredman, *New bounds on the complexity of the shortest path problem*, SIAM J. Comput., 5 (1976), pp. 83–89.

[18] M. Fredman and R. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[19] Z. Galil and O. Margalit, *Witnesses for boolean matrix multiplication*, J. Complexity, 9 (1993), pp. 201–221.

[20] Z. Galil and O. Margalit, *All pairs shortest distances for graphs with small integer length edges*, Inform. Comput., 134 (1997), pp. 103–139.

[21] Z. Galil and O. Margalit, *All pairs shortest paths for graphs with small integer length edges*, J. Comput. System Sci., 54 (1997), pp. 243–254.

[22] D. Johnson, *Efficient algorithms for shortest paths in sparse graphs*, J. Assoc. Comput. Mach., 24 (1977), pp. 1–13.

[23] D. R. Karger, D. Koller, and S. J. Phillips, *Finding the hidden path: Time bounds for all-pairs shortest paths*, SIAM J. Comput., 22 (1993), pp. 1199–1217.

[24] A. Liestman and T. Shermer, *Additive graph spanners*, Networks, 23 (1993), pp. 343–363.

[25] L. Lovász, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383–390.

[26] C. McGeoch, *All-pairs shortest paths and the essential subgraph*, Algorithmica, 13 (1995), pp. 426–461.

[27] D. Peleg and A. Schäffer, *Graph spanners*, J. Graph Theory, 13 (1989), pp. 99–116.

[28] R. Seidel, *On the all-pairs-shortest-path problem in unweighted undirected graphs*, J. Comput. System Sci., 51 (1995), pp. 400–403.

[29] T. Takaoka, *A new upper bound on the complexity of the all pairs shortest path problem*, Inform. Process. Lett., 43 (1992), pp. 195–199.

[30] M. Thorup, *Undirected single source shortest paths in linear time*, in Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, 1997, pp. 12–21.

[31] J. D. Ullman and M. Yannakakis, *High-probability parallel transitive-closure algorithms*, SIAM J. Comput., 20 (1991), pp. 100–125.

[32] R. Wenger, *Extremal graphs with no C^4's, C^6's and C^{10}'s*, J. Combin. Theory Ser. B, 52 (1991), pp. 113–116.

[33] U. Zwick, *All pairs shortest paths in weighted directed graphs—exact and almost exact algorithms*, in Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, 1998, pp. 310–319.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1761–1815

IMPROVED DATA STRUCTURES FOR FULLY DYNAMIC BICONNECTIVITY[*]

MONIKA R. HENZINGER[†]

Abstract. We present fully dynamic algorithms for maintaining the biconnected components in general and plane graphs.

A fully dynamic algorithm maintains a graph during a sequence of insertions and deletions of edges or isolated vertices. Let m be the number of edges and n be the number of vertices in a graph. The time per operation of the best deterministic algorithms is $O(\sqrt{n})$ in general graphs and $O(\log n)$ in plane graphs for fully dynamic connectivity and $O(\min\{m^{2/3}, n\})$ in general graphs and $O(\sqrt{n})$ in plane graphs for fully dynamic biconnectivity. We improve the later running times to $O(\sqrt{m \log n})$ in general graphs and $O(\log^2 n)$ in plane graphs. Our algorithm for general graphs can also find the biconnected components of all vertices in time $O(n)$.

Key words. dynamic graph algorithms, biconnectivity, data structures

AMS subject classifications. 68P05, 68Q25

PII. S0097539794263907

1. Introduction.

Many computing activities require the recomputation of a solution after a small modification of the input data. Thus algorithms are needed that update an old solution in response to a change in the problem instance. *Dynamic graph algorithms* are data structures that, given an input graph G, maintain the solution of a graph problem in G while G is modified by insertions and deletions of edges.[1] In this paper we study the problem of maintaining the biconnected components (see below) of a graph.

We say that a vertex x is an *articulation point separating* vertex u and vertex v (or that x separates u and v) if the removal of x disconnects u and v. Two vertices are *biconnected* if there is no articulation point separating them. In the same way, an edge e is a *bridge separating* vertex u and vertex v if the removal of e disconnects u and v. Two vertices are 2-*edge connected* if there is no bridge separating them. A *biconnected component* or *block* (resp., 2-*edge connected component*) of a graph is a maximal set of vertices that are biconnected (resp., 2-edge connected). Note that biconnectivity implies 2-edge connectivity but not vice versa.

Given a graph $G = (V, E)$, a *dynamic biconnectivity algorithm* is a data structure that executes an arbitrary sequence of the following operations:

insert(u, v): Insert an edge between node u and node v.
delete(u, v): Delete the edge between node u and node v if it exists.
query(u, v): Returns *yes* if u and v are biconnected, and *no* otherwise.
complete-*block*-*query*: Return for all nodes all the blocks they belong to.

Operations *insert* and *delete* are called *updates*. To compare the asymptotic performance of dynamic graph algorithms, the time per update, called *update time*, and the time per query, called *query time*, are compared. Let m be the number of edges

[*]Received by the editors March 2, 1994; accepted for publication (in revised form) July 21, 1999; published electronically April 4, 2000.
http://www.siam.org/journals/sicomp/29-6/26390.html

[†]Google, Inc., 2400 Bayshore Parkway, Mountain View, CA 94043 (monika@google.com). This research was done while at the Department of Computer Science, Cornell University, Ithaca, NY 14853.

[1]Insertions or deletions of isolated vertices are usually trivial.

and n be the number of vertices in the graph. Prior to this work, the best update time was $O(\min(m^{2/3}, n))$ [11, 2] with a constant query time. This paper presents an algorithm with $O(\sqrt{m \log n})$ update time and constant query time. Subsequently, the sparsification technique was applied to the algorithm in this paper and its running time was improved to $O(\sqrt{n \log n} \log(m/n))$ [13]. Recently, a data structure that requires only polylogarithmic amortized time per operation was presented [15]. The data structure presented in this paper can answer complete-block-queries in time $O(n)$. This was not mentioned explicitly but can also be done with the data structure in [11, 2].

Additionally, we give an algorithm with $O(\log^2 n)$ update time and $O(\log n)$ query time for planar embedded graphs, under the condition that each insertion maintains the planarity of the embedding. The best previous algorithm took time $O(\sqrt{n})$ per update and $O(\log n)$ per query.

Related work. Frederickson [5] gave the first dynamic graph algorithm for maintaining a minimum spanning tree and the connected components. His algorithm takes time $O(\sqrt{m})$ per update and $O(1)$ per query operation. The first dynamic 2-edge connectivity algorithm by Galil and Italiano [8] took time $O(m^{2/3})$ per update and query operation. It was consequently improved to $O(\sqrt{m})$ per update and $O(\log n)$ per query operation [6]. The sparsification technique of Eppstein et al. [3] and Eppstein, Galil, and Nissenzweig [2] improves the running time of an update operation to $O(\sqrt{n})$. Subsequently, a dynamic connectivity algorithm was given with $O(n^{1/3} \log n)$ update time and $O(1)$ query time [12]. It can also output all nodes connected to a given node in time linear in their number. Very recently, an algorithm with polylogarithmic amortized time per operation was presented [15]. Note that there is a lower bound on the amortized time per operation of $\Omega(\log n / \log \log n)$ for all these problems [7, 16].

The best known dynamic algorithms in plane graphs take time $O(\log n)$ per operation for maintaining connected components by Eppstein et al. [1], $O(\log^2 n)$ for maintaining 2-edge connected components by Hershberger, Rauch, and Suri [14] and Eppstein et al. [4], and $O(\sqrt{n})$ for maintaining biconnected components by Eppstein et al. [4].

Outline of the paper. First (section 2), we study the dynamic biconnectivity problem for *general* graphs. Our basic approach is to partition the graph G into small connected subgraphs called *clusters* (see [5] for a first use of this technique in dynamic graph algorithms). Each biconnectivity query between a vertex u and a vertex v can be decomposed into a query in the cluster of u, a query in the cluster v, and a query between clusters. To test biconnectivity between clusters we use the 2-dimensional topology tree data structure [5] in a novel way and extend the ambivalent data structure [6]. These data structures were used before to test connectivity and 2-edge connectivity.

To test biconnectivity within a cluster we need to know how the vertices outside the cluster are connected with each other. Thus, we build two graphs, called *internal* and *shared* graphs. Each graph contains all vertices and edges inside the cluster C and a *compressed certificate* of $G \setminus C$. A compressed certificate is a graph that has the same connectivity properties as $G \setminus C$, but is not necessarily a minor of $G \setminus C$. This approach is similar to the concept of strong certificates in the sparsification technique: a strong certificate is not necessarily a subgraph of the given graph. The crux in the analysis of the algorithm is that we can show that only an amortized constant number of compressed certificates need "major" updates after an update in G (see Lemma 2.44).

Second (section 3), we study the dynamic biconnectivity problem for *plane* graphs. We use a topology tree approach based on [5].

An earlier version of this paper appeared in [17].

2. General graphs.

Let G be an undirected graph with n vertices and m edges. The *size* $|G|$ of a graph is the total number of its nodes and edges. We assume in the paper that G is connected, which implies $m \geq n-1$. If G is not connected, we build the data structure described below for each connected component and during an update combine two data structures or split a data structure in time $O(\sqrt{m \log n})$.

Let $m/\log^2 n \geq k \geq \sqrt{m}$ be a parameter to be determined later. Note that $m/k \leq k$. We build a data structure for G that we rebuild from scratch every m/k update operations. The operations between two rebuilds form a *phase*. This allows us to limit the necessary maintenance of the data structure within a phase, i.e., the data structure slowly "deteriorates" within a phase.

The data structure consists of the following parts: We map G to a graph G' of degree at most 3 and partition G' into $O(m/k)$ many subgraphs of size $O(k)$, called *clusters*. We keep data structures for (1) the graph of clusters, (2) each cluster (called *cluster graph*), and (3) for special *shared* nodes (called *shared graphs*). We will show how to rebuild these data structures in time $O(m \log n)$ and update them in time $O(\sqrt{m \log n})$ after an edge insertion or deletion in G.

2.1. The graph G' and the relaxed partition of order k.

We want to partition G into about equally sized subgraphs (see the relaxed partition below). For this purpose, we map G to a graph G' of degree at most 3 as in [5]. At each rebuild, G' is created by *expanding* a vertex u of degree $d \geq 4$ by $d-2$ new degree-3 vertices $u'_1, \ldots, u'_{d-2}$ and connecting u'_i and u'_{i+1} by a *dashed* edge, for $1 \leq i \leq d-3$. Every node u of degree at most 3 is represented by one node u'_1 in G'. Every edge (u, v) is replaced by a *solid* edge (u'_i, v'_j), where i and j are the appropriate indices of the edge in the adjacency lists for u and v. We say that the edge (u'_i, u'_{i+1}) *belongs* to u and that every u'_i is a *representative* of u. The vertex u of G is called the *origin* of the vertex u'_i in G', for $1 \leq i \leq d-2$. We denote vertices of G' by variables with prime, like u' or u'_i, and their origin by variables without prime, like u. Thus, at the beginning of a phase the graph G' contains at most $2m$ vertices and at most $3m$ edges. We use $deg(u)$ to denote the degree of a vertex u in G and $num(u)$ to denote the number of representatives of u in G'.

The graph G' is maintained during insertions and deletions of edges as follows: Consider how the representatives of u are updated when an edge (u, v) is inserted. If $num(u)$ is 1 and $deg(u)$ was 3 before the insertion, then add a new vertex u'_2, connect u'_1 and u'_2 by a dashed edge, and make u'_1 and u'_2 each incident to 2 edges incident to u. If $num(u)$ is greater than 1, then one new vertex $u'_{num(u)+1}$ is created and connected by a dashed edge to $u'_{num(u)}$. Thus an edge insertion increases the number of vertices in G' by up to 2 and the number of edges by up to 3. If an edge is deleted, the corresponding edge is removed from G', *but no vertices are deleted from* G'. Thus, within a phase there are at most $2m + 2m/k$ vertices and $3m + 3m/k$ edges in G'.

Data structure: The algorithm keeps the following mapping from G to G' and vice versa:

(G1) Each vertex of G stores a list of its representatives, ordered by index, and pointers to the beginning and the end of this list. Each vertex of G' keeps a pointer to its position in this list, to its origin, and a list of incident edges.

(G2) Each dashed edge of G' stores a pointer to the vertex of G that it belongs to; each solid edge of G' stores a pointer to the edge of G that it represents.

Note that there exists a spanning tree of G' that contains every dashed edge. The algorithm maintains such a spanning tree, denoted by T'. Let T be the corresponding spanning tree in G. We denote by $\pi_{T'}(u', v')$ the path from u' to v' in T'. If the spanning tree is understood, we use $\pi(u', v')$. Let u'_v denote the representative of u with the shortest tree path to a representative of v. Note that every articulation point separating u and v must have a representative that lies on $\pi_{T'}(u'_v, v'_u)$.

Data structure:

(G3) Both T and T' are stored in a degree-k ET-tree data structure [12] and in a dynamic tree data structure [18]. The ET-tree has constant depth.

We build a "balanced" decomposition of G' into subgraphs of size $O(k)$ and maintain data structures based on this decomposition. At the beginning of each phase, the decomposition and data structures are rebuilt from scratch in time $O(m \log n)$, adding an amortized cost of $O(k \log n)$ to each update. The rebuilds significantly simplify the "rebalancing" operations needed to maintain the decomposition balanced during updates.

We next describe the balanced decomposition of G' into *clusters*. A *cluster* is a set of vertices of G' that induces a connected subgraph of T'. If the representatives of a vertex u of G belong to different clusters, u is called a *shared vertex*. An edge is *incident* to a cluster if exactly one of its endpoints is in the cluster. An edge is *internal* if both endpoints are in the cluster. Let (x, y) be a tree edge incident to a cluster C and let $x \in C$. Then x is called a *boundary node* of C. The *tree degree* of a cluster is the number of tree edges incident to the cluster. Let $|C|$ denote the number of nodes in a cluster.

A *relaxed partition of order k* with respect to T' is a partition of the vertices into clusters so that

(C1) each cluster contains at most $k + 2m/k$ vertices of G';
(C2) each cluster has tree degree at most 3;
(C3) each cluster with tree degree 3 has cardinality 1;
(C4) if a cluster contains a shared vertex, then all boundary nodes are representatives of this shared vertex;
(C5) there are $O(m/k)$ many clusters; and
(C6) at the beginning of the phase if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster either is adjacent to a representative of the shared vertex or is incident to another cluster sharing s.

This definition is an extension of [6]. We denote the cluster containing a node u' of G' by $C_{u'}$.

If all representatives of a vertex u of G belong to the same cluster C, we denote C by C_u and say that C *contains* u and u *belongs to* C. For a cluster C, let $V(C)$ denote the set of origins in G of the nodes of G' that (1) either belong to C or (2) are connected to a node in C by a solid tree edge.[2]

Each cluster containing a representative of a shared vertex u is called a cluster *sharing* u or u*-cluster*. Condition (C4) implies that each cluster shares at most one vertex. Together with condition (C5), it follows that there are $O(m/k)$ shared vertices.

[2]The origin of a node s' that is connected by a dashed edge to a node t' in C belongs to $V(C)$ since the origin of s' equals the origin of t', which belongs to $V(C)$ according to (1). Thus set C_T of [11] equals $V(C)$.

Data structure:

(G4) Each cluster keeps (a) a doubly linked list of all its vertices, (b) a doubly linked list of all its incident tree edges, and (c) a pointer to its shared vertex (if it exists).

(G5) Each nonshared vertex of G' keeps a pointer to the cluster it belongs to. Each shared vertex keeps a doubly linked list of all the clusters that share the vertex.

We repeatedly make use of the following fact.

FACT 2.1. *Let G be an n-node graph and let $u_1, \ldots, u_a$ be a set of articulation points that lie on a (simple) path in G. Then $\sum_i deg(u_i) \leq 2n$.*

To guarantee that conditions (C1)–(C4) are maintained within a phase any cluster violating the conditions is split into two clusters (see section 2.2). We show below that all these splits create only $O(m/k)$ new clusters, i.e., condition (C5) is always fulfilled.

2.2. Maintaining a relaxed partition of order k.

To maintain a relaxed partition during updates, we create a more restricted partition at each rebuild, i.e., at the beginning of each phase, and let it gradually "deteriorate" during updates. Specifically, a cluster might be split but two clusters will never be merged. This implies that if a vertex becomes shared at some point in a phase it stays shared until the end of the phase.

A *restricted partition of order k* with respect to T' is a partition of the vertices into clusters so that

(C1′) each cluster has cardinality at most k;
(C2′) each cluster has tree degree ≤ 3;
(C3′) each cluster with tree degree 3 has cardinality 1;
(C4′) if a cluster contains a shared vertex, then all boundary nodes are representatives of this shared vertex;
(C5′) there are $O(m/k)$ clusters; and
(C6′) if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster is either adjacent to the representative of the shared vertex or is incident to another cluster sharing s.

LEMMA 2.2. *A partition fulfilling* (C1′)–(C5′) *can be found in linear time.*

Proof. The algorithm in [6] shows how to find a partition fulfilling (C1′)–(C3′) and (C5′) in time $O(m + n)$.

Enforcing (C4′) *for a cluster in time linear in its size:* Each cluster that does not fulfill (C4′) has tree degree 2. Let x' and y' be the two boundary nodes in the cluster. Since x' and y' represent different shared vertices, the tree path between x' and y' contains at least one solid edge. Splitting the cluster at the solid edge on $\pi(x', y')$ closest to x' and at the solid edge on $\pi(x', y')$ closest to y' creates at most three clusters that fulfill conditions (C1′)–(C4′). Each split takes time linear in the size of the cluster. The lemma follows. □

LEMMA 2.3. *By modifying the spanning tree T' a partition fulfilling* (C1′)–(C5′) *can be modified in time $O(m \log n)$ to additionally fulfill* (C6′).

Proof. We need to enforce that if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster is either adjacent to a representative of the shared vertex or is incident to another cluster sharing s. The main idea of our approach is to make every edge that violates this condition a tree edge, i.e., to remove a subtree of T' connected to the violating edge from the cluster sharing s.

To be precise, mark all vertices in clusters sharing s and test every edge incident to such a vertex whether it is adjacent either to a shared vertex or to a marked vertex. If x' is in an s-cluster and the edge (x', y') does not fulfill the above condition, then determine the representative s' of s that is closest to x' in T' and the tree edge e incident to s' on $\pi(s', x')$. Make e a non-tree edge and make (x', y') a tree edge. Remove the subtree $\tilde{T}$ of $T' \setminus e$ containing x' from the s-cluster and add it to the cluster of y' if the resulting cluster $C_{y'}$ does not violate (C1$'$). Otherwise, create a cluster containing $\tilde{T}$. This increases the tree degree of $C_{y'}$ by 1. Since $C_{y'}$ contains y', a vertex which had an adjacent non-tree edge, it follows from (C3$'$) that $C_{y'}$ has now tree degree at most 3. If it has degree 3 and consists of more than one vertex, $C_{y'}$ is split as follows.

Let v', y', and z' be the three (not necessarily distinct) boundary nodes of $C_{y'}$. There exists a tree degree-3 node w' that belongs to $\pi(v', y')$, $\pi(v', z')$, and $\pi(y', z')$. The algorithm splits $C_{y'}$ at w' by creating a tree degree-3 cluster for w' alone and up to three additional tree degree-2 clusters, namely, if $v' \neq w'$, a cluster containing v', if $y' \neq w'$, a cluster containing y', and if $z' \neq w'$, a cluster containing z'. The new clusters fulfill (C1$'$)–(C3$'$). It is possible that (a) w' was not a shared vertex before the split, but is a shared vertex after the split, and that (b) $C_{y'}$ shared a vertex u' different from w' before the split. If (a) and (b) hold, then one of the new tree degree-2 clusters might violate condition (C4$'$). Split these clusters in the same way as described in the proof of Lemma 2.2, namely, at the solid edge on $\pi(w', u')$ closest to w' and at the solid edge on $\pi(w', u')$ closest to u'. This creates at most three clusters that fulfill conditions (C1$'$)–(C4$'$).

Note that a constant number of clusters was formed from the vertices in $C_{y'}$ and the vertices in $\tilde{T}$ and that there are more than k vertices in $C_{y'}$ and $\tilde{T}$ combined. Thus the total number of clusters when (C6$'$) holds for all clusters is still $O(m/k)$, i.e., (C5$'$) continues to hold.

To implement the above algorithm in time $O(m \log n)$ we use the following data structure to represent T', G', and the current cluster partitioning.

(1) We store a list of clusters and for each cluster we store its boundary vertices, its incident tree edges, and its shared vertex if it exists. For each shared vertex s we keep a list of all s-clusters.

(2) Each vertex of G' stores a doubly linked list of all incident edges, separated by tree and non-tree edges. Each edge points to its two positions in these lists.

(3) We also assume data structure (G1) exists already. This data structure will not be modified by this algorithm.

(4) Each vertex of G stores a bit indicating whether it is shared or not.

(5) We build in linear time a degree-k ET-tree data structure [12]. By removing the tree edges incident to a cluster, determining the size of the resulting spanning tree inside the cluster, and adding the tree edges again this data structure allows us to determine the number of vertices of a cluster in time $O(\log n)$.

(6) We build two dynamic tree data structures [18]. We mark in linear time each edge of T' in one of the dynamic tree data structures (G3) by a weight which is 1 for solid edges and 0 for dashed edges. These data structures are updated in time $O(\log n)$ after each edge insertion or deletion in T'. We use the second dynamic tree data structure to mark or unmark in time $O(\log n)$ all the vertices on an arbitrary path and to determine in time $O(\log n)$ the marked vertex on a (second) path closest to one of the endpoints of the path.

This data structure can be built in time $O(m)$.

The algorithm checks each cluster on the list of clusters to determine whether it has a shared vertex s and, if so, it traverses and marks all vertices in the s-clusters starting at a boundary vertex. This takes time linear in the number of these vertices. Afterward it traverses them a second time and tests in constant time each adjacent non-tree edge whether it is incident to a marked vertex or a representative of a shared vertex. To test whether a vertex u' is the representative of a shared vertex u of G we determine the vertex u of G which u' represents and test the bit at u. This takes constant time.

We next show how to deal efficiently with an edge that violates the condition. Making a tree edge a non-tree edge or vice versa takes time $O(\log n)$. Determining the size of a cluster takes time $O(\log n)$ as discussed above. We still need to show how to determine s', e, w' and the solid tree edges closest to w' and closest to u' on $\pi(w', u')$ in time $O(\log n)$.

Determining s' and e: Mark in the second dynamic tree data structure all the representatives of s using (G1). Root the dynamic tree data structure at a vertex not in an s-cluster, for example, at y', and determine the marked vertex on $\pi(x', y')$ closest to x'. This vertex is s'. Then root the dynamic tree at x' and determine the edge from s' to its parent. This is the edge e. Finally unmark the marked vertices.

Determining w': Mark in the second dynamic tree data structure all the vertices on $\pi(v', z')$. Root the (updated) dynamic tree data structure at v' and determine the marked vertex on $\pi(y', v')$ closest to y'. This vertex is w'. Unmark the marked vertices.

Determining the solid tree edges closest to w' (resp., u') on $\pi(w', u')$: Root the first dynamic tree data structure at u' (resp., w') and determine the edge with weight 1 closest to w' (resp., u'). This is the desired edge.

Thus, for each non-tree edge that violates the condition we spend time $O(\log n)$ to make it a tree edge and to restore conditions (C1)–(C4). Note that if a tree edge becomes a non-tree edge, it stays a non-tree edge since it is adjacent to a shared vertex. Thus the cost of $O(\log n)$ is incurred $O(m)$ times, for a total time of $O(m \log n)$. ∎

We discuss next how to maintain a relaxed partition during updates. We show that an update does not violate condition (C1), (C4), or (C5). Obviously condition (C6) can never be violated since it only has to hold at the beginning of a phase. Condition (C2) or (C3) might be violated but can be restored by splitting a constant number of clusters. Restoring (C3) might lead to a violation of (C4), which can also be restored with an additional constant number of cluster splits.

We use the following *update algorithm for the relaxed partition*: If an insert(u, v) operation replaces u by two nodes, add both to C_u. If only one new representative $u_{num(u)+1}$ is created, add it to the cluster of $u_{num(u)}$ and increment $num(u)$ afterward. A deletion does not remove any vertices.

LEMMA 2.4. *The update algorithm for the relaxed partition does not violate condition* (C1), (C4), *or* (C5). *Conditions* (C2) *and* (C3) *might be violated for the clusters containing the endpoints of the newly inserted edge (in the case of an insertion) or the endpoints of the new tree edge (in the case of a deletion).*

Proof. Condition (C1)*:* An update increases the number of nodes in a cluster by at most two, implying that at the end of the phase each cluster contains at most $k + 2m/k$ nodes. It follows that condition (C1) is never violated.

Condition (C4)*:* Every newly added dashed edge has both endpoints in the same cluster, i.e., it is *not* incident to a cluster. Thus, condition (C4) is not violated.

Condition (C5)*:* A delete(u,v) operation might disconnect the connected component of C_u if $C_u = C_v$, leading to one additional cluster. Since there are m/k updates in a phase, there exist $O(m/k)$ clusters during a phase, i.e., condition (C5) is not violated by the update algorithm.

Conditions (C2) *and* (C3)*: Deletions:* Note first that the deletion of a non-tree edge does not invalidate condition (C2) or (C3) and, thus, does not require any cluster splits. A deletion of a tree edge might make a (solid) non-tree edge into a tree edge, and, if this edge is an intercluster edge, add one new incident tree edge to its endpoint clusters. This might lead to a violation of condition (C2) or (C3) for the clusters incident to the new tree edge. *Insertions:* In the case that G is disconnected, a newly inserted edge might become a tree edge, adding one new (solid) tree edge to at most two clusters. As before, this might lead to a violation of conditions (C2) and (C3) for the clusters incident to the new tree edge. Additionally, an insertion might increase the number of nodes in a tree degree-3 cluster to two, violating condition (C3).

In conclusion, an update violates conditions (C2) and/or (C3) for at most two clusters, namely, the clusters containing the endpoints of the newly inserted edge or of the new tree edge. $\square$

The algorithm first restores condition (C2) and then condition (C3). However, restoring (C3) might lead to the violation of condition (C4). If this happens, condition (C4) is restored after condition (C3).

Restoring condition (C2)*:* If a cluster violates (C2), it has tree degree 4 and consists of exactly two tree degree-3 nodes. Splitting it into two clusters creates two connected 1-node clusters of degree 3, fulfilling conditions (C1)–(C4).

Restoring condition (C3)*:* If a cluster C violates condition (C3), let x', y', and z' be the three (not necessarily distinct) boundary nodes of C. There exists a tree degree-3 node w' that belongs to $\pi(x', y')$, $\pi(x', z')$, and $\pi(y', z')$. The algorithm splits C at w' by creating a tree degree-3 cluster for w' alone and up to three additional tree degree-2 clusters, namely, if $x' \neq w'$, a cluster containing x', if $y' \neq w'$, a cluster containing y', and if $z' \neq w'$, a cluster containing z'. It is possible that (a) w' was not a shared vertex before the split, but is a shared vertex after the split, and that (b) C was incident to up to two dashed edges before the update, i.e., shared a vertex different from w'. If (a) and (b) hold, then one of the new tree degree-2 clusters might violate condition (C4). Splitting these clusters in the same way as in the proof of Lemma 2.2 creates at most three additional tree degree-2 clusters, each fulfilling conditions (C1)–(C4).

Thus, restoring conditions (C1)–(C4) requires creating a constant number of additional clusters after an update operation. Since there are m/k updates in a phase, condition (C5) is fulfilled at any point in a phase.

We summarize this discussion in the following lemma.

Lemma 2.5.

(1) *An insertion does not split any cluster, but restoring the relaxed partition after an insertion might require a constant number of cluster splits, namely, of the clusters that contain the endpoints of the inserted edge.*

(2) *A deletion of a non-tree edge does not require any cluster splits.*

(3) *A deletion of a tree edge might split the cluster containing the endpoints of the deleted edge. Additionally, restoring the relaxed partition after a deletion might require a constant number of cluster splits, namely, of the clusters that contain the endpoints of the new tree edge.*

Testing whether a cluster violates (C2) or (C3) takes constant time; splitting a cluster takes time linear in its size. Thus, it takes time $O(k)$ to update the relaxed partition of order k after each update.

2.3. Queries. Mapping G to G' causes correctness problems: If two nodes u and v are biconnected in G, they are also biconnected in G', but the reverse statement does *not* always hold (see [11] for an example).

The following lemma (an extension of Lemma 2.2 of [11]) relates the biconnectivity properties of G and of G'. To *contract* an edge (u, v) identify u and v and remove (u, v). To *contract* a vertex of G contract all dashed edges in G' belonging to the vertex.

LEMMA 2.6. *Let u and v be two vertices of G.*

(1) *Let G_1 be the graph that results from G' by contracting every vertex on $\pi_T(u, v)$ (excluding u and v). The vertices u and v are biconnected in G iff u'_v and v'_u are biconnected in G_1.*

(2) *Let y be a node on $\pi_T(u, v)$ that does not separate u and v in G. Let G_2 be the graph that results from G' by contracting every vertex on $\pi_T(u, v)$, excluding y, u, and v. The vertices u and v are biconnected in G iff u'_v and v'_u are biconnected in G_2.*

Proof. The graphs G_1, resp., G_2, can be created from G by expanding appropriate vertices. Thus, if u and v are biconnected in G, then u'_v and v'_u are biconnected in G_1, resp., G_2.

For the other direction, assume that u and v are separated by an articulation point x in G. Then x belongs to $\pi_T(u, v)$. It follows that x is represented by one node in G_1, resp., G_2. Assume by contradiction that u'_v and v'_u are biconnected in G_1, resp., G_2, i.e., there exists a path P' between them not containing x. The corresponding path P in G connects u and v and does not contain x, which leads to a contradiction. ∎

Note that the lemma also holds if additional vertices of G_1, resp., G_2, are contracted.

To test the biconnectivity of u and v in G we decompose the problem into subproblems, such that each subproblem is either (a) a biconnectivity query in a graph of size $O(k + m/k)$ or (b) a connectivity query in a graph of size $O(m)$. Subproblems of type (a) can be solved efficiently since $k + m/k$ is chosen to be "small." For subproblems of type (b) we use the existing efficient data structures for maintaining connectivity dynamically. Since no data structure is known that solves both subproblems efficiently, we maintain two different data structures, called *cluster graphs* and *shared graphs.*

To be precise, for each shared vertex s, a *shared graph* is maintained. Given a shared vertex s and two of its tree neighbors x and y, the shared graph of s is used to test in constant time whether s is an articulation point separating x and y. This is equivalent to testing if x and y are disconnected in $G \setminus s$. Thus, we use the dynamic connectivity data structure [12] to maintain the shared graphs.

Recall that $V(C)$ for a cluster C denotes the set of origins in G of the nodes of G' that either (1) belong to C or (2) are connected to a node in C by a solid tree edge. We maintain for C a *cluster graph* which is built to test (in constant time) if any two nodes of $V(C)$ that are not separated by s_C are biconnected in G, where s_C is the shared vertex of C. In particular, the cluster graph can be used to test whether s_C is biconnected with another node in C. To maintain the cluster graphs we use the data structure of [11]. To test if two nodes x and y are biconnected in G, the data structure contracts in G' all vertices on $\pi_T(x, y)$ (and potentially additional vertices).

We describe next how we use these two data structures to answer a biconnectivity query. We use the following lemma.

LEMMA 2.7. *Let u and v be nodes of G and let $(x^{(i)'}, y^{(i)'})$, for $1 \leq i \leq p$, denote* *the solid intercluster tree edges on $\pi_{T'}(u'_v, v'_u)$, in the order of their occurrence. Then u and v are biconnected in G iff*

(Q1) *u and $y^{(1)}$ are biconnected in G,*

(Q2) *$x^{(i)}$ and $y^{(i+1)}$ are biconnected in G, for $1 \le i < p$, and*

(Q3) *$x^{(p)}$ and v are biconnected in G.*

Proof. If u and v are biconnected, then all nodes on $\pi(u, v)$ are pairwise biconnected, and thus (Q1)–(Q3) hold. If u and v are not biconnected, then u and v are separated by an articulation point z. Since z belongs to $\pi_T(u, v)$, either (Q1), (Q2), or (Q3) is violated. This is a contradiction. □

For a query(u, v), let C be the cluster of u'_v. If C shares a vertex, call it s. We test the conditions of the lemma using only a cluster graph if this is possible and using a cluster graph and a suitable shared graph otherwise.

Testing condition (Q1)*:* Condition (Q1) of Lemma 2.7 can be tested using two cluster graphs and the shared graph of s: if s does not lie on $\pi(u, y^{(1)})$, then s does not separate u and $y^{(1)}$, and $y^{(1)}$ belongs to $V(C)$. Thus, we use the cluster graph of C to test whether u and $y^{(1)}$ are biconnected.

If s lies on $\pi(u, y^{(1)})$, then let x_u and y_u be the nodes incident to s on $\pi(u, y^{(1)})$. Let s' be the node representing s closest to $y^{(1)}$ on $\pi(u, y^{(1)})$. Note that s belongs to $V(C)$ and that $y^{(1)}$ belongs to $V(C_{s'})$. Test in the cluster graph of C if u and s are biconnected, test in the cluster graph of $C_{s'}$ if s and $y^{(1)}$ are biconnected, and test in the shared graph of s if x_u and y_u are biconnected. If all tests are successful, u and $y^{(1)}$ are biconnected in G, since the last test guarantees that s does not separate u and $y^{(1)}$ and the first two tests guarantee that no other node of $\pi(u, y^{(1)})$ separates u and $y^{(1)}$.

Testing condition (Q2)*:* If $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same cluster C_i and C_i does not share a vertex, then both, $x^{(i)}$ and $y^{(i+1)}$, belong to $V(C_i)$ and are not separated by a shared vertex of C_i. Thus, condition (Q2) can be tested using the cluster graph of C_i.

We show that otherwise $x^{(i)}$ and $y^{(i+1)}$ are tree neighbors of a shared vertex s_i and the shared graph of s_i can be used to test condition (Q2): either (a) $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same cluster C_i that shares a vertex s_i, or (b) $y^{(i)'}$ and $x^{(i+1)'}$ belong to different clusters. In case (a), C_i is incident to two solid tree edges and one dashed tree edge, i.e., C_i has tree degree 3. By condition (C3) it follows that C_i contains only one node, i.e., $y^{(i)'} = x^{(i+1)'}$, and both are representatives of s_i. It follows that $x^{(i)}$ and $y^{(i+1)}$ are both tree neighbors of s_i. In case (b), all intercluster edges between the cluster of $y^{(i)'}$ and the cluster of $x^{(i+1)'}$ are dashed. By condition (C4) of a relaxed partition, all these dashed edges and also $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same shared vertex s_i. Thus, also in this case $x^{(i)}$ and $y^{(i+1)}$ are tree neighbors of s_i. It follows that the shared graph of s_i can be used to test whether $x^{(i)}$ and $y^{(i+1)}$ are biconnected in G.

Testing condition (Q3)*:* Condition (Q3) is tested analogously to condition (Q1).

Since each test takes constant time, this leads to a query algorithm whose running time is linear in the number of solid intercluster edges on $\pi_{T'}(u'_v, v'_u)$, which is $O(m/k)$. However, we will give in section 2.11 a data structure that allows all these tests to be executed in constant time.

Our next goal is to describe *cluster graphs* and *shared graphs* in detail. Maintaining them requires a third data structure, called *high-level graphs*, which we describe first.

2.4. Overview of high-level graphs. There are two *high-level graphs*, H_1 and H_2. Basically, H_1 is a graph where each cluster is contracted to one node, and H_2 is a copy of H_1 with intercluster dashed edges contracted as well.

To be precise, the graph H_1 contains a node for each cluster of G'. Two nodes C and C' of H_1 are connected by an edge in H_1 iff there is an edge between a vertex of C and a vertex of C'. If there exists an edge between C and C', we call C' the *neighbor* of C. The edge between C and C' in H_1 is a dashed edge iff there is a dashed edge between a vertex of C and a vertex of C'. Otherwise the edge in H_1 is *solid.*

The graph H_2 is the graph H_1 with all dashed edges of H_1 contracted.

We call vertices of H_1 or H_2 *nodes* and refer to vertices of G of G' as *vertices.*

Since each node of H_1 represents exactly one cluster, we will use the terms *node of* H_1 and *cluster* interchangeably. Each node of H_2 represents at least one cluster and it also represents the vertices of G with a representative in these clusters. Note that each vertex of G is represented by a unique node of H_2, while this does not hold for H_1: a shared vertex is represented by more than one node of H_1.

The spanning tree T' of G' induces a spanning tree T_1 on H_1 and T_2 on H_2. We say C' is a *tree neighbor* of C if there is a tree edge between C' and C in H_1. Otherwise C' is a *non-tree neighbor.*

We need the high-level graphs to define and maintain the cluster graphs and the shared graphs. Roughly speaking, a cluster graph tests (under certain conditions) whether two vertices represented by the same node of H_1 are biconnected in G, and a shared graph tests whether two vertices represented by the same node of H_2 are biconnected in G.

When maintaining cluster and shared graphs we make use of the following data structures. Details of some of these data structures are delayed until section 2.7. Let $i = 1, 2$.

(HL1) We store for each node of H_i all the vertices of G' belonging to the node,[3] and we store at each vertex of G' the node of H_i to which the vertex belongs.

(HL2) We keep the following adjacency list representation for H_i (of size $O((m/k)^2) = O(m)$): For each node of H_i we keep the list of all incident neighbors and a list of pointers to all of its positions in the lists of its neighbors. Thus, in constant time an edge between two nodes can be removed from this representation.

(HL3) We maintain a data structure that given two nodes C and C' returns the tree neighbor C'' of C such that C'' lies on $\pi_{T_i}(C, C')$. For H_1 this takes constant time; for H_2 it takes time $O(\log n)$.

(HL4) We store a data structure that implements the following query operations in H_i:

biconnected?(C,C′,C″): Given that nodes C' and C'' are both tree neighbors of a node C, test whether C' and C'' are biconnected in H_i.

blockid?(C,C′): Given that C and C' are tree neighbors in T_i, output the name of the biconnected component of H_i that contains both C and C'.

components?(C): Output the tree neighbors of C in H_i grouped into biconnected components.

Operations *biconnected?* and *blockid?* take constant time, and *components?* takes time linear in the size of the output.

(HL5) We keep a *mapping* h from H_1 to H_2 and a mapping h^{-1} from H_2 to H_1. For each node of H_2 we keep a list of pointers to all the nodes of H_1 whose contraction formed H_1, and for each node of H_1 we keep a pointer back to the corresponding node of H_2.

[3]For H_1 this is already part of (G4) and, of course, does not need to be stored twice.

(HL6) We keep an empty array of size $O(m/k)$ at each node in H_i (needed for various bucket sorts—see sections 2.5 and 2.6).

Note that using (HL3) and (HL4) one can test in constant time whether two neighbors of a node C in H_i are biconnected in H_i.

Recall that after an update operation clusters might be split to restore the relaxed partition.

LEMMA 2.8. *If a cluster C is split while restoring the relaxed partition, then the clusters containing the vertices of C when the relaxed partition is restored form a connected subgraph of H_1.*

Proof. Splitting simply regroups the partitioning of vertices into clusters but does not change the connectivity properties in G. Since the vertices in C form a connected subgraph of G before the split, they also do so after the split. Hence, the clusters containing these vertices form a connected subgraph of H_1. □

Next we need to assign "ancestors" to nodes and edges in H_1 and H_2. This is necessary for our lazy update scheme: if a node is split into two nodes, then the resulting nodes have the same ancestor. However, we cannot afford to update all cluster and shared graphs accordingly. Therefore, we will treat nodes with the same ancestor that fulfills certain additional conditions as one node in some of the cluster and shared graphs.

First we define a unique *ancestor* for each node in H_2. Let C be a node in the current graph H_2. Since clusters are only split, never joined, all nodes represented by C were represented by the same cluster A at the beginning of a phase. This node A is called the *ancestor* of C.

To define ancestors for nodes in H_1, we denote by B_s the cluster that contains $s_{num(s)}$ at the beginning of a phase for each shared vertex s. Note that if a cluster contains only representatives that were created after the last rebuild, then these representatives represent a shared vertex. Consider a cluster C of H_1. If C only contains representatives that were created after the last rebuild, then the ancestor of C is B_s. Otherwise, C has at least one representative that existed at the last rebuild. In this case, the representatives that existed at the last rebuild belonged to the same cluster A at the last rebuild. This node A is called the *ancestor* of C.

LEMMA 2.9. *Let $i = 1$ or 2. Assume a node C of H_i is split into clusters C_1 and C_2. Then C_1 and C_2 have the same ancestor as C.*

Proof. For $i = 2$ this follows immediately from the definition. For $i = 1$, we only give the argument for C_1; the same argument applies to C_2. If C_1 contains representatives that existed at the last rebuild, then these representatives all belonged to the same cluster at the last rebuild. This cluster is also the ancestor of C. Otherwise, C_1 contains only representatives that were created after the last rebuild. In this case the ancestor of C_1 is B_s. We show below inductively that representatives that are created after the last rebuild are added into clusters whose ancestor is B_s. Thus, the ancestor of C is B_s as well.

When a new representative is added for s, it is added to the cluster containing $s_{num(s)}$. We show inductively that the cluster containing $s_{num(s)}$ for the current value of $num(s)$ has B_s as ancestor. The induction goes over the number of representatives of s added to G' after the last rebuild. By the definition of ancestor for clusters with representatives that existed at the last rebuild the claim holds before a new representative was added for s. Consider next the addition of the ith new representative $s_{num(s)+1}$ of s. If the cluster containing $s_{num(s)}$ is not split, then the claim holds inductively. Otherwise two situations can arise: either the cluster C containing $s_{num(s)+1}$

after the split contains only representatives that were created after the last rebuild, or not. In the former case, B_s is the ancestor of C by definition. In the latter case, C contains representatives that belonged to B_s at the last rebuild. Thus, in either case, B_s is the ancestor of C. $\square$

In particular, C_1 and C_2 have the same ancestor and each node has a unique ancestor.

Data structure:

(HL7) We store at each node of H_i its ancestor.

(HL8) We number the edges of H_i in the order in which they were added to H_i with the edges added during a rebuild in arbitrary order.

Recall that at the beginning of each phase a cluster contains at most k vertices of G'. This enables us to prove the following lemma.

LEMMA 2.10. *The total number of vertices of G' in all clusters with the same ancestor is $k + 2m/k$. The total number of edges incident to these vertices is $O(k)$.*

Proof. Let A be a cluster at the beginning of a phase. All nodes of a cluster with ancestor A (in G') that existed at the time of the last rebuild belonged to A at the beginning of the phase. Thus, there are at most k of them. Every other node was created by one of the m/k update operations. Since each update creates at most two new vertices, the bound follows. $\square$

2.5. Cluster graphs. Let C be a cluster. The *cluster graph* $I(C)$ of C is used to test if two vertices u and v of $V(C)$ that are not separated by s_C are biconnected in G. This leads to a first requirement for $I(C)$:

(IC1) *If s_C does not separate u and v, then u and v are biconnected in $I(C)$ iff they are biconnected in G.*

As we see below, an amortized constant number of cluster graphs is rebuilt during each update operation. This leads to a second requirement for $I(C)$:

(IC2) *The graph $I(C)$ has size $O(|C|) = O(k)$.*

Recall that $V(C)$ denotes the set of origins in G of the nodes of G' that (1) either belong to C or (2) are connected to a node in C by a solid tree edge.

We next motivate our definition of $I(C)$ and explain why it can be used only if s_C does not separate the two nodes. Obviously, $I(C)$ has to contain all nodes of $V(C)$ and all edges between the nodes of $V(C)$. Since two nodes of $V(C)$ can be connected by a path in $V(C)$ and additionally by a path that contains nodes of a non-tree neighbor of C, we represent each non-tree neighbor of C by a node in $I(C)$ (called either *b-node* or *c-node*) and we add to $I(C)$ all edges incident to C.

However, three questions remain: (1) If C shares a vertex s_C, let C' be one of the neighbors of C connected to C by a dashed tree edge (belonging to s_C). Should $I(C)$ also contain a node representing C', i.e., should s_C and C' be represented by the same or different nodes in $I(C)$? (2) How is the set of b- and c-nodes connected by edges? (3) How can the graph be maintained efficiently when a neighbor of C is split?

Next we describe our solution to these questions. (1) If s_C and C' are represented by the same node, then two nodes u and v of $V(C)$ that are biconnected in G might not be biconnected in $I(C)$. See Figure 2.1 for an example. On the other side, if $I(C)$ contains a node for s_C and a separate node for C', then two vertices u and v of $V(C)$ that are not biconnected in G can be biconnected in $I(C)$. See Figure 2.2 for an example.

However, by Lemma 2.6 and the fact that in the latter approach all nodes on $\pi_T(u, v)$ except for s_C are contracted, it follows that the latter situation can happen

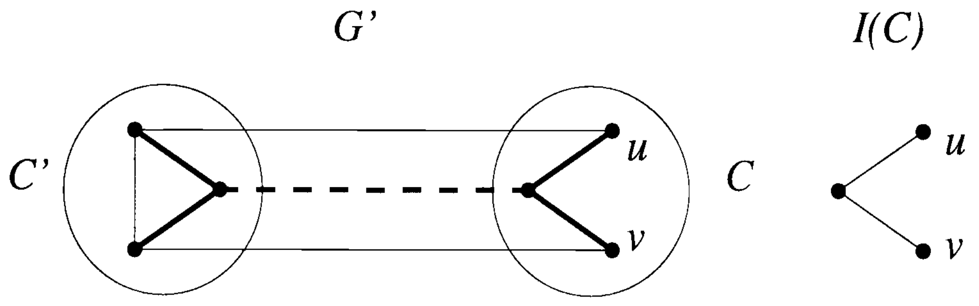


FIG. 2.1. *The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are collapsed to one node. Nodes u and v are biconnected in G but not in $I(C)$.*

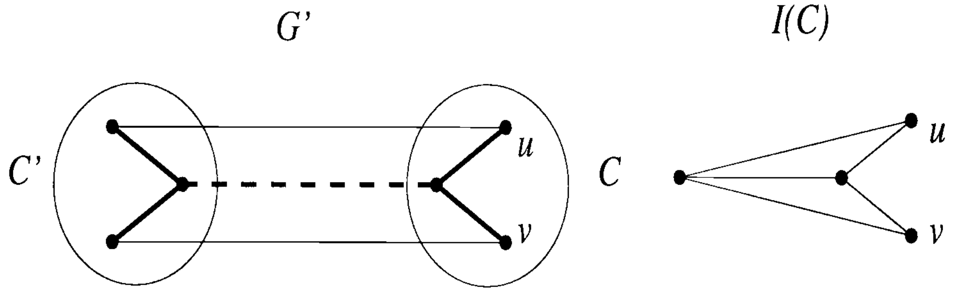


FIG. 2.2. *The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are represented by two different nodes. Nodes u and v are not biconnected in G' but are biconnected in $I(C)$.*

only if s_C separates u and v in G. Since this case is excluded by (IC1), we represent s_C and C' by separate nodes in $I(C)$.

(2) Let j be the number of neighbors of C. There are at most j b- or c-nodes in $I(C)$. Since G' is a graph of degree at most 3, $j = O(|C|)$. To guarantee that $I(C)$ has size $O(|C|)$, $I(C)$ will contain at most $j-1$ many edges between b- or c-nodes. These edges will be colored and will fulfill the condition that two b- or c-nodes are connected by a path of colored edges iff they are connected in $H_1 \setminus C$.

(3) We will split a node representing a neighbor C' of C only if the two clusters resulting from the split of C' are disconnected in $H_1 \setminus C$. Otherwise, both resulting clusters will be represented by the same c-node in $I(C)$, i.e., a c-node in the cluster graph might represent not just one cluster, but a set of clusters. This leads to the following invariant: *Two neighbors of C are represented by the same node in $I(C)$ or are connected by a colored path iff they are connected in $H_1 \setminus C$.*

Let us now give the exact definition of a *cluster graph* $I(C)$ for a cluster C. Let A be the ancestor of C. The cluster graph contains as nodes

(1) a node, called *a-node*, for each vertex with a representative in C,

(2) one node, called *b-node*, for each neighbor C' of C with ancestor A,

(3) one node, called *c-node*, for each maximal set X of clusters such that (a) every cluster $C' \in X$ is a neighbor of C, (b) all clusters in the set are connected in

$H_1 \setminus C$, (c) all clusters in X have the same ancestor which is different from A, (d) at the creation of C, the set X contains only one cluster, and (e) at each previous point in time since the creation of C all clusters that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node in $I(C)$.

Note that for each neighbor C' of C there exists a unique node in $I(C)$ representing C' (and potentially other clusters). Note further that each node of G is represented by at most one node in $I(C)$, except for s_C, which can be represented by an a-node and up to two b- or c-nodes, namely, the tree neighbors of C that share s_C.

The graph $I(C)$ contains the following edges:

(1) All edges between two vertices of G represented by an a-node belong to $I(C)$.

(2) For each edge (u, v) where u is represented by an a-node and v is not, and (u', v') is the corresponding edge in G', there is an edge (u, d) in $I(C)$, where d is the b- or c-node representing $C_{v'}$.

(3) For each tree neighbor C_1 connected to C by a dashed edge there is an edge from the b- or c-node of C_1 to the a-node of s_C.

(4) For each pair C_1 and C_2 of tree neighbors of C there is a *red* edge (d_1, d_2) if C_1 and C_2 are biconnected in H_1, where d_j is the b- or c-node representing C_j, $j = 1, 2$.

(5) For each non-tree neighbor C_1 of C with representative d_1,[4] $I(C)$ contains a *blue* edge (d_1, d_2), where d_2 represents the tree neighbor of C that lies on $\pi_{T_1}(C, C_1)$, if C is an articulation point in H_1 (separating its at most two tree neighbors), and a *blue* edge (d_1, d_3), where d_3 represents an arbitrary tree neighbor of C, otherwise.[5]
Note that $I(C)$ can contain parallel edges. They can be discarded without affecting the correctness.

We show next that the cluster graphs fulfill (IC1) and (IC2).

LEMMA 2.11. *Two neighbors of C are biconnected in H_1 iff either they are represented by the same node in $I(C)$ or their representatives in $I(C)$ are connected by a colored path.*

Proof. We show first that if there is a colored edge between two nodes d_1 and d_2 in $I(C)$, then the clusters that they represent are biconnected in H_1. For red edges this follows immediately from the definition. For a blue edge consider first the case that C is not an articulation point in H_1. In this case all neighbors of C are biconnected in H_1. Since each blue edge connects two neighbors of C, the claim holds. Consider next the case that C separates its tree neighbors C_1 and C_2 in H_1. Note that there are two vertex-disjoint paths in H_1 between C and each of its non-tree neighbors C_1: one path consists of the non-tree edge (C, C_1), and the other path is $\pi_{T_1}(C, C_1)$. Thus C_1 and C's tree neighbor on $\pi_{T_1}(C, C_1)$ are biconnected as well, i.e., the claim holds for each blue edge.

This implies that if the representative of two neighbors of C are connected by a colored path in $I(C)$, then the neighbors of C are biconnected in H_1. If two neighbors are represented by the same node in $I(C)$, then they are biconnected in H_1 by the definition of a c-node.

Next we show that if two neighbors of C are biconnected in H_1 and represented by two different nodes in $I(C)$, then these representatives in $I(C)$ are connected by a colored path. Each non-tree neighbor C_1 is biconnected with the tree neighbor of C

[4]If C has a non-tree neighbor, then C has tree degree at most 2.

[5]Note that C_1 and C are biconnected in H_1. Thus this is equivalent to requiring that for each non-tree neighbor C_1 of C there exists a blue edge (d_1, d_2), where d_2 represents a tree neighbor of C that is biconnected to C_1 in H_1 and d_1 represents C_1.

on $\pi_{T_1}(C, C_1)$ and is connected to this tree neighbor by a colored path (of length at most two). Thus it suffices to show the claim for two tree neighbors of C. But for two tree neighbors the claim holds by definition. ∎

LEMMA 2.12. *Let C be a cluster and let u and v be two nodes of $V(C)$. If s_C does not separate u and v in G, then u and v are biconnected in $I(C)$ iff they are biconnected in G.*

Proof. Assume first that u and v are biconnected in $I(C)$ but are separated by a node x in G. Note that x must belong to $V(C)$. Furthermore, x must be represented by at least two nodes in $I(C)$. However, each node of G is represented by at most one node in $I(C)$, except for s_C. Thus, $x = s_C$, which leads to a contradiction.

Assume next that u and v are biconnected in G, but are separated by a node y in $I(C)$. Since u and v are connected by a tree path whose (internal) nodes all belong to C, no b-node or c-node can separate u and v in $I(C)$. Thus, y must be an a-node.

Consider the path P between u and v in G that does not contain y. Let $\tilde{P}$ be the path created from P by (1) extending P to a path in G', (2) contracting all intracluster edges of P except for the nondashed intracluster edges of C, and (3) by labeling the resulting nodes of $\tilde{P}$ with their clusters of G'.

Every edge of $\tilde{P}$ either is connecting two clusters or is incident to a vertex with a representative in C. We show that $\tilde{P}$ induces a path without y in $I(C)$ connecting u and v. We split $\tilde{P}$ into subpaths. Each subpath either

(1) connects two neighbors of C and does not contain other neighbors of C or vertices with representatives in C, or

(2) is one edge connecting two vertices with representatives in C, or

(3) is a solid edge connecting a vertex with a representative in C with a neighbor of C, or

(4) is a dashed edge connecting a vertex with a representative in C with a neighbor of C.

By Lemma 2.11 the endpoints of the type-(1) subpaths are connected by a colored path in $I(C)$. By definition type-(2), (3), or (4) subpaths are contained in $I(C)$. Thus $\tilde{P}$ induces a path without y from u to v in $I(C)$. Thus we have a contradiction. ∎

LEMMA 2.13. *For each cluster C,*

$$|I(C)| = O(|C|).$$

Proof. Obviously, there are $O(|C|)$ a-nodes and $O(|C|)$ edges incident to them in $I(C)$. Each b-node or c-node in $I(C)$ can be charged to one of the edges that connects the b-node or c-node to a node in C. Thus, there are $O(|C|)$ b- or c-nodes. Since the number of colored edges is linear in the number of b- and c-nodes, it follows that $|I(C)| = O(|C|)$. ∎

We will need the following fact when bounding the time of updates.

LEMMA 2.14. *Let $C_1, \ldots, C_l$ be a set of clusters with the same ancestor. Then*

$$\sum_{i=1}^{l} |I(C_i)| = O(k).$$

Proof. By Lemma 2.13,

$$\sum_{i=1}^{l} |I(C_i)| = \sum_{i=1}^{l} O(|C_i|).$$

Let A be the ancestor of the clusters $C_1, \ldots, C_l$. Recall that either (1) each C_i contains the representative of a vertex and that representative or the (unexpanded) origin of the representative also belonged to A, or (2) C contains only representatives of the shared vertex s of A, all these representatives were created after the last rebuild, and $A = C_s$.

The number of clusters fulfilling (1) is bounded by the number of nodes of G' in A. The total number of clusters fulfilling (2) is bounded by the number of updates since the last rebuild, which is m/k.

Thus,

$$\sum_{i=1}^{l} O(|C_i|) \leq O(|\{v, v \text{ is a node of } G' \text{ in } A\}| + m/k) = O(k). \qquad \square$$

In [11][6] a *cluster data structure* for $I(C)$ is given so that

(1) building the data structure takes time $O(|C|)$, provided that the b-nodes, the c-nodes, and the red and blue edges are given;

(2) changing one or all of the colored edges takes time linear in their total number, provided the new colored edges are given;[7]

(3) testing whether two vertices of $V(C)$ that are not separated by s_C are biconnected in $I(C)$ takes constant time.

We keep as data structure

(CG1) for each cluster C a *cluster data structure* for $I(C)$;

(CG2) for each cluster C the adjacency lists of the graph $I(C)$ with the two occurrences of edges pointing at each other;

(CG3) for each cluster C, a list of pointers to the b- or c-nodes representing C in $I(C)'$ for each neighbor C'; for each b-node in $I(C')$, a pointer back to C, and for each c-node in $I(C')$, a set of pointers to the clusters represented by the c-node.

Note that given the b-nodes and c-nodes of $I(C)$, the red and blue edges of $I(C)$ can be determined in time linear in their number using the data structure (HL3) and (HL4) for H_1. This leads to the following lemma.

LEMMA 2.15. *Let C be a cluster. There exists a cluster data structure for $I(C)$ such that*

(1) *building the data structure takes time $O(|C|)$, provided that the b-nodes and the c-nodes are given;*

(2) *changing one or all of the colored edges takes time linear in their total number, given the b-nodes and c-nodes;*

(3) *testing whether two vertices of $V(C)$ that are not separated by s_C are biconnected in $I(C)$ takes constant time.*

Note: We will use the same data structure and the same update algorithm in section 2.6 for one class of shared graphs. There the same problem has to be solved in

[6]Lemma 4.6 of [11] states the result; section 4.1.2 of [11] describes the data structure. In the notation of [11], $G_3(C)$ is identical to $I(C)$ except that a non-tree neighbor C_1 of C *always* has a blue edge (C_1, C_2) to the tree neighbor C_2 of C that lies on $\pi_{T_1}(C, C_1)$, even if C is not an articulation point. Furthermore, $G_2(C) = G_3(C) \setminus \{\text{red edges}\}$, $C_T = V(C)$, and the *artificial edges* of [11] are identical to the colored edges of $I(C)$.

[7]In [11] changing a red edge actually takes no time during an update: the existence of a red edge is not recorded during an update but is checked during queries (by asking a biconnectivity query in H_1). This is possible, since only one red edge exists in a cluster graph. Since we will use the same data structure also for one class of shared graphs, we treat red edges as blue edges in the data structure of [11].

H_2 instead of H_1. Since nodes in H_2 are not guaranteed to have bounded tree degree, we will not make use of this property of H_1 in our update algorithm.

2.5.1. Updates. We show in this section that it takes amortized time $O(k)$ to update the data structures for all cluster graphs after an edge insertion or deletion in G. The major difficulty is to maintain the b-nodes and c-nodes of each cluster graph. Once it has been determined how they change, it will be quite straightforward to update data structures (CG1)–(CG3). Let C be a cluster. A c-node of $I(C)$ has to be partitioned either (i) because C is split or (ii) because conditions (a) or (b) of a c-node are no longer fulfilled for the c-node. We call the latter kind of update a *split by condition violation (CV-split)* of a c-node. We say a CV-split *occurs* at a node C of H_1 if one of the c-nodes of C is CV-split.

Section 2.8 presents a data structure that given an update operation decides which if any c-nodes have to be CV-split and for each CV-split c-node it returns the set of clusters forming each new c-node. In section 2.9 we show that during m/k update operations CV-splits occur at $O(m/k)$ different nodes of H_1. When a CV-split occurs at a node C of H_1 in a cluster graph, all data structures for the cluster graph of C are rebuilt from scratch in time $O(k)$. Thus the CV-splits add an amortized cost of $O(k)$ to the time per update. We describe below the remaining work that is necessary after an update to maintain the cluster graphs.

Insertion. Let u' and v' be the (potentially newly added) representatives that are incident to the newly inserted edge (u, v). The insertion affects the cluster graphs as follows:

(1) The clusters $C_{u'}$ and $C_{v'}$ might become neighbors because of the edge insertion and they also might be split while rebalancing the relaxed partition. In either case some b-nodes and/or c-nodes in $I(C_{u'})$ and $I(C_{v'})$ change. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) For a cluster C' incident to a split cluster the b-node representing the split cluster might have to be partitioned into a constant number of b-nodes. If this happens the data structures for $I(C')$ are rebuilt from scratch.

(3) Nodes on $\pi_{T_1}(C_{u'}, C_{v'})$ that separated $C_{u'}$ and $C_{v'}$ in H_1 before the insertion no longer separate $C_{u'}$ and $C_{v'}$. For each such node, a suitable red edge is added to the cluster graph, without rebuilding the cluster graph from scratch.

We next discuss each case in detail. (1) If $C_{u'}$ and $C_{v'}$ become neighbors because of the edge insertion, then a new b- or c-node representing $C_{v'}$, resp., $C_{u'}$, has to be added to $I(C_{u'})$, resp., $I(C_{v'})$. Consider the split of a cluster C into a constant number of clusters C_j. Then each c-node in $I(C_j)$ represents only one cluster by part (d) of the definition of a c-node. Thus, for each neighbor of C_j simply test whether it has the same ancestor as C_j. If yes, it is represented by a b-node in $I(C_j)$, otherwise it is represented by a c-node.

(2) We describe next how the changes in the b-nodes of $I(C')$ are determined when cluster C is split into a constant number of nodes C_j. Bucketsort the edges incident to C in lexicographic order of its two endpoints in the updated graph H_1 (using (HL6)). Each neighbor of C with the same ancestor as C that is incident to edges from $l > 1$ different buckets (i.e., new clusters) of its array receives l new b-nodes representing these new clusters, discards the b-node of C, and keeps all the other old b-nodes.

(3) The articulation points on $\pi_{T_1}(C_{u'}, C_{v'})$ are found by testing in constant time each of the clusters on $\pi_{T_1}(C_{u'}, C_{v'})$ using data structure (HL4) for H_1 (before the update). Then a red edge between its tree neighbors on $\pi_{T_1}(C_{u'}, C_{v'})$ is added to each

articulation point.

We next show that this takes total time $O(k)$. In (1) it takes constant time to test whether a b-node or a c-node has to be added for the neighbor of a cluster. Each test takes constant time using (HL7). By Lemma 2.10 there are $O(k)$ many such tests. Building the data structure (CG1) for a constant number of clusters takes time $O(k)$ by Lemma 2.15; building (CG2) and (CG3) takes $O(k)$ as well.

In (2) there are $O(k)$ edges to bucketsort. If each bucket with a count larger than 1 is put in a separate list, then all new b-nodes can be determined in time $O(k)$, and (CG3) can be updated accordingly. By Lemma 2.10 the total size of all the cluster graphs rebuilt in (2) is $O(k)$. Thus building (CG1) and (CG2) for all of them takes total time $O(k)$ by Lemma 2.15.

Each test in (3) takes constant time, and adding the red edge for a former articulation point C' in H_1 takes time linear in the degree of C'. By Fact 2.1 the H_1-degree of all articulation points on $\pi_{T_1}(C_{u'}, C_{v'})$ sums to $O(m/k)$.

Deletion of a non-tree edge. The deletion of a non-tree edge does not change the spanning tree and does not split a cluster (Lemma 2.5). Thus, the deletion of a non-tree edge affects the cluster graph as follows:

(1) The clusters $C_{u'}$ and $C_{v'}$ might no longer be neighbors in H_1. In this case the corresponding b-node or c-node has to be removed from $I(C_{v'})$, resp., $I(C_{u'})$. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) A red edge has to be removed and the blue edges have to be updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{u'}, C_{v'})$ in the updated graph H_1.

We next discuss each case in detail: in case (1) we simply need to test whether the edge $(C_{u'}, C_{v'})$ was removed from H_1. If so, and if $C_{u'}$ and $C_{v'}$ have the same ancestor, then the corresponding b-nodes are removed from $I(C_{v'})$ and $I(C_{u'})$. Otherwise, we need to test whether the c-node representing $C_{u'}$ in $I(C_{v'})$ represents further clusters. If not, then the c-node is removed. We proceed in the same way for the c-node representing $C_{v'}$ in $I(C_{u'})$.

In case (2) we determine each new articulation point C on $\pi_{T_1}(C_{u'}, C_{v'})$ by testing in constant time each of the clusters on $\pi_{T_1}(C_{u'}, C_{v'})$ using data structure (HL4) for H_1 (after the update). Then we remove all old red and blue edges from $I(C)$. Using (HL3) we determine for each neighbor C' of C the tree neighbor C'' of C such that C'' lies on $\pi_{T_1}(C, C')$, and connect C' by a blue edge with C''. Finally we determine the biconnected components of the tree neighbors of C using a *components?(C)* operation in (HL4) and add the suitable red edges.

We next show that this takes total time $O(k)$. In case (1) testing and rebuilding the data structures takes time $O(k)$ by the same argument as for insertions. In case (2) finding all articulation points takes time $O(m/k)$. Determining the new red and blue edges takes time linear in the degree of each articulation point. By Fact 2.1 the total cost for all articulation points is $O(m/k)$.

Deletion of a tree edge. Let u' and v' be the representatives that are incident to the deleted edge (u, v) and let x' and y' be the representatives that are incident to the new tree edge (x, y), if it exists. By Lemma 2.5, $C_{u'}$, $C_{v'}$, $C_{x'}$, and $C_{y'}$ are the only clusters that might be split.

The cluster graphs are affected as follows: (1) The clusters $C_{u'}$ and $C_{v'}$ might no longer be neighbors in H_1. In this case the corresponding b-node or c-node has to be removed from $I(C_{v'})$, resp., $I(C_{u'})$. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) The clusters $C_{u'}$, $C_{v'}$, $C_{x'}$, and $C_{y'}$ might be split. The data structures (CG1)–(CG3) are rebuilt from scratch for each cluster created by the splits.

(3) For a cluster C' incident to a split cluster the b-node representing the split cluster might have to be partitioned into a constant number of b-nodes. If this happens the data structures for $I(C')$ are rebuilt from scratch.

(4) A red edge has to be removed and the blue edges have to be updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{x'}, C_{y'})$ in the updated graph H_1.

We implement (1), (2), (3), and (4) as in the cases of edge insertions or deletions of non-tree edges. Thus, the same arguments as above show that all this can be implemented in time $O(k)$.

We summarize the section with the following theorem.

THEOREM 2.16. *The given data structure*

(1) *tests in constant time whether two vertices u and v of $V(C)$ for a cluster C that are not separated by s_C are biconnected in G,*

(2) *can be updated in amortized time $O(k)$ after each update in G, and*

(3) *can be built in time $O(m)$.*

2.6. Shared graphs. We maintain a shared graph $G(s)$ for every shared vertex s. Given a shared vertex s and two of its tree neighbors x and y, the shared graph of s is used to test in constant time whether s is an articulation point separating x and y in G.

Let C_s be the node of H_2 representing s, i.e., it represents the nodes in all s-clusters. Let $\mathcal{V}(C_s) = \{v; v \in G$, and v is represented by $C_s\}$. Shared graphs are used to test whether a pair of two special vertices of G that either are represented by the same node of H_2 or are incident to the same node of H_2 (to be precise, two tree neighbors of s) are biconnected in G. Note that cluster graphs solve this problem in H_1: a cluster graph tests whether two vertices of G that are represented by or are incident to the same node of H_1 are biconnected in G, under the additional condition that no dashed edge is incident to this node of H_1. (For nodes of H_1 that are incident to a dashed edge only a restricted version of the problem is solved.) Since there are no dashed edges in H_2, we simply can define shared graphs analogously to cluster graphs and use the data structure for cluster graphs also for shared graphs. However, it is possible that $|C_s| = \Theta(m)$ and, thus, rebuilding the data structure from scratch can take time $\Theta(m)$.

This leads to the following definition. Let us call a shared vertex s *new* if s became shared by a cluster split after the last rebuild, and let it be called *old* otherwise (i.e., if it became a shared vertex during the last rebuild). Note that if s is new, then $|\mathcal{V}(C_s)| = O(k)$ by Lemma 2.10, and, thus, a solution analogous to cluster graphs is efficient.

For old shared vertices we use a new technique, which exploits the fact that the tree neighbors x and y of s are biconnected iff x and y are connected in $G \setminus s$. Thus, we maintain a "compressed" version of $G \setminus s$ in which we ask *connectivity* queries. The shared graph will be stored in a dynamic connectivity data structure. Note that there are $O(m/k) = O(k)$ many shared vertices, which implies we have to maintain $O(k)$ many shared graphs.

2.6.1. Shared graphs for new shared vertices. Let s be a new shared vertex represented by node C_s in H_2, and let A_s be the ancestor of C_s. The *shared graph* $G(s)$ contains as nodes

(1) a node, called *a-node*, for each vertex in $\mathcal{V}(C_s)$,

(2) one node, called *b-node*, for each neighbor of C_s with ancestor A_s,

(3) one node, called *c-node*, for each maximal set X of nodes of H_2 such that (a) every cluster $C' \in X$ is a neighbor of C_s, (b) all nodes in X are connected in $H_2 \setminus C_s$, (c) all nodes in X have the same ancestor which is different from A_s, (d) at the creation of C_s, the set X contains only one node, and (e) at each previous point in time since the creation of C_s all nodes of H_2 that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node in $G(s)$.

Note that for each neighbor C' of C_s there exists a unique node in $G(s)$ representing C' and potentially other clusters.

The graph $G(s)$ contains the following edges:

(1) All edges between two vertices of $\mathcal{V}(C_s)$ belong to $G(s)$.

(2) For each edge (u, v), where u belongs to $\mathcal{V}(C_s)$, v does not belong to $\mathcal{V}(C_s)$, and (u', v') is the corresponding edge in G, there is an edge (u, d), where d is the b- or c-node representing $C_{v'}$ in $G(s)$.

(3) All b- or c-nodes representing tree neighbors of C that are biconnected in H_2 are connected by a tree of *red* edges.

(4) For each non-tree neighbor C_1 of C, $G(s)$ contains a *blue* edge (d_1, d_2), where d_2 represents a tree neighbor of C that is biconnected to C_1 in H_2, and d_1 represents C_1.

Note that for each non-tree neighbor C_1 of C there always exists a tree neighbor of C that is biconnected to C_1 in H_2—the tree neighbor of C that lies on $\pi_{T_2}(C, C_1)$ always is biconnected to C_1.

Since all clusters sharing s have the same ancestor, Lemma 2.10 shows that $|G(s)| = O(k)$.

A tree neighbor of s either belongs to $\mathcal{V}(C_s)$ and is represented by an a-node, or does not belong of $\mathcal{V}(C_s)$ and is represented by a b- or c-node. We need to show the following lemma.

LEMMA 2.17. *Let x and y be two tree neighbors of a new shared vertex s. Then (the representative of) x and y are biconnected in $G(s)$ iff x and y are biconnected in G.*

Proof. Note that $G(s)$ can be created by contracting edges in G. Thus, biconnectivity in $G(s)$ implies biconnectivity in G.

Vertices x and y are connected by a tree path $(u, s), (s, v)$. Thus s is the only node that could be an articulation point separating x and y. Contracting edges not incident to s cannot make s into an articulation point separating x and y. Hence, biconnectivity in G implies biconnectivity in $G(s)$. □

We use the same data structure as for cluster graphs to store shared graphs for new shared vertices. Given the b- and c-nodes the red edges can be found in time linear in their number using the data structure (HL4) for H_2. We determine the blue edges of $G(s)$ by connecting each non-tree neighbor C_1 of a node C to the tree neighbor of C on $\pi_{T_2}(C, C_1)$. Using the data structure (HL3) for H_2 this takes time linear in the number of blue edges times $O(\log n)$. Using the data structure of [11] results in the following lemma.

LEMMA 2.18. *Let s be a shared vertex represented by the node C of H_2. Then there exists a data structure for the shared graph of s such that*

(1) *building the data structure takes time $O(|C_s| + (m/k) \log n)$, provided that the b-nodes and the c-nodes are given;*

(2) *changing one or all of the colored edges in the data structure takes time linear in their total number times $O(\log n)$, given the b-nodes and c-nodes;*

(3) *testing whether two vertices of* $\mathcal{V}(C_s)$ *are biconnected in* $G(s)$ *takes constant time.*

These data structures are updated with the algorithm of section 2.5.1 with H_1 replaced by H_2. Since the test in (HL3) takes time $O(\log n)$, the amortized time per operation is $O(k + (m/k)\log n)$.

2.6.2. Shared graphs for old shared vertices. Let s be an old shared vertex and let C_s be the node of H_2 representing s. We cannot use the data structure of the previous section for the shared graph of s since rebuilding the data structure from scratch would take time $\Omega(|C_s|)$, which might be $\Theta(m)$. Still we use an approach similar to the one in the previous section but avoid rebuilds from scratch for the whole data structure.

We will exploit the following fact: Condition (C6) of a relaxed partition guarantees that at any time in a phase $O(m/k)$ vertices that do not belong to C_s are incident to a vertex of C_s. Only the $O(m/k)$ shared vertices and the $O(m/k)$ endpoints of edges inserted in the phase can be neighbors of a vertex in C_s and not belong to C_s.

The graphs $G(s)$. Let s be an old shared vertex,
(1) let C_s be the node representing s in H_2,
(2) let A_s be the ancestor of C_s in H_2, and
(3) let $\mathcal{V}_s = \{v, v \in G$ and v is represented by a node of H_2 with ancestor $A_s\}$.

Note that at the beginning of a phase $\mathcal{V}_s$ consists exactly of all the vertices in C_s. Later in the phase, the vertices of G in C_s are all contained in $\mathcal{V}_s$, but $\mathcal{V}_s$ can contain additional vertices.

The graph $G(s)$ contains as vertices
(1) a node, called *a-node*, for each vertex in $\mathcal{V}_s$, except for s, and
(2) a node, called *d-node*, for each vertex of G that does not belong to $\mathcal{V}_s$ but is a neighbor of a vertex in $\mathcal{V}_s$.
The graph $G(s)$ contains as edges every edge between two a-nodes or between an a-node and a d-node.

LEMMA 2.19. *Let s be an old shared vertex. The number of d-nodes in $G(s)$ is $O(m/k)$.*

Proof. By condition (C6) of a relaxed partition the number of d-nodes at the beginning of a phase connected to a vertex of $\mathcal{V}_s$ by non-tree edges is $O(m/k)$. Additionally there are $O(m/k)$ tree edges. During the phase only the endpoints of edges inserted during the phase can become neighbors of vertices in $\mathcal{V}_s$. Thus, the lemma follows. ☐

Data structure:

(S1) We store $G(s)$ in a fully dynamic connectivity data structure. This data structure allows us to execute the following operations:
(1) *insert(u,v)/delete(u,v):* Insert or delete the edge (u, v) in time $O(\sqrt{m'})$, where m' is the number of edges in $G(s)$.
(2) *insert(u):* Insert the degree-0 vertex u in time $O(\sqrt{m'})$, where m' is the number of edges in $G(s)$.
(3) *connected?(u,v):* Test whether u and v are connected in constant time.
(4) *component?(u):* Return the connected component of u in constant time.

(S2a) We keep for each connected component of $G(s)$ a list of all the d-nodes that belong to it.

(S2b) We keep for each connected component of $G(s)$ a list of all neighbors of C_s with ancestor A_s whose vertices belong to the connected component. We also store for each node of H_2 its position in the at most one such list to which it belongs.

LEMMA 2.20. *The data structures for $G(s)$ for all old shared vertices s can be built in time linear in its size at the beginning of a phase and can be updated in time $O(\sqrt{m})$ after an edge insertion or deletion in G.*

Proof. At the beginning of a phase the vertices of $G(s)$ are simply the vertices in C_s and their neighbors and are given by (HL1) and (G1). The edges are given by (G1). Building (S1) and (S2a) takes linear time; (S2b) is an empty list.

Each edge insertion or deletion in G affects the data structure for $G(s)$ of at most one old shared vertex s since the sets $\mathcal{V}_s$ are vertex-disjoint for different old shared vertices s. Since the graph $G(s)$ consists of at most m edges, its fully dynamic connectivity data structure can be updated in time $O(\sqrt{m})$.

In case of the deletion of the edge (u', v') we test after the removal of the edge from $G(s)$ whether u' and v' are still connected. If not, we test each d-node in the old connected component of u' and v' whether it is either connected to u' or to v'. In this way we construct the list of d-nodes for the two new connected components of $G(s)$. By Lemma 2.19 this requires $O(m/k)$ tests. In the same way we split the (S2b) list of the old connected component to create the (S2b) lists of the two new connected components and update the corresponding positions at the nodes of H_2.

In case of an edge insertion we insert a new d-node if one of the endpoints of the new edge does not belong to $G(s)$. Then we test whether the endpoints of the edge belonged to the same connected component before the insertion. If they did not, we combine their lists of d-nodes and their lists of neighbors of C_s. Note that these lists were disjoint before the combination since otherwise the two connected components would have shared a vertex.

Additionally we determine for each cluster split by an update operation whether its node in H_2 is split as well and whether a position in an (S2b) list is stored at the node. If so, we update the entry in the (S2b) list accordingly and store the appropriate positions at the new nodes. Since only a constant number of nodes is split by Lemma 2.5, this takes constant time.

Thus, the total time of an update is $O(\sqrt{m})$. ∎

Note that each tree neighbor x of s either belongs to C_s or is represented by a node of H_2 that is a neighbor of C_s. In the latter case we call the neighbor of C_s representing x the *x-neighbor* of C_s. In the former case we determine a node that is a neighbor of C_s and connected to x in $G(s)$ as follows: Using (S1) and (S2b) we can determine in constant time a neighbor of C_s with ancestor A_s that is in the connected component of x. If no such node exists, then using (S1) and (S2a) we can determine in constant time a d-node, and using (HL1) a node of H_2, that is connected to x in $G(s)$ if such a d-node exists. Note that in the latter case the d-node is connected to a node in C_s, i.e., the node of H_2 containing the d-node is a neighbor of C_s, since no node with ancestor A_s belongs to the connected component of x. In either case the determined neighbor of C_s is called an *x-neighbor* of C_s.

The graphs $\tilde{G}(s)$. The intuition for the graph $\tilde{G}(s)$ is as follows: Initially, and whenever $\tilde{G}(s)$ is rebuilt, each neighbor of C_s is represented by a vertex in $\tilde{G}(s)$. If this neighbor is split into two nodes C_1 and C_2 of H_2, then the corresponding vertex of $\tilde{G}(s)$ is updated (and the whole graph $\tilde{G}(s)$ is rebuilt) only if C_1 and C_2 are not connected in $H_2 \setminus C_s$ after the update. If C_1 and C_2 are connected in $H_2 \setminus C_s$ after the update, then the vertex of $\tilde{G}(s)$ is not modified, but represents now both nodes of H_2. This would guarantee that after an edge update we only have to update the graph $\tilde{G}(s)$ of an old shared vertex s if an edge of $G(s)$ was modified (i.e., $\mathcal{V}_s$ contains an endpoint of the updated edge) or the connected components of $H_2 \setminus C_s$ are modified by the update. However, to bound the number of these graphs using the amortization lemma of section 2.9 we need to treat nodes with ancestor A_s in a special way.

We formalize this as follows: The graph $\tilde{G}(s)$ contains as vertices

(1) one node, called *e-node*, for each maximal set X of nodes of $H_2 \setminus C_s$ such that (a) every node in X is a neighbor of C_s, (b) all nodes in X are connected in $H_2 \setminus C_s$, (c) all nodes in X have the same ancestor which is different from A_s, (d) at the creation of C_s, the set X contains only one element, and (e) the vertices in $\cup_{C' \in X} C'$ used to belong to the same node C_{old} of H_2 and since the split of C_{old}, the graph $G(s)$ was not modified and the connected components of $H_2 \setminus C_s$ did not change;

(2) one node, called *b-node*, for each neighbor of C_s with ancestor A_s.

Each vertex in $\tilde{G}(s)$ *represents* the nodes in the set X and thus also the vertices of G contained in these nodes.

The graph $\tilde{G}(s)$ contains the following edges.

(1) All vertices of $\tilde{G}(s)$ that contain vertices that are connected in $G(s)$ are connected by a tree of *yellow* edges in $\tilde{G}(s)$.

(2) All vertices of $\tilde{G}(s)$ whose nodes are connected in $H_2 \setminus C_s$ are connected by a tree of *green* edges in $\tilde{G}(s)$.

Let x be a vertex of $\mathcal{V}_s$ and a tree neighbor of s. Note that by definition all nodes of $\tilde{G}(s)$ representing an x-neighbor are connected in $\tilde{G}(s)$ by yellow edges, i.e., belong to the same connected component of $\tilde{G}(s)$.

Data structure:

(S3) We store $\tilde{G}(s)$ in an adjacency list representation and label each node with its connected component.

(S4) We store for each node C of H_2 an array with one entry per old shared vertex. The array stores for each old shared vertex s the vertex in $\tilde{G}(s)$ that represents C in $\tilde{G}(s)$ if such a vertex exists and *null* otherwise.

The next lemma shows how to use $G(s)$ and $\tilde{G}(s)$ to test whether two neighbors of s are biconnected in G.

LEMMA 2.21. *Two tree neighbors x and y of s are biconnected in G iff*

(1) *either x and y are connected in $G(s)$, or*

(2) *the connected component of the vertices of $\tilde{G}(s)$ representing x-neighbors is identical to the connected component of the vertices of $\tilde{G}(s)$ representing y-neighbors.*

Proof. Each edge in $G(s)$ or $\tilde{G}(s)$ corresponds to a path in G that does not contain s, and each vertex $u \in \mathcal{V}_s \setminus \{s\}$ is either contained in the u-neighbor or connected to every u-neighbor by a path in $G(s)$ that does not contain s. Additionally connectivity of x and y in $G \setminus \{s\}$ implies biconnectivity in G. Thus, if x and y are connected in $G(s)$ or if a vertex of $\tilde{G}(s)$ representing an x-neighbor of C_s is connected with a vertex of $\tilde{G}(s)$ representing a y-neighbor of C_s in $\tilde{G}(s)$, then x and y are biconnected in G.

To show the other direction consider a path P in G that connects x and y and does not contain s. If all vertices of P belong to $\mathcal{V}_s \setminus \{s\}$, then x and y are connected in $G(s)$.

Otherwise, recall that all vertices of $G(s)$ representing an x-neighbor (resp., y-neighbor) form a connected component of $\tilde{G}(s)$. It follows that it suffices to show that one of the vertices of $\tilde{G}(s)$ representing an x-neighbor is connected in $\tilde{G}(s)$ to one of the vertices of $\tilde{G}(s)$ representing a y-neighbor. In this case P contains a d-node d_x that is connected to x by a path in $G(s)$. If the path from x to d_x contains only vertices of C_s (excluding d_x), then let b_x be the vertex of $\tilde{G}(s)$ representing d_x and let u_x be d_x. If the path from x to d_x contains vertices not in C_s, let b_x be the vertex of $\tilde{G}(s)$ representing the first such node u_x on the path. In either case b_x represents an x-neighbor. Define u_y and b_y in the same way.

Consider the subpath P' of P between the u_x and u_y. Partition P' into subpaths such that each subpath is a maximal sequence of edges such that either (a) each edge is incident to a vertex of C_s or (b) no edge is incident to a vertex of C_s. Note that the endpoints of each subpath belong to nodes in H_2 that are neighbors of C_s.

Since P does not contain s, each subpath fulfilling (a) corresponds to a path in $G(s)$. Thus the vertices of $\tilde{G}(s)$ representing the endpoints of the subpath are connected by a path of yellow edges in $\tilde{G}(s)$. Each subpath fulfilling (b) connects the nodes of H_2 containing the endpoints of the subpath by a path in $H_2 \setminus C_s$. Thus the vertices of $\tilde{G}(s)$ representing these nodes of H_2 are connected by a path of green edges in $\tilde{G}(s)$. It follows that b_x is connected to b_y in $\tilde{G}(s)$. □

The first condition is tested in constant time using (S1). To test the second condition we determine the x-neighbor and y-neighbor using (S1), (S2a), (S2b), and (HL1). We determine the vertices of $\tilde{G}(s)$ representing the x-neighbor and the y-neighbor using (S4) and then test their connected components in $\tilde{G}(s)$ using (S3). All this takes constant time.

Let $deg_2(C_s)$ be the degree of C_s in H_2.

LEMMA 2.22. *If only the green edges of* $H_2 \setminus C_s$ *change, then* $\tilde{G}(s)$ *and the data structures* (S3) *and* (S4) *can be updated in time* $O(deg_2(C_s) \log n)$.

Proof. Discard all old green edges. To compute the new green edges, map each neighbor of C_s to a tree neighbor of C_s in H_2 using (HL3) and determine the biconnected component of this tree neighbor using *blockid?*-queries in (HL4) for H_2. Then bucketsort the neighbors according to these biconnected components using (HL6). For each neighbor in a biconnected component determine its vertex in $\tilde{G}(s)$ using (S4) and connect it by a green edge to the vertex in $\tilde{G}(s)$ of the previous neighbor in the same biconnected component. Then compute a spanning forest of the green edges by performing a depth-first search on the graph of green edges, and discard all green edges not in the spanning tree. Finally recompute the connected components of $\tilde{G}(s)$. This takes time $O(\log n)$ per neighbor and constant time per green edge. Since the number of green edges is linear in the number of neighbors, the lemma follows. □

LEMMA 2.23. *Let s be an old shared vertex. At the beginning of a phase or whenever* $G(s)$ *has changed, the graph* $\tilde{G}(s)$ *can be constructed in time* $O((m/k) \log n)$.

Proof. Note first that the size of $\tilde{G}(s)$ is linear in its number of vertices of $\tilde{G}(s)$, which is bounded by $deg_2(C_s)$.

The vertices of $\tilde{G}(s)$ are the neighbors of C_s and are given by (HL2). To determine the yellow edges, process the d-nodes belonging to the same connected component of $G(s)$ as follows. Map each d-node in (S2a) for this connected component to its node in H_2 using (HL1). Append to this list the neighbors of C_s stored in (S2b) for this connected component. Now process each node C in this list as follows: Determine the vertex of $\tilde{G}(s)$ representing C and connect it by a yellow edge to the vertex of $\tilde{G}(s)$ representing the previous node on the list. This takes time linear in the length of the list. Then compute a spanning tree of yellow edges by performing a depth-first search on the graph of yellow edges. Discard the yellow edges that do not belong to the spanning tree. This takes time linear in the number of yellow edges, which is linear in the length of the list. The length of the list is linear in the number of d-nodes in $G(s)$ in the connected component and the length of the (S2b) list for the connected component. By Lemma 2.19 there are $O(m/k)$ many d-nodes in $G(s)$, i.e., in the (S2a) lists for all connected components. Also there are $O(m/k)$ many neighbors, i.e., in the (S2b) lists for all connected components. Thus, the total time for determining all yellow edges in $\tilde{G}(s)$ is $O(m/k)$.

The green edges are computed in time $O((m/k)\log n)$ as in Lemma 2.22. ∎

Lemma 2.24. *Constructing the graphs $\tilde{G}(s)$ for all old shared vertices at the beginning of a phase takes time $O(m \log n)$. Building the data structures* (S3) *and* (S4) *for all s at the beginning of a phase takes time $O(m)$.*

Proof. By Lemma 2.23, constructing one graph $\tilde{G}(s)$ takes time $O((m/k)\log n)$, for a total of $O((m/k)^2 \log n) = O(m \log n)$.

When $\tilde{G}(s)$ is given, then building (S3) for s takes time linear in the size of $\tilde{G}(s)$. To build (S4) we allocate and initialize with *null* all the necessary arrays. This takes time $O((m/k)^2) = O(m)$. Then we process each graph $\tilde{G}(s)$ and set the entry for s in the array of node C of H_2 to the vertex of $\tilde{G}(s)$ representing C if it exists. ∎

Lemma 2.25. *The graphs $\tilde{G}(s)$ and the data structures* (S3) *and* (S4) *for all old shared vertices s can be updated in amortized time $O((m/k)\log n)$ after an edge insertion or deletion in G.*

Proof. We will show that after the insertion or deletion of the edge (u, v) in G the only old shared vertices s for which $\tilde{G}(s)$ has to be updated are the ones (1) where $\mathcal{V}_s$ contains u or v, (2) where a node of H_2 that is split by the update has ancestor A_s, or (3) where C_s is an articulation point on $\pi_{T_2}(C_u, C_v)$ before or after the current update. For type-(3) old shared vertices either (i) a vertex of $\tilde{G}(s)$ has to be split and the green and yellow edges have to be recomputed or (ii) only the green edges have to be updated. If a yellow edge has to be changed without splitting a vertex, then $\mathcal{V}_s$ must contain u or v, i.e., it is a type-(1) old shared vertex.

To guarantee that all graphs that have to be changed are indeed updated we use the following update algorithm:

(1) Rebuild from scratch the graphs $\tilde{G}(s)$ for all old shared vertices s where $\mathcal{V}_s$ contains either u or v (type (1) above), where C_s has the same ancestor as a split node in H_2 (type (2) above), or where a vertex of $\tilde{G}(s)$ is split (type (3(i)) above).

(2) Update the green edges in the graph $\tilde{G}(s)$ for the remaining old shared vertices where C_s is an articulation point on $\pi_{T_2}(C_u, C_v)$ before or after the update (case (3(ii)) above).

To find all type-(2) old shared vertices, we determine for each split node of H_2 its ancestor A using (HL7) and test for each old shared vertex s whether the ancestor of C_s equals A. The old shared vertices for which this test returns true are the type-(2) old shared vertices.

We discuss next how to determine all type-(3(i)) and type-(3(ii)) old shared vertices. There are two kinds of vertices in $\tilde{G}(s)$, e-nodes and b-nodes. If a b-nodes has to be split, then s is also a type-(2) old shared vertex and will be updated correctly. Thus, it suffices to determine all type-(3(i)) old shared vertices s where an e-nodes has to be split. In section 2.8 we give a data structure that determines all e-nodes that have to be split in time $O((m/k)\log n)$. Furthermore, we show in section 2.9 that only an amortized constant number of e-nodes has to be split. We determine all articulation points on $\pi_{T_2}(C_u, C_v)$ using (HL4). This gives all type-(3) old shared vertices. We remove the ones that are of type (2). In the remaining set the ones that contain an e-node that has to be split are the type-(3(i)) but not type-(2) old shared vertices; the rest are the type-(3(ii)) old shared vertices.

Whenever $\tilde{G}(s)$ is rebuilt, we also rebuild (S3) from scratch. The data structure (S4) is updated as follows:

(1) For each new node that is created by a split of an old node during the phase, the array of the old node is copied to create the array for the new node.

(2) For each old shared vertex s where $\tilde{G}(s)$ is rebuilt, the entry of s in the array of every neighbor of C_s is replaced by the vertex representing the neighbor in $\tilde{G}(s)$.

Lemma 2.23 shows that for type-(1), type-(2), and type-(3(i)) old shared vertices the graph $\tilde{G}(s)$ can be constructed in time $O((m/k)\log n)$. Lemma 2.22 shows that for type-(3(ii)) old shared vertices the green edges can be updated in time $O(deg_2(C_s)\log n)$. Determining all old shared vertices s such that a vertex of $\tilde{G}(s)$ has to be split or that $A_s = A$ takes time $O(m/k)$ per split cluster. By Lemma 2.5 there are a constant number of split clusters per update, i.e., the total time spent to determine type-(2) and type-(3(i)) old shared vertices is $O(m/k)$. Determining all articulation points and all type-(3(ii)) old shared vertices takes time $O(m/k)$. Building (S3) takes time $O(deg_2(C_s))$. The first type of update of (S4) creates a new array and takes time $O(m/k)$; the second type takes time $O(deg_2(C_s))$. Thus, updating the graphs $\tilde{G}(s)$ and the data structure (S3) for type-(1), (2), and (3(i)) old shared vertices takes time $(O(m/k)\log n)$ each; updating the graphs $\tilde{G}(s)$ and the data structure (S3) for type-(3(ii)) old shared vertices takes time $O(deg_2(C_s)\log n)$ each.

We show next that there are only an amortized constant number of type-(1), (2), and (3(i)) old shared vertices whose graphs $\tilde{G}(s)$ have to be updated, for a total time of $O((m/k)\log n)$ to update them. There are at most two type-(1) old shared vertices. There is at most one type-(2) old shared vertex per split node C of H_2, since the fact that C has ancestor A_s implies that the vertices of C belong to $\mathcal{V}_s$ and the sets $\mathcal{V}_s$ are disjoint for different old shared nodes. By Lemma 2.43 and Lemma 2.44 there are an amortized constant number of type-(3(i)) old shared vertices, where an e-node has to be split.

By Fact 2.1 the degree of all articulation points on a path in H_2 sums to $O(m/k)$. Thus, $deg_2(C_s)$ for all type-(3(ii)) old shared vertices sums to $O(m/k)$. Hence, updating all graphs $\tilde{G}(s)$ and building (S3) for the type-(3(ii)) old shared vertices takes time $O((m/k)\log n)$. Finally, a new array in (S4) is created for a constant number of new nodes of H_2. Summing all the cost gives a total time of $O((m/k)\log n)$ to update all graphs $\tilde{G}(s)$ and their data structures (S3) and (S4).

We still need to show the above claim about which graphs $\tilde{G}(s)$ have to be changed and how. A graph $\tilde{G}(s)$ has to be updated if (A) a vertex, (B) a yellow edge, or (C) a green edge has to be changed.

(A) If a b-node of $\tilde{G}(s)$ has to be changed, then a node in H_2 with ancestor A_s was split (case (2) above). If an e-node of $\tilde{G}(s)$ has to be changed, then either condition (a), (b), or (c) of an e-node does not hold anymore. If (a) does not hold anymore, then the current update deleted an edge of $G(s)$, i.e., $\mathcal{V}_s$ contains an endpoint of the updated edge (case (1) above). If (b) does not hold anymore, then C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above). If (c) does not hold, then again either the current update modified an edge of $G(s)$ (case (1) above), or C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above).

(B) If a yellow edge but no vertex of $\tilde{G}(s)$ has to be changed, then an update occurred in the graph $G(s)$. It follows that $\mathcal{V}_s$ contains at least one of the endpoints of the updated edge (case (1) above).

(C) If a green edge has to be changed, then C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above).

This completes the proof of the lemma. ◻

We summarize the section with the following theorem.

THEOREM 2.26. *The given data structure*

(1) *tests in constant time whether two tree neighbors x and y of a shared vertex s are biconnected in G,*

(2) *can be updated in amortized time $O((m/k)\log n + \sqrt{m})$ after each update in G, and*

(3) *can be built in time $O(m \log n)$.*

2.7. The high-level graphs. In this section we give the details of the high-level graph data structures and explain how they are updated. For (HL1), (HL2), (HL5), (HL6), and (HL7) the details are given in section 2.4 and it is obvious how to update them in time $O(k)$ per update in G, provided the change in the cluster partition is known. The description in section 2.2 gives an $O(k)$-time algorithm to update the cluster partition. Thus, we concentrate in this section on the details of (HL3) and (HL4).

2.7.1. The data structure (HL3). Consider the graph H_i. Given the two nodes C and C' of H_i we use (HL3) to find the tree neighbor of C on $\pi_{T_i}(C, C')$. For H_2 (HL3) consists of a dynamic tree data structure of T_2. To find the tree neighbor, root T_2 at C' and return the parent of C. Update (HL3) by executing link and cut operations whenever T_2 changes. It takes time $O(\log n)$ to test or update (HL3).

For H_1, (HL3) consists of a degree-k ET-tree data structure of T_1. To find the tree neighbor proceed as follows. For the node C', and the tree neighbors of C, label one of the leaves representing the node with the name of the node. Then traverse the ET-tree from all labeled leaves in lockstep, labeling each internal node with the concatenation of the label of its left child and the label of its right child. At the root the label incident to C''s label is the desired tree neighbor. To update the ET-tree, whenever T_1 changes, split and join the ET-tree accordingly. Since the ET-tree has $O(1)$ depth and C has $O(1)$ tree neighbors, a test takes $O(1)$ time. Each update takes time $O(k)$.

2.7.2. The data structure (HL4). Recall that (HL4) implements the following query operations in H_i:

(1) *biconnected?(C,C′,C″):* Given that nodes C' and C'' are both tree neighbors of a node C test whether C' and C'' are biconnected in H_i.

(2) *blockid?(C,C′):* Given that C and C' are tree neighbors in T_i, output the name of the biconnected component of H_i that contains both C and C'.

(3) *components?(C):* Output the tree neighbors of C in H_i grouped into biconnected components.

As we show below, *biconnected?* and *blockid?* take constant time, and *components?* takes time linear in the size of the output.

We say a node C of H_i is *avoidable on the tree path P* iff C and two of its tree neighbors, called D and D', belong to P and there is an edge in $H_i \setminus C$ between the subtree of $T_i \setminus C$ containing D and the subtree containing D'. Note that if C is avoidable, then C does not separate D and D'. However, if C does not separate D and D' it does not follow that C is avoidable.

To implement (HL4) we build a compressed graph $H_i(C)$ of $H_i \setminus C$ for each node C in H_2. Let C be a node in H_i. The graph $H_i(C)$ contains a node for each tree neighbor of C in H_i. There is an edge between two tree neighbors D and D' of C iff C is avoidable on $\pi_{T_i}(D, D')$.

We keep the following data structures. The second is needed to efficiently maintain the $H_i(C)$.

(HL4-1) For $i = 1, 2$, and for each node C we store $H_i(C)$ in a dynamic connectivity data structure.

(HL4-2) A 2-dimensional topology tree [5] of T and one ambivalent data structure [6] are maintained. They implement the following operations in time $O((m/k) \log n)$:

(1) *insert&return_avoidable(u,v):* Return all nodes on $\pi(C_u, C_v)$ that become avoidable on $\pi(C_u, C_v)$ by the insertion of edge (u, v), where C_u and C_v are the endpoints in H_i of the newly inserted edge.

(2) *delete&return_unavoidable(u,v):* Return all nodes on $\pi(C_u, C_v)$ that become unavoidable on $\pi(C_u, C_v)$ by the deletion of edge (u, v), where C_u and C_v are the endpoints in H_i of the newly deleted edge.

We show how to implement the operations of (HL4) using (HL4-1).

(1) *biconnected?(C,C′,C″)*: Return *connected?(C′, C″)* in $H_i(C)$.

(2) *blockid?(C,C′)*: Return *component?(C′)* in $H_i(C)$.

(3) *components?*: Return all connected components of $H_i(C)$, and for each connected component output all its nodes.

The correctness of this implementation is shown by the following lemma.

LEMMA 2.27. *Two tree neighbors D and D' of C in H_i are biconnected in H_i iff they are connected in $H_i(C)$.*

Proof. If D and D' are connected in $H_i(C)$, then every edge on the path connecting D and D' corresponds to a path in $H_i \setminus C$. Thus they are biconnected in H_i. If D and D' are biconnected in H_i, they are connected by a path in $H_i \setminus C$. Every edge on this path either lies in a subtree of $T_i \setminus C$ or connects two subtrees. The edges connecting two subtrees give a path in $H_i(C)$ connecting D and D'. □

Updates in $H_i(C)$. Consider an insertion of edge (u, v) in G and let C_u and C_v be the nodes in H_i incident to the edge. The graph $H_i(C)$ has to be modified only for the nodes that become avoidable on $\pi(C_u, C_v)$. These nodes can be found with one *insert&return_avoidable* operation in (HL4-2). Let C be such a node. Note that exactly one edge is added to $H_i(C)$, namely, the edge between the tree neighbors of C on $\pi(C_u, C_v)$.

An edge deletion in G is handled analogously.

To analyze the running time recall that the operation in (HL4-2) takes time $O((m/k) \log n)$. The update in a graph $H_i(C)$ takes time $O((deg_T(C))^{1/3} \log deg_T(C))$, where $deg_T(C)$ is the degree of C in T_i. Since $\sum_C deg_T(C) = O(k)$, the cost for updating all $H_i(C)$ is $O(k)$. This shows the following theorem.

THEOREM 2.28. *There exists a data structure that implements the operations biconnected?, blockid?, and components? in time linear in their output. The data structure can be updated in time $O(k + (m/k) \log n)$ after each update operation in G.*

2.7.3. The data structure (HL4-2).

We present now a data structure that implements *insert&return_avoidable* and *delete&return_unavoidable* in time $O((m/k) \log n)$.

We will actually show a slightly more general result: we give a data structure that can be updated in time $O(m/k)$ after each update in G and that can return all avoidable nodes on a tree path P in H_i in time $O((m/k) \log n)$. Obviously this data structure can be used to execute the above operations in time $O((m/k) \log n)$.

In this section we assume that T_1 is rooted at a node R.

The data structure consists of two parts: (1) a 2-*dimensional topology tree* and (2) an *extended ambivalent data structure*. Both are slight variations of data structures defined in [5, 6].

Let $e = (D, C)$ be an edge of T_1. We denote by $ST(D, C)$ the subtree of $T_1 \setminus e$ that contains D. Obviously a node C of H_1 is avoidable on a tree path P iff there exists an edge between $ST(D, C)$ and $ST(D', C)$, where D and D' are the neighbors of C on P. We call a node on P that is not an endpoint of P an *internal* node of P.

The 2-dimensional topology tree and the extended ambivalent data structure are based on H_1, not H_2. Thus, to test avoidability in H_2 we need to reduce it to testing avoidability in H_1. For each path P in T_2, let P_{T_1} denote the corresponding path in T_1 starting, resp., ending, at an arbitrary node of H_1 that maps to the start node, resp., end node, of P. Recall that each node C of H_2 is created by a set of H_1-nodes that are connected by a path of dashed edges. For an internal node C on P, let $C(P_{T_1})$ and $C(P_{T_1})'$ denote the extreme-most nodes of that dashed path on P_{T_1}. We use the following lemma.

LEMMA 2.29. *Let C be a node of H_2 on a tree path P in T_2. Let D, resp., D', be the neighbor of $C(P_{T_1})$, resp., $C(P_{T_1})'$, on P_{T_1} that does not map to C. Then C is avoidable on P iff there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$.*

Proof. Note that the edges incident to the subtree containing $h(D)$, resp., $h(D')$, in $H_2 \setminus C$ are identical to the edges incident to $ST(D, C(P_{T_1}))$, resp., $ST(D', C(P_{T_1})')$. □

As we show below, the 2-dimensional topology tree can test in time $O(m/k)$ whether there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$. We extend the ambivalent data structure of [6] such that it can test in time $O(m/k)$

(1) for all but $O(\log n)$ nodes C of H_2 on a path P of T_2 whether there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$; and

(2) for all but $O(\log n)$ nodes C of H_1 on a path P of T_1 whether there is an edge between $ST(D, C)$ and $ST(D', C)$.

Thus, to test the avoidability on a path P we use the ambivalent data structure to get the avoidability information for all but $O(\log n)$ nodes of P and we use the 2-dimensional topology tree for the remaining nodes.

Note that the term *tree edge* refers to an edge in T, T', or T_i, never to an edge in a topology tree.

The 2-dimensional topology tree. Given a restricted partition of order k we call each cluster of the partition a *level*-0 *cluster* or *basic cluster*. A *level-i cluster* is

(1) the union of two level-$(i-1)$ clusters that are connected by a tree edge such that one of them has tree degree 1 or both have tree degree 2, or

(2) one level-$(i-1)$ cluster if the previous rule does not apply.

A *topology tree* TT is a tree such that each node C at level i corresponds to a level-i cluster. If C is the union of two clusters C_1 and C_2 at level $i-1$, then C_1 and C_2 are the children of C and the tree edge (C_1, C_2) is stored at C. If C consists of one level-$(i-1)$ clusters C at level i, then C_1 is the only child of C in the topology tree. A *rooted topology tree* TT is a topology tree with the additional condition that R is only unioned when no other unions are possible.[8]

A 2-*dimensional topology tree* $2TT$ for TT is a tree that contains a node $C \times D$ at level i for every pair (C, D) of level-i clusters in TT. A level-$(i-1)$ node $C_1 \times D_1$

[8]We will exploit the rootedness in the next section.

is a child of a level-i node $C \times D$ iff C_1 is a child of C and D_1 is a child of D. We call each node in a topology tree or a 2-dimensional topology tree a *topology node*.

We keep TT and $2TT$. We store at every node $C \times D$ of $2TT$ with $C \neq D$ a bit that is set to 1 iff there is a non-tree edge between cluster C and cluster D.

Next we show how to use $2TT$ to test whether an edge exists between $ST(D, C)$ and $ST(D', C')$. Let (D, C) be a tree edge in T_1. The topology nodes *representing* $ST(D, C)$ are the nodes of TT (1) that are children of ancestors of C but are not ancestors of C, and (2) whose leaf descendants in TT belong to $ST(D, C)$. Since TT has depth $O(\log n)$ [5], $ST(D, C)$ is represented by $O(\log n)$ topology nodes.

LEMMA 2.30. *For $D \neq D'$ let (D, C) and (D', C') be edges of T_1 such that D and D' do not belong to $\pi_{T_1}(C, C')$. Let $X_1, \ldots, X_p$ be the topology nodes representing $ST(D, C)$ and let $Y_1, \ldots, Y_q$ be the topology nodes representing $ST(D', C')$ such that X_i and Y_i are level-i topology nodes.*

There is a non-tree edge between $ST(D', C')$ and $ST(D, C)$ iff a bit is set to 1 at a node $X \times Y$ of $2TT$ such that either

(1) *$X = X_i$ for some $1 \leq i \leq p$ and Y is a level-i descendant of a node Y_j for some $1 \leq j \leq q$, or*

(2) *$Y = Y_j$ for some $1 \leq j \leq q$ and X is a level-j descendant of a node X_i for some $1 \leq i \leq p$.*

Proof. Note that $ST(D, C)$ and $ST(D', C')$ are disjoint. It follows that the subtrees of TT rooted at $X_1, X_2, \ldots, X_p$ are disjoint from the subtrees rooted at $Y_1, Y_2, \ldots, Y_q$.

Let (A, B) be the non-tree edge between $ST(D, C)$ and $ST(D', C')$ with $A \in ST(D, C)$ and $B \in ST(D', C')$. Let X_i (Y_j) be the lowest ancestor of A (B) in TT that is a topology node representing $ST(D, C)$ ($ST(D', C')$). If $i \leq j$, there exists a level-i cluster Y which is a descendant of Y_j such that there is an edge between X_i and Y. It follows that the bit stored at $X_i \times Y$ is set to 1. If $j > i$, a symmetric argument applies.

If a bit is set to 1 at a node $X \times Y$ of $2TT$ such that either (1) $X = X_i$ for $1 \leq i \leq p$ and Y is a level-i descendant of a node Y_j for $1 \leq j \leq q$ or (2) $Y = Y_j$ for $1 \leq j \leq q$ and X is a level-j descendant of a node X_i for $1 \leq i \leq p$, then there is an edge between X and Y and, thus, between $ST(D, C)$ and $ST(D', C')$. ∎

Let X_i and Y_j be defined as in the lemma. Let $\mathcal{Y}_i = \{Y | Y$ is a level-i descendant of a topology node Y_j for some $i \leq j \leq q\}$ and let $\mathcal{X}_j$ be defined symmetrically. Note that the subtree in $2TT$ induced by the set of nodes $\{X_i \times Y$, for all $1 \leq i \leq p$, and all $Y \in \mathcal{Y}_i\}$ is isomorphic to a subtree of TT. The same holds with the roles of X and Y reversed. Thus, Lemma 2.29 shows how to check the avoidability of C on a path P by checking the bits of $O(m/k)$ nodes in $2TT$. Finding the topology nodes X_i and Y_i for all i takes time $O(\log n)$. As was shown in [5], $2TT$ can be maintained in time $O(k + m/k)$ after each update operation in G.

LEMMA 2.31. *A 2-dimensional topology tree can test in time $O(m/k)$ whether a node C is avoidable on a path P in a high-level graph H_i. It can be updated in time $O(k)$.*

The extended ambivalent data structure. To determine the avoidability of all but $O(\log n)$ nodes on a path P in H_i, for $i = 1, 2$, we simply extend TT and $2TT$ with additional labels to construct the *extended ambivalent data structure*. We will use two types of avoidability information, one for H_1 and one for H_2. Our approach is to partition T_i into complete paths and to keep avoidability information for each complete path. Then we show that each path P consists of subpaths of $O(\log n)$ complete paths.

Thus, P's avoidability can be determined from the avoidability information of these complete paths. To be precise let $P = \pi_{T_i}(A, B)$ in H_i. Let $P_1 = P$ if $i = 1$, and let $P_1 = P_{T_1}$ if $i = 2$. We partition P_1 at the least common ancestor LCA of its endpoints A_1 and B_1 into the paths $P_A = \pi_{T_1}(A_1, LCA)$ and $P_B = \pi_{T_1}(B_1, LCA)$. Note that both paths are *increasing*, i.e., they consist of a directed path toward the root R of T_1. We show below how to test the avoidability of all but $O(\log n)$ nodes of an increasing path by breaking it into $O(\log n)$ complete paths.

Recall that T_1 is stored in a rooted topology tree TT. Note that T_1 induces a rooted spanning tree TT_j of the nodes at each level j of TT whose root is R. Note further that when given TT, R, and also the least common ancestor between any two basic clusters can be determined in time $O(\log n)$.

We now give the necessary definitions. For a basic cluster X_1 let the graph $G(X_1)$ be (1) the graph induced by the vertices of X_1, if the tree degree of X_1 is 1 or 3, and (2) the graph induced by the vertices of X_1 with all vertices between the two boundary nodes contracted to one vertex, otherwise.

To construct complete paths we first need to introduce partial paths. We define the *partial path* of a basic cluster X_1 to be the (unique) endpoint $x(X_1)$ in $G(X_1)$ of the tree edge incident to X_1. In the following we often identify X_1 and $x(X_1)$. Note that if X_1 shares a vertex s, then the partial path of X_1 consists of s. The *partial path*[9] of a level-i cluster X_1 with $i > 0$ consists of

(1) *Case A:* the partial path of X_2, if X_1 consists of one level-$(i-1)$ cluster X_2,

(2) *Case B:* the concatenation of the partial path of X_2 and of X_3, if X_1 is the union of X_2 and X_3, and neither X_2 nor X_3 has tree degree 3.

(3) *Case C:* the vertex $x(X_3)$ and the two tree edges incident to it that are not incident to X_2 if X_1 is the union of X_2 and X_3, and X_2 has tree degree 1 and X_3 has tree degree 3. In this case the *complete path* of X_1 consists of the partial path of X_2 concatenated with the vertex $x(X_3)$. In all previous cases, the complete path of X_1 is not defined. Note that X_3 is the parent of X_2 in TT_j.

(4) *Case D:* an empty path if X_1 is the union of X_2 and X_3, and X_2 has tree degree 1 and X_3 has tree degree 1. In this case the *complete path* of X_1 consists of the partial path of X_2 and the partial path of X_3.

For every complete path not stored at the root of TT note that one endpoint has tree degree 1, and one has tree degree 3 (namely, the vertex of X_3). The endpoints of the complete path stored at the root of TT either both have tree degree 1 or one has tree degree 1 and one has tree degree 3. We call the endpoint with tree degree 1 the *tail* and the endpoint with tree degree 3 the *head* of the complete path. A tree-degree 3 node belongs to two complete paths; in one it is an internal node and in one it is a head. All other nodes belong to exactly one complete path.

If $x(X_1), \ldots, x(X_p)$ is the sequence of nodes on a partial or complete path P^c, then either $X_1, \ldots, X_p$ is an increasing path in T_1 or $X_1, \ldots, X_j$ and $X_p, X_{p-1}, \ldots, X_j$ are increasing paths, for some $1 < j < p$.

Next we show that P_A and also P_B consist of $O(\log n)$ increasing subpaths of complete paths. We will store avoidability information for the complete paths and use it to test the avoidability of P_A and P_B except for the nodes that are heads in the complete paths. Let $P_1^c, P_2^c, \ldots, P_l^c$ be the complete paths whose intersection with P_A is nonempty such that the head of P_j^c belongs to P_{j+1}^c. Let X_j be the topology node in TT at which P_j^c is stored. Note that all topology nodes whose partial path contains a vertex of P_j^c are true descendants of X_j in TT. Note further that the head

[9]A path is formed by a list of vertices.

of P^c_{j-1} belongs to P^c_j. Thus, X_{j-1} is a true descendant of X_j. Since TT has depth $O(\log n)$ it follows that P_A is contained in the union of $O(\log n)$ complete paths, i.e., $l = O(\log n)$. The same holds for P_B.

We use the algorithm described in the previous section to test the avoidability of LCA on P_A and P_B and for the heads of the complete paths. For all remaining nodes on P_A and P_B we use the extended ambivalent data structure. It consists of further labels for the 2-dimensional topology tree $2TT$ and search trees for the partial and complete paths. The labels and search trees will be oblivious of the rooting of T_1, which is important for the efficiency of rebuilds.

Every node $A \times B$ with $A \neq B$ is labeled with two additional labels *maxcov* and *shared* that are explained later. Each node $A \times A$ of $2TT$ is labeled with a pointer to the partial path and complete path (if it exists) of A. The partial and complete paths are stored in shared search trees as follows:

(1) The partial path of a level-0 cluster X is represented by one node $x(X)$.

(2) In Case A, the search tree of the partial path of X_1 is identical to the search tree of the partial path of X_2.

(3) In Case B, the partial path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

(4) In Case C, the partial path of X_1 consists of one node. The complete path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

(5) In Case D, the partial path of X_1 is empty. The complete path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

Since the topology tree has depth $O(\log n)$, every search tree has depth $O(\log n)$. A vertex v in the balanced search tree of a partial or complete path is labeled with two bits $somecov_i(v)$ for $i = 1, 2$.

Let C be an internal node on the increasing path Q in H_i whose avoidability we have to test. Let D and D' be the neighbors of C on Q such that D is the child and D' is the parent of C in T_i. Lemma 2.37 below shows that

(1) for $i = 1$, if a complete path P^c exists to which $x(D')$, $x(C)$, and $x(D)$ belong, then $somecov_1(v)$ is set to 1 for an ancestor v of $x(C)$ in P^c iff C is avoidable on Q, and

(2) for $i = 2$, let $C_1, \ldots, C_l$ form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$. If a complete path exists to which $x(D')$, $x(D)$, and $x(C_q)$ for all $1 \leq q \leq l$ belong, then for all C_q in P^c, $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ in P^c iff C is avoidable on Q.

If no such complete path exists, then some C_q is the head of a complete path and hence C is tested for avoidability using the 2-dimensional topology tree.

Note that P^c is the lowest ancestor of the least common ancestor of C for H_1 (resp., C_1 for H_2) and D in TT that has a complete path. It can be found in time $O(\log n)$.

Thus, if l nodes of an increasing path of H_1 lie on a complete path, we can test their avoidability except for the head of the complete path in time $O(l + \log n)$. Since the nodes of P_A and P_B are contained in $O(\log n)$ complete paths, we can test the avoidability of all nodes on P_A or P_B excluding LCA and the heads in time $O(m/k + \log^2 n) = O(m/k)$ for $k \leq m/\log^2 n$.

LEMMA 2.32. *Given a path P in H_i the extended ambivalent data structure can test the avoidability of all but $O(\log n)$ nodes on P in time $O(m/k)$.*

Let us now define *somecov*, *maxcov*, and *shared* and prove Lemma 2.37. For a cluster A the *projection* of a non-tree edge (u, v) with $u \in A$ and $v \notin A$ onto the partial or complete path P^c of A is the node x on P^c such that the tree path from u to x in $G(A)$ does not contain any other node on P^c.

Recall that every node $A \times B$ with $A \neq B$ is labeled with a constant number of labels: (1) For each tree edge E incident to A, there exists a label $\mathit{maxcov}(A, B, e)$ which is the node with maximum distance from e on the partial path of A that is avoidable because of a non-tree edge between A and B, assuming that the tree edge e incident to A lies on the tree path between A and B. (2) For each shared vertex s of A, there exists a label $\mathit{shared}(A, B, s)$ which is a bit that is set to 1 iff A shares s and there is an edge between A and B whose projections onto the partial path of A and of B are not nodes belonging to s.

Note that for each subpath of a complete path P^c there exist $O(\log n)$ nodes in the search tree of P^c whose leaf descendants form exactly a subpath of P^c. We say we *set the* $\mathit{somecov}_i$ *bits of a subpath* when we set the $\mathit{somecov}_i$ bits of these $O(\log n)$ nodes, excluding the nodes representing the endpoints.

The $\mathit{somecov}_i$ bits in the partial and complete paths are defined bottom-up. No basic cluster has a complete path and the partial path of every basic cluster consists of one node whose *somecov* bit is set to 0. The partial and complete path of a level-$(j+1)$ cluster X_1 is computed with the help of the *maxcov* and *shared* labels at the nodes of $2TT$ as follows. If X_1 has two children let $e = (x, y)$ be the tree edge connecting them.

(1) In Case A, the partial path of X_1 is identical to the partial path of this child.

(2) In Case B, the partial path of X_1 is built by adding a node pointing to the balanced search trees of X_2 and X_3. The $\mathit{somecov}_i$ bits of this node are unset. We set the $\mathit{somecov}_1$ bit to 1 for the path p between $\mathit{maxcov}(X_2, X_3, e)$ and $\mathit{maxcov}(X_3, X_2, e)$. We remove from p all but one representative of the shared vertices of the endpoints of p. This results in a subpath p'. If e is solid or if e is dashed, belonging to the shared vertex s, and $\mathit{shared}(X_2, X_3, s)$ is 1, then we also set the $\mathit{somecov}_2$ bits of p'. If e is dashed and $\mathit{shared}(X_2, X_3, s)$ is 0, then we split p' at the representatives of s and remove all but one representative of s from each of the resulting subpaths. We set the $\mathit{somecov}_2$ bits for these subpaths.

(3) In Case C, the partial path of X_1 consists of a tree of one node whose $\mathit{somecov}_i$ bits are set to 0. The complete path of X_1 consists of the partial path of X_2 unioned with the partial path of X_3 which consists only of the node $x(X_3)$. We describe next which $\mathit{somecov}_i$ bits of this complete path are set. We set the $\mathit{somecov}_1$ bits of the subpath p between $x(X_3)$ and $\mathit{maxcov}(X_2, Y, e)$ for any level-j cluster Y in $TT_j \setminus X_2$. We remove from p all but one representative of the shared vertices of the endpoints of p. We set the $\mathit{somecov}_2$ bits of this subpath.

(4) In Case D, the partial path of X_1 is empty. The complete path of X_1 consists of the partial path of X_2 unioned with the partial path of X_3. We set the $\mathit{somecov}_1$ bits of the subpath p between $\mathit{maxcov}(X_2, X_3, e)$ and $\mathit{maxcov}(X_3, X_2, e)$. We remove from p all but one representative of the shared vertices of the endpoints of p. This results in a subpath p'. If e is solid or if e is dashed, belonging to the shared vertex s, and $\mathit{shared}(X_2, X_3, s)$ is 1, then we also set the $\mathit{somecov}_2$ bits of p'. If e is dashed and $\mathit{shared}(X_2, X_3, s)$ is 0, then we split p' at the representatives of s and remove all but one representative of s from each of the resulting subpaths. We set the $\mathit{somecov}_2$ bits for these subpaths.

Given the *maxcov* and *shared* labels, the above description also is an algorithm to build the partial paths of X_1 from the partial paths of the children of X_1 in time $O(\log n)$ and the complete path of a level-j cluster in time linear in the number of level-j nodes in TT.

Next we show how to use the $somecov_i$ bits to test the avoidability of a node e. We start with the $somecov_1$ bits.

LEMMA 2.33. *Let P^c be a complete path and let D, D', and C be basic clusters such that $x(C), x(D)$, and $x(D')$ belong to P^c. Then $somecov_1(v)$ is set for an ancestor v of $x(C)$ in P^c iff there exists a non-tree edge between $ST(C,D)$ and $ST(C,D')$.*

Proof. Consider the lowest level node X_1 such that the partial or complete path P^x of X_1 contains $x(c), x(D)$, and $x(D')$. Let $j+1$ be the lowest level of X_1 at which a $somecov_1$ bit is set for an ancestor v of $x(C)$. Then X_1 has two children X_2 and X_3 in TT, connected by an edge e in TT_j. Without loss of generality (w.l.o.g.) $x(C)$ is a node of the partial path of X_2. Then $somecov_1(v)$ is set in the path of X_1 because $x(C)$ is an internal node of the subpath of P^x between $maxcov(X_2, X_3, e)$ and the first node of X_3 on P^x. By the definition of $maxcov$ there exists an edge between $ST(C,D)$ and $ST(C,D')$.

Assume next that an edge exists between $ST(C,D)$ and $ST(C,D')$. Let X_1 be the least common ancestor of D and D' in TT and let X_2 and X_3 be its two children. W.l.o.g. the partial path of X_2 contains $x(C)$ and $x(D)$. Since there exists a non-tree edge between X_2 and X_3 whose projection is $x(D)$, $x(C)$ lies between $maxcov(X_2, X_3, e)$ and the first node of X_3 on the partial or complete path P^x of X_1. Thus the $somecov_1$ bit is set for an ancestor of $x(C)$ in the partial or complete path of X_1 and, hence, also in P^c. ∎

Next we discuss under which conditions $somecov_2(v)$ is set for an ancestor v of $x(C)$.

LEMMA 2.34. *Let C be a basic cluster and let P^c be a complete path containing C. If C is not connected in T_1 to its neighbors in P^c by a dashed edge of P^c, then $somecov_2(v)$ is set for an ancestor v of $x(C)$ in P^c iff $somecov_1(v)$ is set.*

Proof. The lemma follows immediately from the definition of $somecov_2$. ∎

LEMMA 2.35. *Let P^c be a complete path and let $C_1, \ldots, C_l$ be basic clusters such that $x(C_1), \ldots, x(C_l)$ forms a maximal subpath of P^c sharing the same vertex s. Let j be the lowest level such that the $somecov_2$ bits are set for an ancestor for every node $x(C_1), \ldots, x(C_l)$. Then all nodes $x(C_1), \ldots, x(C_l)$ belong to the partial or complete path of the same cluster at level j.*

Proof. Let P^c be stored at a level j^* node. The claim obviously holds for level j^*. Assume it does not hold for a level $j < j^*$. Then there exists at least one node $x(C_q)$ that is an endpoint of its partial path on level j and in this partial path there exists an ancestor of $x(C_q)$ whose $somecov_2$ bit is set. Note that $x(C_q)$ was the endpoint of this partial path for every level $\leq j$. Note further that the $somecov_2$ bit is never set to 1 for an ancestor of an endpoint of a partial path. Thus at level j, the $somecov_2$ bit is not set for any ancestor of $x(C_q)$. This is a contradiction. ∎

LEMMA 2.36. *Let P^c be a complete path and let D, D', and $C_1, C_2, \ldots, C_l$ be basic clusters such that $x(D), x(C_1), x(C_2), \ldots, x(C_l), x(D')$ is a subpath of P^c and $x(C_1), x(C_2), \ldots, x(C_l)$ forms a maximal subpath of P^c sharing the same vertex. Then, for all $1 \leq q \leq l$, $somecov_2(v)$ is set for an ancestor v of $x(C_q)$ in P^c iff there exists a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$.*

Proof. Let s be the vertex shared by the cluster C_q to which the dashed edges incident to C_q belong. Consider the lowest level $j+1$ at which, for all $1 \leq q \leq l$, $somecov_2(v_q)$ is set for an ancestor v_q of $x(C_q)$. By Lemma 2.35, all $x(C_q)$ belong to the partial or complete path P^x of the same level-$(j+1)$ cluster X_1. Then X_1 has two children X_2 and X_3 connected by an edge e in TT_j. Note that one of the $somecov_2(v_q)$ bits was set while constructing P^x.

If neither X_2 nor X_3 has tree degree 3, a $somecov_2(x(C_q))$ bit was set when constructing the path for X_1 because either (1) $x(D), x(D')$, and all nodes $x(C_q)$ are nodes on the path between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$, and e does not belong to s, or (2) there exists an edge between X_2 and X_3 whose projections do not belong to s. In either case there exists a non-tree edge between $ST(D, C_1)$ and $ST(C_l, D')$.

If X_2 has tree degree 1 and X_3 has tree degree 3, a $somecov_2(x(C_q))$ bit is set while constructing the path for X_1 only if all nodes $x(C_q)$ are internal nodes on the path between $maxcov(X_2, Y, e)$ and $x(X_3)$ for a level-j cluster Y in $TT_j \setminus X_2$. It follows that there is a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$.

Assume next that an edge exists between $ST(C_1, D)$ and $ST(C_l, D')$. Let X_1 be the least common ancestor of D and D' and let X_2 and X_3 be its two children. If neither X_2 nor X_3 has tree degree 3, we consider two cases. If the tree edge e between X_2 and X_3 does not belong to s, the $somecov_2$ bits of the subpath $x(C_1), \ldots, x(C_l)$ including endpoints are set because the tree edge lies between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$. If e belongs to s, the $somecov_2$ bits of the subpath are set because $shared(X_2, X_3, s)$ is 1.

If X_2 has tree degree 1 and X_3 has tree degree 3, then $x(X_3) = x(D')$ or $x(X_3) = x(D)$, i.e., e is solid. The subpath $x(C_1), \ldots, x(C_l)$ is internal to the path between $maxcov(X_2, Y, e)$ and $x(X_3)$ for some level-j cluster Y in $TT_j \setminus X_2$. Thus, the $somecov_2$ bits of the subpath $x(C_1), \ldots, x(C_l)$ including endpoints are set. ∎

LEMMA 2.37. *Let C be an internal node on the increasing path Q in T_i and let D and D' be the neighbors of C on Q such that D is the child and D' is the parent of C.*

(1) *For $i = 1$, if a complete path P^c exists to which $x(D')$, $x(C)$, and $x(D)$ belong, then $somecov_1(v)$ is set to 1 for an ancestor v of $x(C)$ in P^c iff C is avoidable on Q.*

(2) *For $i = 2$, let $C_1, \ldots, C_l$ form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$. If a complete path exists that contains $x(D')$, $x(D)$, and $x(C_q)$ for all $1 \le q \le l$, then for all C_q in P^c $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ iff C is avoidable on Q.*

Proof. Let P^c be C's complete path. We discuss first the case $i = 1$. Lemma 2.33 shows that $somecov_1$ is set for an ancestor of $x(C)$ iff there is a non-tree edge between $ST(C, D)$ and $ST(C, D')$. By the definition of avoidability, the latter holds iff C is avoidable on Q.

Next we discuss the case $i = 2$. Lemma 2.34 shows the claim if $l = 1$. If $l > 1$, then C represents at least two clusters in H_1. Lemma 2.36 shows that for all C_q in P^c $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ iff there exists a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$. The latter holds iff C is avoidable on Q. ∎

LEMMA 2.38. *The extended ambivalent data structure can determine the avoidability of all but $O(\log n)$ nodes on a path P in H in time $O(m/k)$.*

Updates. Next we show how to maintain the *maxcov* and *shared* values and the partial and complete paths in time $O(k)$ after each update operation in G. First, we discuss the *maxcov* and *shared* values. By definition an update (u, v) operation affects only the $maxcov(X, Y, e)$ and $shared(X, Y, s)$ values iff either X or Y contains either u, v, x, or y, where (x, y) is the new tree edge. Consider the subtree of $2TT$ induced by marking all nodes $A \times B$ such that A and B contain either u, v, x, or y.

Since this subtree forms a structure which is isomorphic to two copies of TT, the total number of affected *maxcov* and *shared* values is $O(m/k)$. As was shown in [6] for the *maxcov* values and as we show below for the *shared* values each such value at an internal node of $2TT$ can be computed in constant time from the values of its children and in time $O(k)$ for a basic cluster. Thus, updating all *maxcov* and all *shared* values takes time $O(k)$.

Lemma 2.39 shows that the only clusters whose partial or complete paths are affected by updates are the ones that are ancestors in TT of the basic cluster containing u, v, x, or y. Thus, the partial and complete paths of at most two clusters at each level have to be updated. We discuss below how to restore the partial and complete paths of the clusters that do not contain u, v, x, or y, but are children of clusters containing u, v, x, or y. The partial path of a cluster C can be computed in time $O(\log n)$ from the partial paths of the children of C and the *shared* and *maxcover* values. The complete path of a cluster C at level j can be computed in time linear in the number of level j nodes in TT from the partial path of the child of C with tree degree 1 and from the *shared* and *maxcover* values. Since TT has depth $O(\log n)$ and size $O(m/k)$, all affected partial and complete paths can be updated in time $O(m/k + \log^2 n) = O(m/k)$ for $k \leq m/\log^2 n$.

Whenever we build the partial or complete path we keep a *back log* that stores for each node X_1 of $2TT$ all the operations that were executed to build the partial or complete path of X_1 from the partial or complete path of its children. Whenever we execute an update operation, we walk top-down in TT and restore the path of the suitable clusters and their children. The partial and complete path of the root of TT are given. Assume inductively the partial and complete path of a node X at level j are restored. Undo the operations in the back log of X to restore the partial and complete paths of the children of X. Then recurse on the suitable child(ren) of X. The same argument as above shows that this takes time $O(m/k)$. Note also that modifications in the back log of one child does not affect the back log of its sibling.

LEMMA 2.39. *An insert(u, v) and a delete(u, v) operation only modifies the balanced tree of partial or complete paths of clusters containing u, v, x, or y, where (x, y) is a new tree edge.*

Proof. A *somecov* bit is set at a node in the balanced tree representing the partial path of a cluster C only if there exists an edge internal to C that covers the corresponding nodes. For a cluster not containing u, v, x, or y neither the partial path nor the non-tree edges internal to the cluster have changed. Thus, the balanced search tree of its partial path does not have to be updated.

Next we discuss complete paths. If a cluster C which has a complete path does not contain u, v, x, or y, then the partial path of its child C' with tree degree 1 and the non-tree edges incident to C' are not affected by the above argument. The modifications to this partial path that create the data structure for the complete path of C depend only on the projection of edges incident to C' onto the partial path of C'. Since the partial path of C' and the non-tree edges incident to C' did not change, the balanced search tree of the complete path of C is not affected by the operation. ∎

We are left with showing how to compute $shared(A_1, B_1, s)$ from the *shared* values of the children of A_1 and B_1 and information stored at their children in constant time. Recall that for each pair of clusters at the same level $shared(A_1, B_1, s)$ is 1 iff A_1 shares s and there is an edge between A_1 and B_1 whose projection onto the partial path of A_1 and onto the partial path of B_1 is not s. If A_1 does not share a vertex s, then $\mathit{shared}(A_1, B_1, s)$ is not defined. Since each basic cluster and each cluster with tree degree 1 or 3 shares at most one vertex and each nonbasic cluster with tree degree 2 shares at most two vertices, at most four $\mathit{shared}(A_1, B_1, .)$ values are defined for every pair of clusters A_1 and B_1. We distinguish cases depending on the number of children of A_1 and of B_1 under the assumption that A_1 shares the vertex s.

Case 1: A_1 and B_1 are basic clusters.

Then $\mathit{shared}(A_1, B_1, s) = 0$, since every edge incident to A_1 is projected onto s when it is projected onto the partial path of A_1.

Case 2: A_1 and B_1 are clusters at level $j > 0$.

Case 2.1: A_1 has one child A_2 and B_1 has one child B_2.

Then $\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_2, B_2, s)$.

Case 2.2: A_1 has one child A_2 and B_1 has two children B_2 and B_3.

If neither B_2 nor B_3 has tree degree 3, then $\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_2, B_2, s)$ *or* $\mathit{shared}(A_2, B_3, s)$.

If the tree degree of B_3 is 3 and the tree degree of B_2 is 1, then $\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_2, B_2, s)$ if B_3 does not share s and 0 if B_3 shares s.

Case 2.3: A_1 has two children A_2 and A_3, A_3 shares s, and B_1 has one child B_2.

W.l.o.g. A_3 is incident to the tree edge incident to A_1. Thus, A_3 has tree degree at least 2. If the tree degree of A_3 is 3, then $\mathit{shared}(A_1, B_1, s) = 0$, since every edge incident to A_1 is projected onto s when it is projected onto the partial path of A_1.

If the tree degree of A_3 is 2, then we distinguish between the cases that A_2 shares s and that A_2 does not share s. If A_2 shares s, then

$$\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_2, B_2, s),$$

since the projection of every edge incident to A_3 onto the partial path of A_1 is s.

If A_2 does not share s, then we distinguish between the cases that B_2 shares s and that B_2 does not share s. If B_2 shares s, then

$$\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_3, B_2, s) \mathit{or}\ \mathit{shared}(B_2, A_2, s).$$

(Note that $\mathit{shared}(A_2, B_2, s)$ is not defined in this case, but $\mathit{shared}(B_2, A_2, s)$ is defined.)

If B_2 does not share s, then

$$\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_3, B_2, s) \mathit{or}\ \mathit{edge}(A_2, B_2),$$

where $\mathit{edge}(A_2, B_2) = 1$ iff there exists an edge between A_2 and B_2 iff $\mathit{maxcov}(A_2, B_2, e)$ (for any tree edge e incident to A_2) is defined.

Case 2.4: A_1 has two children A_2 and A_3 and B_1 has two children B_2 and B_3.

W.l.o.g. A_3 is incident to the tree edge incident to A_1. Thus, A_3 has tree degree at least 2. If the tree degree of A_3 is 3, then $\mathit{shared}(A_1, B_1, s) = 0$, since the projection of every non-tree edge incident to A_1 onto the partial path of A_1 is s.

If the tree degree of A_3 is 2, then we distinguish between the cases that A_2 shares s and that A_2 does not share s. If A_2 shares s, then

$$\mathit{shared}(A_1, B_1, s) = \mathit{shared}(A_2, B_2, s) \mathit{or}\ \mathit{shared}(A_2, B_3, s),$$

since the projection of every non-tree edge incident to A_3 onto the partial path of A_1 is s.

If A_2 does not share s, then we distinguish between the case that (1) B_2 shares s and B_3 does not share s, that (2) B_3 shares s and B_2 does not share s, that (3) both share s, and that (4) both do not share s.

In case (1) (B_2 shares s and B_3 does not share s)

$$\begin{aligned}shared(A_1,B_1,s) = {} & shared(A_3,B_2,s)\, or\; shared(A_3,B_3,s)\, or \\ & shared(B_2,A_2,s)\, or\; edge(A_2,B_3).\end{aligned}$$

(Note that $shared(A_2,B_2,s)$ is not defined in this case, but $shared(B_2,A_2,s)$ is defined.) The case (2) is symmetric to case (1).

In case (3) (B_2 and B_3 share s)

$$\begin{aligned}shared(A_1,B_1,s) = {} & shared(A_3,B_2,s)\, or\; shared(A_3,B_3,s)\, or \\ & shared(B_2,A_2,s)\, or\; shared(B_3,A_2,s).\end{aligned}$$

In case (4) (B_2 and B_3 do not share s)

$$\begin{aligned}&rclshared(A_1,B_1,s) \\ &\quad = shared(A_3,B_2,s)\, or\; shared(A_3,B_3,s)\, or\; edge(A_2,B_2)\, or\; edge(A_2,B_3).\end{aligned}$$

This shows that the $shared(A_1,B_1,.)$ bit can be computed in constant time from the *shared* and *maxcov* values of the children of A_1 and B_1 and it finishes the proof of the following lemma.

LEMMA 2.40. *The extended ambivalent data structure can be updated in time $O(k)$.*

2.8. The c-structure. In this section we address the following problem. Given the c-nodes of a cluster graph (see section 2.5) or the c-nodes of a shared graph $G(s)$ for a new shared vertex s (see section 2.6.1), determine which c-nodes are CV-split by an update operation. For this problem we give in this subsection a data structure, called *c-structure* and show in the next subsection that in a phase $O(m/k)$ splits of c-node occur because of violation of condition (a) or (b) (called *CV-splits*). Additionally we show that the e-nodes of a shared graph $\tilde{G}(s)$ for an old shared vertex s (see section 2.6.2) fulfill the conditions of a c-node and therefore the same data structure and proof apply.

First we recall the definitions of c-node and e-node; next we show that an e-node also fulfills the conditions of a c-node, and then we define the c-structure exactly .

Let $H = H_1$ for c-nodes in cluster graphs and $H = H_2$ for c-nodes in shared graphs. Given a node C in H with ancestor A a c-node represents a maximal set X of nodes of H such that

(a) every node $C' \in X$ is a neighbor of C,
(b) all nodes in X are connected in $H \setminus C$,
(c) all nodes in X have the same ancestor which is different from A,
(d) at the creation of C, the set X contains only one node, and
(e) at each previous point in time since the creation of C all nodes of H that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node.

We show next that every e-node in the graph $\tilde{G}(s)$ of an old shared vertex s fulfills the conditions of a c-node with $H = H_2$. Thus the amortization lemma of the next section also applies to e-nodes.

LEMMA 2.41. *Every e-node in the graph $\tilde{G}(s)$ of an old shared vertex s fulfills the conditions of a c-node with $H = H_2$.*

Proof. By definition every e-node fulfills conditions (a)–(d) of a c-node. We only have to show that it also fulfills condition (e).

Let X be an e-node. By definition the vertices in $\cup_{C' \in X} C'$ used to belong to the same node C_{old} of H_2 and since the split of C_{old}, the graph $G(s)$ was not modified and the connected components of $H_2 \setminus C_s$ did not change. Thus, before the last change in $G(s)$ or $H_2 \setminus C_s$ the nodes in X belonged to the same node of H_2 and thus were represented by the same e-node.

Assume by contradiction that the nodes in X were represented by different e-nodes at some point since the last change in $G(s)$ or $H_2 \setminus C_s$. While the nodes were represented by different e-nodes they either must violate condition (a) or (b) of an e-node, since conditions (c), (d), and (e) continue to hold. However, now they fulfill conditions (a) and (b), i.e., either $G(s)$ or $H_2 \setminus C_s$ must have changed, which is a contradiction. □

Thus, e-nodes are just a special case of c-nodes and we will just use the term c-node in the following to denote c-nodes as well as e-nodes.

A c-node of C is *CV-split* iff conditions (a) or (b) of a c-node are no longer fulfilled.

Given the high-level graph H and its data structures the c-structure maintains the c-nodes of each node in H under the following operations:

(1) *c-split* (C, C_1, C_2, u, v), where C is a node of H split by the *delete*(u, v) or *insert*(u, v) operation, C_1 and C_2 are the two nodes of H created by the split. Split the node C into C_1 and C_2.

(2) *c-add* (C_1, C_2), where C_1 and C_2 are nodes of H. Add one edge between C_1 and C_2.

(3) *c-remove* (C_1, C_2), where C_1 and C_2 are nodes of H. Remove the edge between C_1 and C_2 and return a (possibly empty) list of CV-split c-nodes and for each newly created c-node return its element list.

We use the following data structure for the c-structure, which uses $O((m/k)^2)$ space.

(T1) For each node of H we keep a list of its c-nodes. For each c-node we keep a list of its elements. For each node in H we keep a list of all the c-nodes it belongs to. The position of the node in the list of the c-node and the position of the c-node in the list of the node point to each other.

We keep two c-structures, namely, one with $H = H_1$ to determine the CV-splits in c-nodes of the cluster graphs, and one with $H = H_2$ to determine the CV-splits in c-nodes of the shared graph for new shared vertices.

2.8.1. Implementing the c-structure.

We implement the operations as follows.

c-split (C, C_1, C_2, u, v): This requires (i) updating the c-nodes of C and (ii) updating the c-nodes containing C. (i) Discard all c-nodes of C. Each neighbor of C_1 (resp., C_2) forms a 1-element c-node for C_1 (resp., C_2). (ii) Use (T1) to determine all c-nodes X to which C belongs. Replace C by either C_1 or C_2 or both in X, depending on which of the new nodes are incident to D. Note that all nodes in X still fulfill (a), (c), (d), and (e) of a c-node. By Lemma 2.8 all nodes continue to fulfill (b) as well.

c-add (C_1, C_2): If C_1 and C_2 have the same ancestor, do nothing. Otherwise, search the c-nodes to which C_1 belongs to determine whether C_2 is one of them. If not, add to the c-nodes of C_2 a 1-element c-node consisting of C_1. Repeat with the roles of C_1 and C_2 exchanged.

c-remove (C_1, C_2): If C_1 and C_2 have the same ancestor, do nothing. Otherwise, determine the c-node of C_2 to which C_1 belongs and remove C_1 from it. If this c-node becomes empty, discard it. If the c-node is modified, output it and its new element list. Repeat with the roles of C_1 and C_2 exchanged. Finally determine all articulation points D on $\pi(C_1, C_2)$ in the (updated) graph H using (HL4). Test as follows for each c-node X of D whether (b) is violated, and if so, how to partition X. Using (HL3) determine for each node C' in X the tree neighbor of D on $\pi(C', D)$ and bucketsort the node according to the blockid of "its" tree neighbor using (HL4) and (HL6). This results in either one or two nonempty buckets. In the former case (b) is not violated. In the latter case, split the list of X according to the two buckets and report the CV-split of X and return the two resulting lists.

To analyze the running time note that the intersection of two different c-nodes of C is empty. The time spent by a *c-split* or *c-add* operation is linear in the number of c-nodes of a node in H, which is $O(m/k)$. In *c-remove* we spend the time $O(m/k)$ to determine all articulation points and then the following time per articulation point D: $O(\log n)$ per non-tree neighbor of D to bucketsort it and constant time to remove it from the bucket again. Since the nodes D are articulation points on a path in H, this sums up to $O((m/k)\log n)$ by Lemma 2.1. Additionally the *c-remove* operation spends time $O(m/k)$ to update the c-nodes of C_1 and the c-nodes of C_2. Thus, the total time spent is $O((m/k)\log n)$.

2.8.2. Updating the c-structure. At the beginning of a phase each c-node consists of one node: every neighbor of a node in H forms its own c-node.

Whenever an edge is inserted in G and H changes, then first the data structures of H are updated and then a *c-add* and potentially afterward a constant number of *c-split*'s are executed in the c-structure. Whenever an edge is deleted from G and H changes, then first the data structures of H are updated and then potentially a constant number of *c-split*'s and afterward a *c-remove* are executed in the c-structure. If an internal tree edge of a cluster C is deleted, then this implies that first the cluster is split at this tree edge and afterward the tree edge is deleted. Each operation can be implemented in time $O((m/k)\log n)$. Since there are only a constant number of them per update in G, this gives a total time of $O((m/k)\log n)$ to update the c-structure.

THEOREM 2.42. *The c-structure*

(1) *can be updated in time $O((m/k)\log n)$ after each update in G, and returns all the c-nodes CV-split by the update and the resulting c-nodes, and*

(2) *can be built in time $O(m)$.*

2.9. The amortization lemma. We show next that during a sequence of l updates in a phase $O(l)$ CV-splits of c-nodes occur. A similar, but less general lemma was shown in [11].

LEMMA 2.43. *During l updates in G in a phase at most $2l$ CV-splits of a c-node occur because of violation of condition* (a).

Proof. Condition (a) is violated if a node C' belongs to a c-node of node C, but C' is no longer incident to C. This is only possible if C as well as C' contains an endpoint of the update edge. Since all c-nodes of C are disjoint, condition (a) is violated for at most one c-node of C. Similarly, condition (a) is violated for at most one c-node of C' and for no c-nodes at other nodes. ☐

LEMMA 2.44. *During l updates in G in a phase $O(l + m/k)$ CV-splits occur because of the violation of condition* (b) *for $l \geq m/k$.*

Proof. We construct a bipartite graph K consisting initially of $O(m/k)$ blue nodes, $O(m/k)$ red nodes, and $O(m/k)$ edges between red and blue nodes. A red node is incident to at least one blue node. We show that during a sequence of l updates in G the number of blue nodes in K increases by $O(l)$ (Proposition 2.45), each CV-split of a c-node increases the number of connected components of K by at least one (Proposition 2.46), and no other operation decreases the number of components (Proposition 2.51). Thus, there are at most $O(l + m/k)$ splits of c-nodes during l updates in G.

The proof will exploit the following fact: Whenever a c-node at C containing D_1 and D_2 is CV-split, then there is no c-node at another node C' that contains both D_1 and D_2. (Otherwise the path "through" C' would connect D_1 and D_2 in $H \setminus C$.) Thus, the split discards the last common c-node of D_1 and D_2. We will show that an even stronger property holds: Assume the relation $r(D_1, D_2)$ holds iff D_1 and D_2 have a common c-node. Let r^* be the transitive closure of r. Then whenever the c-node containing D_1 and D_2 is split, then $r^*(D_1, D_2)$ does not hold after the split. The graph K is constructed so that this fact implies that the connected components of K increase.

Recall that an edge of H consists of a set of edges of G'. An edge of H is called *new* if all edges of G' in its set are new, i.e., have been inserted after the last rebuild. All other edges of H are *old*. We "treat" new edges in a special way to guarantee that edge insertions do not decrease the number of connected components of K. Note that there are at most l new edges in H at each point in time.

We next define K.

(1) For each node C in H and each c-node X at C, K contains a red node (C, X).

(2) For each node D in H, K contains a blue node D.

(3) For each node D in a c-node X at C such that (D, C) is new there exists a blue node (D, X). These nodes are called *special*.

(4) Let D be in the c-node X at C. If both a red node (C, X) and a blue node (D, X) exist, there exists an edge between (C, X) and (D, X). If (D, X) does not exist, there is an edge between (C, X) and D.

By abuse of notation we will equate a blue node in K with the node of H represented by the blue node. Note that every edge in K corresponds to an edge of H. Thus, if two blue nodes are connected in K, their nodes are connected in H.

There are four events that modify K: (A) a *c-split* operation, (B) a *c-add* operation, (C) a *c-remove* operation, and (D) the change of an edge from old to new.

Next we describe each event in detail:

(A) A *c-split* (C, C_1, C_2, u, v) operation. (1) Every red node (C, X) is removed. For each $D \in X$ incident to C_1 we create a red node $(C_1, \{D\})$, and if the blue node (D, X) exists, it is replaced by a blue node $(D, \{D\})$. The new red node is connected to $(D, \{D\})$ if it exists and to D otherwise. We proceed in the same way with C_2. (2) The blue node C and all blue nodes (C, X) are split into two nodes and connected to the appropriate neighbors of the split nodes.

(B) A *c-add* (C_1, C_2) operation. It might add a new red node at $(C_1, \{C_2\})$, a new blue node $(C_2, \{C_2\})$, and connect them by an edge. It might do the same with the roles of C_1 and C_2 reversed.

(C) A *c-remove* (C_1, C_2) operation. Let X be the c-node at C_1 containing C_2. Remove the edge between the red node (C_1, X) and the corresponding blue node representing C_2. If the blue node (C_2, X) exists, remove it. If $X = \{C_2\}$, also remove (C_1, X). Proceed in the same way with the roles of C_1 and C_2 reversed. Finally for each split c-node X' of an articulation point D on

$\pi(C_1, C_2)$ in H replace the red node (D, X') by two red nodes, one for each new c-node and connect their blue neighbors suitably.

(D) An old edge (D, C) of H becomes new. If D belongs to the c-node X at C, then add a new blue node (D, X) with edge to (C, X) and remove the edge from D to (C, X). Then proceed in the same way with the roles of D and C reversed.

We prove next the three missing propositions.

PROPOSITION 2.45. *During l update operations the number of blue nodes increases by $O(l)$.*

Proof. A sequence of l update operations in G leads to at most $7l$ *c-split* operations, l *c-add* operations, and l *c-remove* operations. At each point there are at most $2l$ special blue nodes. Next we bound the number of nonspecial blue nodes. A *c-add* operation, a *c-remove* operation, and the change of an edge from old to new do not increase the number nonspecial blue nodes. A *c-split* increases the number of nonspecial blue nodes by at most 1. Thus, the number of nonspecial blue nodes increases by $O(l)$. □

Next we show that the CV-split of a c-node increases the number of connected components by at least 1.

PROPOSITION 2.46. *A CV-split of a c-node increases the number of connected components by at least* 1.

Proof. Consider the split of the c-node X at node C'. A c-node is CV-split only during a *c-remove* operation. So consider the removal of edge (C_1, C_2). Let $\{X_1, X_2, \ldots, X_p\}$ be all the c-nodes that are CV-split at C'. Let D_1 and D_2 be the tree neighbors of C' on $\pi(C_1, C_2)$ and let Y_1 and Y_2 be the blue nodes representing D_1 and D_2 and incident to the red node (C', X_i) for some $1 \le i \le p$ in K. To update K, each red node (C', X_j) is replaced by two new red nodes, one representing each new c-node. Each (blue) neighbor of (C', X_j) is connected to exactly one of the new c-nodes depending on which new c-node it belongs to. Obviously, D_1 and D_2 are connected to different new red nodes. As we show in Proposition 2.50 after the *c-remove* operation there exists no path in K anymore between D_1 and D_2, i.e., the number of connected components in K has increased by at least 1. □

We are left with proving Proposition 2.50 and showing that the number of connected components of K does never decrease. We first need some intermediate results.

PROPOSITION 2.47. *Every blue node (D, X) has degree* 1 *in K.*

Proof. Let X be a c-node of node C of H. By definition of K, a blue node (D, X) can only be adjacent to node (C, X). □

PROPOSITION 2.48. *Let C be a node in H. Each blue node representing a node D in H is incident to at most one red node (C, X), and X is the c-node to which D belongs at C.*

Proof. The proof follows from the construction of K since each node D belongs to at most one c-node at C. □

PROPOSITION 2.49. *If two blue nodes are connected in K, then they represent nodes with the same ancestor.*

Proof. If there is a path in K between the two blue nodes Y and Y', then let $B_1, \ldots, B_j$ be the blue nodes on this path with $Y = B_1$ and $Y' = B_j$. Since B_i and B_{i+1} are adjacent to the same red node, they belong to the same c-node at that node and thus have the same ancestor. By the transitivity of the ancestor relation the claim follows. □

PROPOSITION 2.50. *Consider the operation c-remove(C_1, C_2). Let $\{X_1, X_2, \ldots,$*

$X_p\}$ *be all the c-nodes that are CV-split at a node* C' *of* H. *Let* D_1 *and* D_2 *be the tree neighbors of* C' *on* $\pi(C_1, C_2)$. *Let* Y_1 *and* Y_2 *be the blue nodes representing* D_1 *and* D_2 *that are incident to a red node* (C', X_i) *for some* $1 \leq i \leq p$. *Then after the c-remove operation no path exists connecting* Y_1 *and* Y_2.

Proof. Consider first the case that either (D_1, C') or (D_2, C') is new. By Proposition 2.47 every path between Y_1 and Y_2 contains (C', X_i) and hence is disconnected after the remove operation.

Assume next that both edges are old, i.e., $Y_1 = D_1$ and $Y_2 = D_2$, and assume that a path P exists between them after the *c-remove* operation. Since the c-node of D_1 and D_2 was CV-split, after the deletion of (C_1, C_2) every path in H connecting D_1 with D_2 in H contains C'. We will show that the existence of P implies the existence of a path in $H \setminus C'$ connecting D_1 and D_2, which gives the contradiction.

For this we show (1) that no blue node representing C' belongs to P, and (2) that the blue nodes incident to a red node (C', X') on P are connected in $H \setminus C'$ after the update.

(1) Since D_1 and D_2 belonged to a c-node at C', their ancestor differs from the ancestor of C'. By Proposition 2.49 the nodes of H represented by the blue nodes of P all have the same ancestor. Thus, no blue node representing C' belongs to P.

(2) Let F_k and F'_k be the two nodes incident on P to the kth red node (C', X') for some c-node X'. Then F_k and F'_k both belong to the same c-node X'. It follows that F_k and F'_k are connected in $H \setminus C'$ after the deletion of edge (C_1, C_2).

From (1) it follows that P forms a path without a blue node representing C': (2) shows that every red node on P representing C' can be avoided by a path in $H \setminus C'$. Let l be the number of red nodes representing C' on P. Note that the subpaths of P between F'_k and F_{k+1}, the subpath from D_1 to F_1, and the subpath from F'_l to D_2 contain no edge incident to C' and thus correspond to paths in $H \setminus C'$. It follows that P induces a path in $H \setminus C'$ between D_1 and D_2 after the deletion of (D_1, D_2), which is a contradiction. ▯

Proposition 2.51. *The number of connected components of* K *never decreases.*

Proof. As shown in Proposition 2.46 a *c-remove* operation does not decrease the number of connected components.

A *c-split* operation consists of two parts. In part 1 the red node (C, X) is replaced by many red nodes, each being connected at most to all the nodes that (C, X) was connected to. This does not decrease the number of connected components. In part 2 the blue node C and all blue nodes (C, X) are each split into two new nodes such that each new node is connected to at most all the nodes that the original blue node was connected to. So again, the number of connected components is not decreased.

A *c-add* (C_1, C_2) operation might add a new blue node (C_2, X), a new red node (C_1, X), an edge between them, and the same with the roles of C_1 and C_2 reversed. Since they do not connect to the rest of K, an *add* operation does not decrease the number of connected components in K either.

Note that an old edge (D, C) of H can become new, but not vice versa. If this happens an edge is removed from K, a new blue node (D, X) is added and is connected to (C, X), where X is the c-node of C to which D belongs. The same happens with the roles of D and C reversed. Thus the number of connected components does not decrease. ▯

This completes the proof of the lemma. ▯

2.10. Complete block queries. A complete block query determines all the blocks to which a vertex belongs by computing for each tree edge the block to which it belongs. A vertex belongs to exactly the blocks to which the tree edges adjacent to the vertex belong. We can find the blocks in $I(C)$ for every tree edge internal or incident to the cluster C in time $O(k)$ whenever we recompute $I(C)$. To compute all the blocks in G, we have to determine which blocks of different cluster graphs form the same block of G, i.e., have to be combined.

Perform a depth-first traversal of the spanning tree T_2 of H_2. For each tree edge $e = (u, v)$ with $u \in C_1$ and $v \in C_2$ such that u is a parent of v in the (rooted) depth-first search (dfs) tree, test for each tree edge (x, w) with $x \in C_2$ and $w \in C_3$ whether u and w are biconnected: if C_2 is also a node of H_1, then test whether the shared vertex of C_2 separates u and w and if not use the internal data structure of C_2 to test the biconnectivity of u and w in G. If C_2 is no node of H_1, then $v = x$ is a shared vertex. In this case test the biconnectivity of u and w using the shared graph of v. If we recursively know all tree edges of T_2 in the dfs subtree of edge (x, w) that belong to the same block as (x, w), then we can construct for (u, v) the set of all tree edges of T_2 in the dfs subtree of (u, v) that belong to the same block as (u, v). The dfs takes time $O(m/k)$.

When the dfs is completed we combine the blocks of all the tree edges in the same set and mark all the edges in T' accordingly. Thus the total cost is proportional to the number of tree edges in T', which is $n - 1$.

THEOREM 2.52. *A complete block query in a graph of n vertices can be answered in time $O(n)$.*

2.11. Biconnectivity queries. Given a *query*(u, v) operation, let $x^{(i)}$ and $y^{(i)}$ be defined as in Lemma 2.7. The lemma shows that u and v are biconnected in G iff

(Q1) u and $y^{(1)}$ are biconnected in G,
(Q2) $x^{(i)}$ and $y^{(i+1)}$ are biconnected in G, for all $1 \leq i < p$, and
(Q3) $x^{(p)}$ and v are biconnected in G.

Condition (Q2) holds iff $e_i = (x^{(i)}, y^{(i)})$ and $e_{i+1} = (x^{(i+1)}, y^{(i+1)})$ belong to the same block of G for all $1 \leq i < p$. This is equivalent to the requirement that e_1 belongs to the same block as e_p. Thus, it suffices to determine and test e_1 and e_p and to test (Q1) and (Q3).

By definition all edges e_i are solid intercluster tree edges, i.e., tree edges on the T_2-path between the node C_u in H_2 and the node C_v in H_2. Therefore, we keep the following data structure.

(HL9) We store a least common ancestor data structure [10] for T_2 rooted at a leaf R, such that least common ancestor queries between any two nodes of H_2 can be answered in constant time. If C is the least common ancestor of C and D, then the data structure also returns in constant time the tree edge, incident to C on $\pi(C, D)$. We also keep at each node of H_2 the tree edge to its parent.

(HL10) We store at each solid intercluster tree edge its block in G.

Both data structures are recomputed from scratch after each update in G. The computation of (HL10) proceeds in the same way as a complete block query: we perform a dfs on T_2 that determines the sets of solid intercluster tree edges that belong to the same block. This takes time $O(m/k)$. The time to build both data structures is thus $O(m/k)$.

Determining e_1, e_p, $y^{(1)}$, *and* $x^{(p)}$: We first use (HL9) to determine the least common ancestor of C_u and C_v in H_2. If C_u is the least common ancestor, the data structure returns the tree edge incident to C_u on $\pi(C_u, C_v)$. This is edge e_1; the edge from C_v to its parent is the edge e_p. If the least common ancestor of C_u and C_v is a third node, then e_1 is the edge from C_v to its parent and e_p is the edge from C_u to its parent. This also provides $y^{(1)}$ and $x^{(p)}$.

Testing conditions (Q1) *and* (Q3): We test conditions (Q1) and (Q3) in constant time as described in section 2.3.

Testing condition (Q2)*:* To test condition (Q2) we simply test with (HL10) whether e_1 and e_p belong to the same block.

THEOREM 2.53. *The given data structure can answer a biconnectivity query in constant time.* The total update cost is

(1) time $O(k)$ for restoring the relaxed partition,

(2) amortized time $O(k)$ to update all cluster graphs,

(3) amortized time $O((m/k)\log n + \sqrt{m})$ to update all shared graphs,

(4) time $O(k + (m/k)\log n)$ to update all data structures for high-level graphs (HL1)–(HL10), and

(5) time $O((m/k)\log n)$ to update all c-structures.

Thus, choosing $k = \sqrt{m \log n}$ gives the following update time.

THEOREM 2.54. *The given data structure can be updated in amortized time $O(\sqrt{m \log n})$ after an edge insertion or deletion.*

3. Plane graphs.

In this section we present an algorithm for fully dynamic biconnectivity in plane graphs with $O(\log n)$ query time and $O(\log^2 n)$ update time, where insertions are required to maintain the planarity of the embedding. We modify the extended topology tree data structure of [14] and prove that this data structure dynamically maintains biconnectivity information.

3.1. Definitions.

As in general graphs (see section 2) we transform a given graph G into a degree-3 graph G' by replacing every vertex x of degree $d > 3$ with a chain of $d-1$ *dashed* edges $(x_1, x_2), \ldots, (x_{d-1}, x_d)$. We say each x_i is a *representative* of x and x is the *original node* of every x_i. Then we find an embedding of G' and a spanning tree T' of G'. A *topology tree* of G' based on T' is a hierarchical representation of G' introduced by Frederickson [5]. On each level of the hierarchy it partitions the vertices of G' into connected subsets called *clusters*. An edge is *incident* to a cluster if exactly one endpoint of the edge is contained in the cluster. The *external degree* of a cluster is the number of tree edges that are incident to the cluster. Each vertex of G' is a level-0 cluster. Two clusters at level $i > 0$ are formed by either

(1) the union of two clusters of level $i-1$ that are joined by an edge in the spanning tree and either both are of external degree 2 or one of them has external degree 1, or

(2) one cluster of level $i-1$, if the previous rule does not apply.

Each cluster at level i is a node of height i in the topology tree. If a cluster C at level i is formed by two clusters A and B of level $i-1$, then A and B are the children of C in the topology tree. If C is formed by one cluster A of level $i-1$, then A is the only child of C in the topology tree. The topology tree has depth $D = O(\log n)$ [5]. In the following, *node* denotes a vertex of the topology tree.

In [14] the topology tree data structure is extended to maintain non-tree edges of G' and additional connectivity information at each node, called *recipe*. We use the same technique to maintain dynamic 2-vertex connectivity.

Every insert(u,v), delete(u,v), or query(u,v) operation requires that the topology tree is *expanded* at an (arbitrary) representative of u and of v: we mark all clusters containing the two representatives in the topology tree. Note that all these clusters lie on a constant number of paths to the root. Then we build the graph which consists of the two representatives and a compressed representation of all the clusters that are unmarked children of a marked node in the topology tree. This creates a compressed version of G, called $G(u, v)$, of size $O(\log n)$. This graph is used to answer queries. In the case of update operations, the edge is added to or deleted from $G(u, v)$. Afterward the topology tree is *merged together* again, i.e., a topology tree representation is created for the (possibly modified) graph $G(u, v)$.

To add non-tree edges to the topology tree data structure we define a bundle between two clusters C and C' as follows: If neither C is an ancestor of C' nor vice versa, let $e(C, C')$ be the set of all edges between C and C'. Otherwise, assume w.l.o.g. that C' is the ancestor of C. We define $e(C, C')$ to be the set of all edges incident to C whose least common ancestor in the topology tree is C'. Since we are considering an embedded graph, the edges incident to a cluster C are embedded at C in a fixed circular order. A *bundle* between a cluster C and C' is a subset of $e(C, C')$ that forms a maximal continuous subsequence in the circular order at C and C'. Note that this definition is independent of the level of the clusters and planarity guarantees that there are at most three bundles between two clusters [14]. The first and last edge of a bundle in this order are called the *extreme* edges of the bundle. In the topology tree a bundle between C and C' is represented by two bundles, one from C to the least common ancestor of C and C' (called the *LCA-bundle of* C) and one from C' to the least common ancestor. Whenever the topology tree is expanded and the graph $G(u, v)$ is created, we convert these two bundles back into one.

An edge (u, v) with $u, v \in C$ is called an *internal edge* of the cluster C. Assume all dashed internal edges of C are contracted. The *projection* of an edge (x, y) onto a tree path P is the path $\pi(x, y) \cap P$. Note that, by definition, the vertices of each cluster are connected by a subtree of T'. In the following we define the projection edge of an edge, the projection path $p(C)$, and the coverage graph of C which consists of small and big supernodes of C. All these definitions are independent of the level of the cluster.

(1) If C has external degree 1, the *projection path* $p(C)$ of C consists of the endpoint z of the (unique) tree edge incident to C. This endpoint is a *small supernode.* The *coverage graph* of C consists of this supernode and of all LCA-bundles of C. For each edge e incident to C where y is the endpoint in C, the *projection edge* of e is e if $y = z$ and otherwise the tree edge incident to z that lies on $\pi(y, z)$.

(2) If C has external degree 3, it consists of only one vertex z. Both the *projection path* $p(C)$ and the *coverage graph* consist of only this one vertex which is a *small supernode.*

(3) If the external degree of a cluster C is 2, there is a unique simple tree path between the tree edges that are incident to C. This path is the *projection path* $p(C)$ of C. The *projection* $p(x)$ of a vertex x in C is the vertex closest to x on the projection path. The *projection edge* of a vertex x is the edge on $\pi(x, p(x))$ incident to $p(x)$. If $x = p(x)$, the projection edge of x is undefined. The *projection edge* of an edge (x, y) with one endpoint x in C is the projection edge of x, if it is defined and it is (x, y) otherwise. The *projection edges* of an edge (x, y) with $x, y \in C$ are the projection edge of x if it is defined and (x, y) otherwise and also the projection edge of y if it is defined and (x, y) otherwise.

If (x, y) is an internal edge of C, then the subpath $\pi(x, y) \cap p(C)$ is the *projection* of (x, y) on $p(C)$, $p(x)$ and $p(y)$ are the *extreme vertices* of the projection, and all vertices on the subpath except for $p(x)$ and $p(y)$ are the *internal vertices* of the projection.

Let (w, z) and (x, y) be the extreme edges of an LCA-bundle between a cluster C and a cluster C' with $w, x \in C$ and $z, y \in C'$. The path $\pi(w, x) \cap p(C)$ is called the *projection* of the edge bundle on $p(C)$, $p(w)$ and $p(x)$ are called the *extreme vertices* of the projection, and all vertices on the subpath except $p(w)$ and $p(x)$ are *internal vertices* of the projection. The *projection edges* of a bundle are the projection edges of the extreme edges of the bundle.

The *coverage graph* of C is built by compressing $p(C)$ as follows:

(1) Let $u_1, u_2, \dots, u_p$ be a maximal subpath of $p(C)$ such that

(a) $\pi(u_1, u_p)$ intersects the projection of an LCA-bundle on $p(C)$,

(b) u_1 is the extreme vertex of the projection of an LCA-bundle or an internal edge,

(c) u_p is the extreme vertex of the projection of an LCA-bundle or an internal edge, and

(d) every vertex u_i for $1 < i < p$ is an internal vertex of the projection of a bundle or an internal edge, or there exist two projections with projection node u_i and the same projection edge at u_i such that u_i and u_j with $j < i$ are the extreme vertices of one projection and u_i and u_k with $k > i$ are the extreme vertices of the other projection.

If $p > 2$, we contract the path $u_2, \dots, u_{p-1}$ to one vertex u, called *big supernode*, and we say $u_2, \dots, u_{p-1}$ are *replaced* by the big supernode. The vertices u_1 and u_p are called *small supernodes* and the edges (u_1, u) and (u, u_p) are called *superedges*. All edges incident to $u_2, \dots, u_{p-1}$ are now incident to u. This splits a bundle that is incident to u_1 and/or u_p and also u_i with $1 < i < p$ into up to three *subbundles*, one incident to u and the other(s) incident to u_1 and/or u_p. If the edge (u_1, u_2) (resp., (u_{p-1}, u_p)) is dashed, then the edge (u_1, u) (resp., (u, u_p)) is dashed.

If $p \leq 2$ then no nodes are compressed.

(2) After replacing all subpaths that fulfill condition 1, let $v_1, v_2, \dots, v_q$ be a subpath of $p(C)$ such that v_1 and v_q are two small supernodes and no vertex v_i with $1 < i < q$ is a supernode. We contract the path $v_2, \dots, v_{q-1}$ to one *superedge* (v_1, v_q) and we say $v_2, \dots, v_{q-1}$ are replaced by the superedge. If all edges $(v_1, v_2), \dots, (v_{q-1}, v_q)$ are dashed, then the superedge is dashed; otherwise it is solid.

The *coverage graph* of C consists of this compressed representation of $p(C)$ and all LCA-bundles grouped into sets according to their projection edges.

Note that our definition of a supernode replaces a supernode of [14] by two small and one big supernode and each bundle is split into at most three *subbundles*, one incident to each small supernode and one incident to the big supernode.

When expanding the topology tree, we build the coverage graph for each node that was marked and each child of a marked node. For each subbundle that is incident to a supernode in a coverage graph we maintain its projection edges implicitly as described below.

The coverage graph of a cluster C is maintained as a doubly linked path of supernodes. Each supernode stores up to two doubly linked lists of projection edges incident to it (called *projection list*), one list for each side of the tree path $p(C)$. Each projection edge e stores a doubly linked list of the subbundles such that e is the projection edge of the subbundle. If C has external degree 1, there is only one supernode and only one list of projection edges. If C has external degree 3, it consists of only one supernode without any projection edges or subbundles. The projection edges and the subbundles are listed in the counterclockwise order of their embedding. Only the first and last subbundles in a list have direct access to the projection edge and only the first and last projection edges in a list have direct access to the supernode to which they are incident. The data structure lets us coalesce two adjacent supernodes or two projection lists into one in constant time; we can also split a supernode or a projection edge list into two in constant time if we are given pointers that tell where to split the lists. Note that each subbundle can be contained in at most two lists and if it is contained in two lists, it is the first element of the one and the last element of the other list.

3.2. Recipes. Each node in the topology tree is enhanced by a *recipe* that describes how the coverage graph of the children of the node can be created from the coverage graph of the node. The only difference in the algorithm of [14] and this biconnectivity algorithm is in the contents of the recipes. We describe our recipes in the following. A recipe contains four kinds of instructions:

(1) *Split a subbundle.* Replace a subbundle of m edges that have the same target by up to four adjacent subbundles that have that target and whose (specified) sizes sum to m.

(2) *Split a projection edge.* Split the subbundle list at specified locations, and replace the old subbundle list at the supernode by the new subbundle lists.

(3) *Split a supernode.* Split the two projection lists on either side of the supernode into two pieces at specified locations. Replace the old supernode by two new ones linked by a superedge, and give the appropriate piece of each projection list to each of the new supernodes.

(4) *Create a new subbundle.* Create a subbundle with a specified target and number of edges, and insert it at a specified place in a subbundle list of at most two projection edges.

Using these instructions the coverage graphs of the children of a cluster C can be transformed into a coverage graph of C. The sequence of instructions together with the appropriate parameters (e.g., which subbundle list has to be split at which location) is called a *recipe* and is stored at the node in the topology tree that represents C. These parameters are either a record of a subbundle (consisting of the number of edges in the subbundle and its target), a record of a projection edge (consisting of the edge), or a pointer, called *location descriptor.* A location descriptor consists of a pointer to a subbundle and an offset into the subbundle (in terms of number of edges) or a pointer into a projection list. It takes constant time to follow a location descriptor.

Whenever we expand the topology tree, we use the recipes to create the coverage graphs along the expanded path. Whenever we merge the topology tree, we first determine how to combine the coverage graphs of two clusters to create the coverage graph of their parent, and then we remember how to undo this operation in a recipe. We now describe the instructions in the recipe of C, depending on the number of children of C and their external degrees. In the following *subbundle* stands for LCA-subbundle.

Case 1: C has only one child. In this case the coverage graph of C is identical to the coverage graph of its child. The recipe is therefore empty.

Case 2: C has two children with external degrees 3 and 1. Let Y be the child with external degree 3 and let Z be the child with external degree 1. The coverage graph of Y and of Z consists of one supernode. We build the coverage graph of C as follows:

If the tree edge between Y and Z is dashed, we simply contract it by making the projection list of Z the projection list of one side of the path of C. The projection list of the other side is empty. The projection edges of the bundles do not change and, thus, the subbundle lists do not change.

If the edge (Y, Z) is not dashed, then the supernode of C has only one projection edge, namely, the tree edge between Y and Z. Thus, the supernode of C has one projection list (the projection list of the other side is empty) containing one projection edge. The subbundle list of this projection edge consists of the concatenation of all subbundle lists of Z. In the recipe we use location descriptors to point to the locations of the concatenation. The number of location descriptors is proportional to the number of removed projection edges.

Case 3: C has two children, both with external degree 1. In this case C is the root of the topology tree. Its coverage graph is empty. The coverage graphs of the children contain one supernode and at most one subbundle apiece, corresponding to the set of non-tree edges linking the children. Since each subbundle is contained in at most two projection lists, there are at most four projection lists. The recipe stores these projection lists (i.e., whether a bundle is contained in one or two lists) and subbundles (i.e., the number of non-tree edges linking the children).

Case 4: C has two children with external degrees 2 and 1. Let Y be the child of degree 2 and Z be the child of degree 1. We collapse all supernodes of Y to one supernode s to build the coverage graph of C from the coverage graph of Y as follows:

On each side of the tree edge between Y and Z there may be a subbundle that connects Y and Z. We remove these subbundles and make all remaining subbundles incident to s.

If the edge (Y, Z) is dashed, then the projection edge of the subbundles incident to Y does not change. Thus, we concatenate the two projection lists of Y and the projection list of Z (in the order of the embedding). This creates a single supernode with a single projection list.

If the edge (Y, Z) is solid, then this edge becomes the projection edge for all subbundles incident to Y. Thus, we concatenate all bundle lists of all projection edges of Y to create the bundle list for (Y, Z). Then we concatenate the two projection lists of Y and the projection list of Z (in the order of the embedding). This creates a single supernode with a single projection list.

In both cases, if two newly adjacent subbundles have the same target, we merge them into one subbundle and update the subbundle and projection lists appropriately.

In the recipe we need a location descriptor to point to each subbundle where we concatenated projection lists or subbundle lists or merged subbundles. We also have to store any subbundles that connect Y and Z and all projection edges that we removed. The number of location descriptors we store is proportional to the number of supernodes of Y plus the number of removed projection edges.

Case 5: C has two children, both with external degree 2. Let Y and Z be the children of C. To join the coverage graphs of Y and Z we consider two cases: if the tree edge between Y and Z is dashed, we join the two coverage graphs by identifying the appropriate small supernodes (that are terminating the coverage graphs) and concatenating their projection lists. If the tree edge between Y and Z is not dashed, we connect the two coverage graphs by an edge.

In both cases we then remove all subbundles between Y and Z. If one of the supernodes that was incident to a removed subbundle is no longer incident to a bundle, we replace it by a superedge. Afterward we coalesce all the supernodes between the (Y, Z)-subbundle endpoints into three supernodes as follows: If the path P between their endpoints contains only one supernode other than the endpoints, nothing has to be done. Otherwise, we replace these (at least two) supernodes by one supernode by concatenating their projection lists. We also merge newly adjacent subbundles into a single subbundle if they have the same target.

The recipe contains a location descriptor pointing to each subbundle where we coalesced supernodes and concatenated projection lists (and possibly merged adjacent subbundles). We also store the subbundles that were merged together or deleted. If there is a subbundle that loops around the tree, we need two more location descriptors to mark its endpoints. The number of location descriptors is proportional to the number of coalesced supernodes in Y and Z.

New subbundles may be created during recipe evaluation. For each new subbundle, the recipe stores a bundle record, preloaded with the count of bundle edges, and a location descriptor pointing to the place in the old subbundle list where the new subbundle is to be inserted. The target field of the subbundle is easy to set: the least common ancestor of the bundled edges is exactly the node at which the recipe is being evaluated. In a way similar to [14] we can show the following lemma.

LEMMA 3.1. *If the topology tree is expanded at a constant number of vertices and recipes are evaluated at the expanded clusters, the total number of edge bundles, supernodes, and superedges created is $O(\log n)$. The expansion takes $O(\log n)$ time.*

Proof. Since the topology tree has depth $O(\log n)$, there are $O(\log n)$ marked nodes and $O(\log n)$ children of marked nodes. Thus, the cluster graph consists of the coverage graph of $O(\log n)$ clusters. Planarity guarantees that these clusters are connected by $O(\log n)$ bundles; each bundle is split into up to three subbundles. Thus, there are $O(\log n)$ subbundles. Since each supernode in a cluster with more than one supernode is incident to a subbundle, there are $O(\log n)$ supernodes. Because the supernodes and superedges form a tree, the number of superedges is also $O(\log n)$. Each subbundle has two projection edges. Thus, the total number of projection edges is $O(\log n)$.

Evaluating a recipe takes time proportional to the number of supernodes or projection edges created by the recipe plus constant "overhead" time. Thus, the total expansion time is $O(\log n)$. $\square$

3.2.1. Queries. To answer a query (u, v), we mark all the clusters containing u and v in the topology tree. Then we create the graph $G(u, v)$ in the following steps:

(1) We build the cluster graph by expanding the topology tree at a representative of u and of v.

(2) Let $e_1, e_2, \dots, e_p$ with $p > 1$ be all the subbundles whose extreme edges have the same projection edge (x, y) in a cluster C with $x \in p(C)$. We add a small supernode y and connect all these extreme edges to y.

(3) We contract all dashed edges. When contracting a dashed edge between two supernodes, the resulting supernode is a small supernode.

Since the cluster graph consists of $O(\log n)$ supernodes, subbundles, and superedges and can be computed in time $O(\log n)$, the graph $G(u, v)$ resulting from these 3 steps contains $O(\log n)$ supernodes, subbundles, and superedges and can be computed in time $O(\log n)$.

The following lemmas show that two vertices u and v are not biconnected in G iff there is an articulation point in $G(u, v)$ separating u and v that is not a big supernode. Since the cluster graph has size $O(\log n)$ this can be tested in time $O(\log n)$.

LEMMA 3.2. *Let u and v be two vertices of G_2 and of G_1 and let G_2 be a graph created from G_1 by*

(1) *contracting connected subgraphs into one vertex,*

(2) *replacing the only two edges* (a, b) *and* (b, c) *incident to a vertex* b *by the edge* (a, c),

(3) *replacing parallel edges, and*

(4) *removing self-loops.*

Let x *be a vertex of* G_1 *that is not contained in the contracted subgraphs and not a removed degree-2 vertex. Then* x *is an articulation point in* G_1 *separating* u *and* v *iff* x *is an articulation point separating* u *and* v *in* G_2.

Proof. Consider first the case that x separates u and v in G_1. To achieve that u and v are not separated by x in G_2 a cycle has to be created that contains u, x, and v. Contracting pieces of G_1 that do not contain x or removing degree-2 vertices (other that x) cannot create new cycles. Thus, x is also an articulation point separating u and v in G_2.

If x separates u and v in G_2, then expanding vertices (other than x) of G_2 to connected subgraphs, replacing one edge by two edges and a degree-2 vertex, or adding parallel edges to edges not on $\pi(u, v)$ and self-loops does not create a cycle that contains x, u, and v. Thus, x separates u and v also in G_1. $\square$

LEMMA 3.3. *Let* u *and* v *be two vertices of* G. *The graph* $G(u, v)$ *is created from* G *by*

(1) *contracting connected subgraphs into one vertex,*

(2) *replacing the only two edges* (a, b) *and* (b, c) *incident to a degree-*2 *vertex* b *by the edge* (a, c),

(3) *collapsing parallel edges, and*

(4) *removing self-loops.*

No small supernode on $\pi(u, v)$ *(except for* u *and* v *itself) in* $G(u, v)$ *is contained in a contracted subgraph of any of these operations.*

Proof. The graph $G(u, v)$ can be created from G by the three operations given in the lemma using the following steps. Note that $G(u, v)$ does not contain dashed edges and every small supernode of $G(u, v)$ represents a unique vertex x of G.

(1) Mark as red all the nodes that are small supernodes of $G(u, v)$.

(2) Collapse all nodes on the tree path between two red nodes to one blue node.

(3) Contract every blue node and all the subtrees whose roots are uncolored and connected to the blue node by a tree edge to a green node.

Now we are left with red, green, and uncolored nodes and every green node is connected by tree edges to two red nodes.

(4) Replace all parallel edges by one edge and remove all self-loops.

(5) Replace every degree-2 green node by a superedge. (All remaining green nodes correspond to big supernodes.)

(6) If a red node x lies on $\pi_G(u, v)$ and does not lie on $\pi(u, v)$, shrink all subtrees whose roots are uncolored and connected by a tree edge to x to a yellow node. Otherwise contract all subtrees whose roots are uncolored and connected to x by a tree edge to the node x.

(7) Replace all parallel edges by one edge and remove all self-loops.

The resulting graph is $G(u, v)$. Note that u and v are small supernodes in $G(u, v)$ and then marked red. Hence, if a small supernode x lies on $\pi(u, v)$, it is not replaced by step 6. No small supernodes are contained in a connected subgraph that is contracted in steps (1)–(5). The lemma follows. $\square$

LEMMA 3.4. *No vertex on a subpath that is replaced by a big supernode in* $G(u, v)$ *is an articulation point separating* u *and* v *in* G.

Proof. Let C be the cluster of $G(u, v)$ containing a vertex x that is replaced by a big supernode. Since x is replaced by a big supernode, it follows that x is an internal vertex of the projection path P creating this supernode. Let z_1 and z_2 be the extreme vertices of P. Then z_1 and z_2 are connected by a path in G that does not use x. It follows that x does not separate u and v in G. ∎

Lemma 3.5. *If a vertex x of G is replaced by a superedge (y, z) and if x separates u and v in G, then y and z also separate u and v in G.*

Proof. Let C be the cluster of $G(u, v)$ that contains x, and let $v_1, v_2, \ldots, v_q$ be the subpath P that is replaced by (y, z) with $y = v_1$ and $z = v_q$ and $x = v_i$ for some $1 < i < q$. W.l.o.g. let the tree path from v_2 to u contain v_1. From the definition of a superedge it follows that no vertex v_i with $1 < i < q$ is a supernode. Thus, the projection of none of the subbundles incident of C (i.e., edges with one endpoint in C) contains a vertex v_i. Since x is an articulation point separating u and v, no edge with both endpoints outside C exists whose projection on $\pi(u, v)$ contains a node v_i for $1 \leq i \leq q$. Since v_1 is a small supernode, it is the extreme vertex of a projection of an edge or subbundle whose projection onto $\pi(u, v)$ lies inside $\pi(v_1, u)$. Thus, no edge or subbundle exists whose projection onto $\pi(u, v)$ contains a vertex on $\pi(v_1, u)$ other than v_1 and v_2. Additionally, if such a projection contains v_1 it does not have the same projection edge as any edge whose projection contains v_2. Thus, every path from u to v_2 contains y and, hence, every path from u to v contains y. The symmetric argument shows that z separates u and v in G. ∎

Lemma 3.6. *Two vertices u and v are not biconnected in G iff there is an articulation point separating u and v that is not a big supernode in the cluster graph $G(u, v)$.*

Proof. Lemma 3.3 shows that the cluster graph $G(u, v)$ is created from G by contracting subgraphs, removing degree-2 nodes, collapsing parallel edges, and removing self-loops. Thus, Lemma 3.2 does apply with $G_1 = G$ and $G_2 = G(u, v)$.

Let x be an articulation point separating u and v in G. Then x lies on $\pi(u, v)$. From Lemma 3.4 it follows that x is cannot be represented by a big supernode in $G(u, v)$. If x is represented by a small supernode, then according to Lemma 3.3, x was not affected by the contraction of G to $G(u, v)$. Thus, Lemma 3.2 shows that x is an articulation point separating u and v in $G(u, v)$. If x is represented by a superedge (y, z), then according to Lemma 3.5 y is an articulation point separating u and v in G as well. Since y is a small supernode, the same argument as above shows that y separates u and v in $G(u, v)$.

If a small supernode x is an articulation point separating u and v in $G(u, v)$, then by Lemma 3.3 x was not part of a contracted subgraph. It follows from Lemma 3.2 that x is an articulation point separating u and v in G. ∎

Theorem 3.7. *The given data structure can answer biconnectivity queries in time $O(\log n)$.*

Proof. Lemma 3.6 shows that to test the biconnectivity of u and v in G it suffices to test whether u and v are separated by a small supernode in $G(u, v)$. Since $G(u, v)$ has size $O(\log n)$, this can be done in time $O(\log n)$. ∎

3.3. Updates. An insert(u,v) or query(u,v) operation consists of three steps. First, the topology tree is expanded at a representative of u and of v to create the cluster graph as discussed in section 3.2. Second, we add or remove the edge (u, v) from the cluster graph. Third, we merge the topology tree back together.

By adding or deleting a constant number of vertices and edges we guarantee that the graph stays a degree-3 graph. Note that if a tree edge is deleted, we run along the faces adjacent to (u, v) to find a subbundle that connects the two disconnected spanning trees. We can determine one of the edges of the subbundle by repeatedly expanding the clusters containing the endpoints. This edge becomes the new tree edge.

The details of merging the topology tree back together are given in [14]. There are three basic steps. First, the new topology tree for the updated cluster graph is computed. Second, the new subbundles and their LCA-targets are computed. Third, the recipes in all clusters that are affected by the modification of subbundles are recomputed. Steps one and two are identical to [14] and take time $O(\log n)$.

In step three of [14] the recipe of clusters is recomputed that contain the endpoint of an extreme edge of a modified subbundle. The following lemma shows that with the recipes described in section 3.2 it suffices to update these clusters also for 2-vertex connectivity. Thus, the same algorithm as in [14] can be used to update the data structure after each update operation.

LEMMA 3.8. *If a subbundle is split into a constant number of subbundles or if a constant number of subbundles is merged, the only recipes that have to be updated are the recipes of clusters containing the endpoints of the extreme edges of the modified subbundles.*

Proof. A recipe at a cluster C contains location pointers into subbundles, subbundle lists, and projection lists. Additionally, it contains subbundle records and projection edges.

All subbundles in the subbundle lists of C are incident to C. All projection edges for whom we keep a projection list at C are projection edges of subbundles whose extreme edges have at least one endpoint in C. These lists have to be updated only if one of these subbundles is modified.

For each subbundle whose record is stored in the recipe or that is pointed to by a location descriptor at least one of the endpoints is contained in C. The record has to be updated only if the subbundle is modified.

Each projection edge that is stored in the recipe is the projection edge of a subbundle whose extreme edges have at least one endpoint in C. The projection edge information changes only if this subbundle is modified. ☐

This results in the following theorem.

THEOREM 3.9. *The given data structure can answer biconnectivity queries in time $O(\log n)$ and can be updated in time $O(\log^2 n)$ after an edge insertion or deletion under the requirement that the edge insertions maintain the planarity of the embedding.*

REFERENCES

[1] D. EPPSTEIN, G. F. ITALIANO, R. TAMASSIA, R. E. TARJAN, J. WESTBROOK, AND M. YUNG, *Maintenance of a minimum spanning forest in a dynamic planar graph*, J. Algorithms, 13 (1992), pp. 33–54.

[2] D. EPPSTEIN, Z. GALIL, AND G. F. ITALIANO, *Improved Sparsification*, Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA, 1993.

[3] D. EPPSTEIN, Z. GALIL, G. F. ITALIANO, AND A. NISSENZWEIG, *Sparsification—a technique for speeding up dynamic graph algorithms*, J. ACM, 44 (1997), pp. 669–696.

[4] D. EPPSTEIN, Z. GALIL, G. F. ITALIANO, AND T. H. SPENCER, *Separator-based sparsification* II: *Edge and vertex connectivity*, SIAM J. Comput., 28 (1999), pp. 341–381.

[5] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[6] G. N. FREDERICKSON, *Ambivalent data structures for dynamic* 2*-edge-connectivity and k smallest spanning trees*, SIAM J. Comput., 26 (1997), pp. 484–538.

[7] M. R. Henzinger and M. L. Fredman, *Lower bounds for fully dynamic connectivity problems in graphs*, Algorithmica, 22 (1998), pp. 351–362.

[8] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for* 2*-edge connectivity*, SIAM J. Comput., 21 (1992), pp. 1047–1069.

[9] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[10] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[11] M. R. Henzinger, *Fully dynamic biconnectivity in graphs*, Algorithmica, 13 (1995), pp. 503–538.

[12] M. R. Henzinger and V. King, *Maintaining minimum spanning trees in dynamic graphs*, in Proceedings of the 24th International Colloquium on Automata, Languages, and Programming, Bologna, Italy, 1997, pp. 594–604.

[13] M. R. Henzinger and H. La Poutré, *Certificates and fast algorithms for biconnectivity in fully-dynamic graphs*, in Proceedings of the Third Annual European Symposium on Algorithms, Patras, Greece, 1995, pp. 171–184.

[14] J. Hershberger, M. Rauch, and S. Suri, *Fully dynamic* 2*-edge-connectivity in planar graphs*, Theoret. Comput. Sci., 130 (1994), pp. 139–161.

[15] J. Holm, K. de Lichtenberg, and M. Thorup, *Poly-logarithmic fully-dynamic algorithms for connectivity, minimum spanning tree,* 2*-edge, and biconnectivity*, in Proceedings of the 31st Annual Symposium on the Theory of Computing, Dallas, TX, 1998, pp. 79–89.

[16] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, *Complexity models for incremental computation*, Theoret. Comput. Sci., 130 (1994), pp. 203–236.

[17] M. H. Rauch, *Improved data structures for fully dynamic biconnectivity*, in Proceedings of the 26th Annual Symposium on the Theory of Computing, Montreal, Canada, 1994, pp. 686–695.

[18] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 1816–1838

OPTIMAL COMBINATORIAL FUNCTIONS COMPARING MULTIPROCESS ALLOCATION PERFORMANCE IN MULTIPROCESSOR SYSTEMS*

HÅKAN LENNERSTAD† AND LARS LUNDBERG‡

Abstract. For the execution of an arbitrary parallel program P, consisting of a set of processes with any executable interprocess dependency structure, we consider two alternative multiprocessors.

The first multiprocessor has q processors and allocates parallel programs dynamically; i.e., processes may be reallocated from one processor to another. The second employs cluster allocation with k clusters and u processors in each cluster: here processes may be reallocated within a cluster only. Let $T_d(P,q)$ and $T_c(P,k,u)$ be execution times for the parallel program P with optimal allocations. We derive a formula for the program independent performance function

$$G(k,u,q) = \sup_{\text{all parallel programs } P} \frac{T_c(P,k,u)}{T_d(P,q)}.$$

Hence, with optimal allocations, the execution of P can never take more than a factor $G(k,u,q)$ longer time with the second multiprocessor than with the first, and there exist programs showing that the bound is sharp.

The supremum is taken over all parallel programs consisting of any number of processes. Overhead for synchronization and reallocation is neglected only.

We further present a tight bound which exploits a priori knowledge of the class of parallel programs intended for the multiprocessors, thus resulting in a sharper bound. The function $g(n,k,u,q)$ is the above maximum taken over all parallel programs consisting of n processes.

The functions G and g can be used in various ways to obtain tight performance bounds, aiding in multiprocessor architecture decisions.

Key words. dynamic allocation, cluster allocation, static allocation, scheduling, multiprocessor, optimal performance, extremal combinatorics, combinatorial formula, 0,1-matrices, optimal partition

AMS subject classifications. 68M07, 68M20, 90B35, 68R05, 05D99

PII. S0097539799294398

1. Introduction.

It is relatively easy and cheap to build large multiprocessors, i.e., systems with a large number of processors, by connecting a number of small clusters with a communication network. The message passing interface (MPI) [16] and parallel virtual machine (PVM) [2] environments make it possible to write a parallel program which executes on a number of clusters. Systems which consist of a number of clusters can easily scale up; i.e., one can simply connect more clusters to the communication network. Another advantage of using multiple clusters is that communication within one cluster does not interfere with communication within other clusters. An alternative to having multiple small clusters is to have one large cluster. The processors in a cluster are often connected with a shared bus, which becomes a bottleneck when the number of processors grows. Consequently, systems with multiple small clusters have a number of advantages compared with systems consisting of one large cluster. A major disadvantage with multiple clusters is that, since processes

*Received by the editors April 9, 1999; accepted for publication May 14, 1999; published electronically April 4, 2000. The research of this paper was supported in part by the Blekinge Research Foundation.

http://www.siam.org/journals/sicomp/29-6/29439.html

†Department of Mathematics, University of Karlskrona/Ronneby, S-371 79 Karlskrona, Sweden (hakan@itm.hk-r.se).

‡Deptartment of Computer Science, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden (Lars.Lundberg@ipd.hk-r.se).

may not be reallocated between clusters, some clusters may be idle while others are very busy. In most cases the time when processors are idle can be reduced by careful allocation of processes to clusters.

The problem of finding allocations which result in minimum makespan, i.e., minimum completion time, is NP-hard [1]. In this report tight worst case bounds comparing the minimum makespan for a program on two multiprocessors are established. The multiprocessors have identical processors but different organization and different numbers of processors.

2. Problem definition and results.

The parallel program model under consideration is general. The multiprocessors execute parallel programs P which may have any set of processing times and any possible structure of interprocess dependency, only excepting deadlock.

A parallel program consists of a set of processes, some of which may run in parallel, and a set of synchronization signals between the processes, which introduce dependencies between the processes. A synchronization signal is a prohibition on one process at a specific time point to execute, unless another process has reached a certain time point; see section 4 for further details. We have no limitations on such dependencies in a program, other than that the program is required to be executable.

There are no restrictions on the numbers denoting processing times of the process parts or synchronization time points—the numbers are not required to be integers or multiples of a certain time unit. The only requirement is that all numbers are rational numbers, which from a practical point of view is a weak restriction.

The processors are assumed to be identical. In cluster allocation the processors are organized in k groups, the clusters, where each cluster contains u processors. Here we thus have in total ku processors. Once a process is allocated to a processor in a specific cluster it can only be executed on a processor in this cluster. It may be transferred any time, but only to a processor in the same cluster. The cost of transferring processes is neglected. If a process is put into a waiting state, it will thus later be restarted on a processor in the same cluster.

Dynamic and static allocations are both special cases of cluster allocation. Dynamic allocation represents the case of having all processors in one cluster, $k = 1$; hence the processes may be transferred between all processors without limitations. Static allocation can be described as the case when each cluster has one single processor, $u = 1$, so a process may never be transferred from the processor where it was initiated.

A cluster organized multiprocessor with k clusters, each containing u processors, will be compared with a multiprocessor with q processors using dynamic allocation, executing a program with n processes.

For a parallel program P, there is a certain number of different cluster allocations. Consider the simple example of a program P with three processes ($n = 3$) p_1, p_2, and p_3 and a multiprocessor with two clusters ($k = 2$). Then there are four different cluster allocations. One allocation is to put all processes on the same cluster. The other three allocations correspond to the three possible ways to put two processes on one cluster and one process on the other cluster. We may put p_1 and p_2 on the same cluster or we may put p_1 and p_3 on the same cluster or we may put p_2 and p_3 on the same cluster. The number of different cluster allocations is very large even for moderate values of n and k, however certainly finite.

Continuing the same example, assume that the structure of P is such that processes p_1 and p_2 execute for one time unit, and then they terminate. Neither p_1 nor

p_2 has to wait for any other process. Process p_3 may, however, not start its execution until p_2 has completed. Process p_3 then executes for one time unit before it terminates; i.e., the program contains one dependency—the dependency between p_2 and p_3. Consider the allocation where p_1 and p_2 are allocated to cluster one and p_3 to cluster two. At the start of the program both p_1 and p_2 can execute. Process p_3 may, however, not start its execution before p_2 has completed. If p_1 executes before p_2 on cluster one the makespan of the program will be three time units. However, if p_2 executes before p_1 the makespan will be two time units.

This example shows that the makespan of each allocation is affected by the order in which the processes allocated to the same cluster are executed. However, each of the allocations of the program has a well-defined minimal makespan. We are interested in the minimal makespans for two multiprocessor organizations. Therefore, we only need to consider the minimal makespan for each allocation.

Since we now have a well-defined minimal makespan for each allocation, the set of minimal makespans of the program P for all cluster allocations is finite, so it has a minimum. An allocation which results in a minimal makespan, i.e., shorter than or equal to the makespan of any other allocation, is called an optimal allocation.

For any fixed parallel program P, we compare the minimal makespans of P for two multiprocessors.

The first multiprocessor has q processors and allocates parallel programs dynamically. For a parallel program P, we denote the makespan for P with optimal dynamic allocation by $T_d(P, q)$.

The second multiprocessor performs cluster allocation with k clusters and u processors in each cluster. Let $T_{c,A}(P, k, u)$ denote the makespan for the parallel program P with the cluster allocation A. The makespan for the parallel program P with optimal cluster allocation is denoted by $T_c(P, k, u)$. Hence, $T_c(P, k, u) = \min T_{c,A}(P, k, u)$, where the minimum is taken over all possible cluster allocations A.

Overhead for synchronization and reallocation is neglected throughout the report.

We now define the performance function g:

$$g(n, k, u, q) = \max_{\text{all } n\text{-programs } P} \frac{T_c(P, k, u)}{T_d(P, q)}.$$

Here n-programs denote parallel programs with n processes. The formula for g is presented in section 6. Programs P for which we have equality, $g(n, k, u, q) = T_c(P, k, u)/T_d(P, q)$, are referred to as extremal programs.

The function $g(n, k, u, q)$ is also a tight bound for the same ratio taken over all programs with *at most* n processes. This follows from Theorem 6.4, where we show that the function $g(n, k, u, q)$ is increasing in the variable n.

This increasing property can furthermore be used to compute a formula for the process independent performance function

$$G(k, u, q) = \sup_{\text{all parallel programs } P} \frac{T_c(P, k, u)}{T_d(P, q)} = \lim_{n\to\infty} g(n, k, u, q).$$

The performance function $G(k, u, q)$ is applicable for a multiprocessor intended for any parallel program. This is the best possible bound in the case of no prior knowledge of the parallel programs. In the case when the number of processes of the parallel programs is a priori bounded to at most n processes, the function $G(k, u, q)$ is still an upper bound, but not tight. In this case the function $g(n, k, u, q)$ gives a tight bound.

Note that besides the allocations, the performance functions themselves are optimal, while representing bounds which cannot be improved. The term "optimal" thus appears in two senses. We have optimality as a minimum over allocations, and we have optimality as a maximum over programs.

This second sense of "optimal" appears in the expressions "optimal function" and "optimal performance function." However, when we refer to the bounds we use the term "tight bound."

3. Previous results and applications. The present report extends the results in [4]. Here the formula for the function g in the case $u = 1, k = q$ is treated only, representing static versus dynamic allocation on the same multiprocessor.

The mathematics of the subject is focused in this report. It can be viewed as the theoretical base for the reports [6], [8], and [10], which treat different applications from a computer science point of view.

The report [9] provides an application not covered by the present report. Here extra information is provided by a test execution of the program. In light of this information, the general bound is not tight. In [9], tight bounds for this situation are established. The results involve linear programming where values of the function G occur as coefficients.

Furthermore, the basic method presented here has proved to be useful in other contexts. In the report [11], the efficiency of cache memories for single processors is studied. Here tight bounds comparing more flexible with less flexible cache memory organization alternatives are derived. The final part of the argument is similar to that of the present report, while a different set of transformations and arguments is needed to reach the corresponding matrix problem. The report [7] is a survey article of the results in [11].

Other than the reports [4] to [15], the only general results concerning allocation strategies of parallel programs where synchronization is not neglected appear to be the results by R. L. Graham [3]. The overhead for process reallocation and synchronization is neglected also in this work. Here so-called list scheduling algorithms are considered. This term is used for dynamic allocation algorithms where, when a processor becomes idle and there are waiting executable processes, one of the executable processes is immediately allocated to the idle processor. It is established in [3] that the makespan for a program allocated with a list scheduling algorithm is never higher than two times the makespan with optimal dynamic allocation.

The most general result in this report is clearly the one represented by the function G. Here we have no demands on the program. The function g is restricted to programs with at most n processes. It is expected that the techniques presented here can be extended to take advantage of further kinds of program specifics, thus improving the bounds by keeping away from those programs which maximize the ratio studied in this work.

One example of a multiprocessor architecture feature which immediately follows from the function $G(k, u, q)$ is the number of extra processors which would compensate for a more static allocation. The concluding graphics section shows plots of this and many other ways to exploit the performance functions for multiprocessor architecture purposes.

We next give an overview of the report.

In the following section the allocation problem is described and analyzed in detail and transformed to a mathematical problem. In section 5 we give a full formulation of the mathematical problem and introduce necessary notation. In section 6 the formula for the program dependent performance function $g(n, k, u, q)$ is stated and proved. Basic properties of the function $g(n, k, u, q)$ are also established here. Section 7 deals with the general performance function $G(k, u, q)$. The report is concluded with a graphics section. The purpose of this section is twofold: to show the performance functions quantitatively and geometrically, and to suggest how the functions can be used to aid in multiprocessor design decisions.

4. From programs to matrices. A program P consists of n processes of possibly very different execution times. The processes are usually dependent on each other. One can expect dependencies of the type that process i cannot execute further at the time point t_i unless process j has reached the time point t_j. When process j has reached the time point t_j it is said to execute a synchronizing signal to process i, restarting this process. Certainly there can be many synchronizing signals to a time point t_j, in which case all have to be executed before the process restarts. The execution time of synchronizing signals is neglected. Most parallel programs contain many synchronizing signals. In this report any set of synchronization signals is allowed, except those which include a deadlock.

Now consider a parallel program P and a multiprocessor with q processors. Assume that we have found an optimal dynamic allocation, with makespan $T_d(P, q)$. This optimal dynamic allocation will be kept fixed during the rest of this section. Next we introduce a discretization of the time interval in m subintervals (t_i, t_{i+1}) of equal length, such that all synchronizing signals, process initiations, and process terminations appear on the time points t_i, where $t_i = \frac{i}{m} T_d(P, q), i = 0, \dots, m$. Consequently, there is no time point t: $t_i < t < t_{i+1}$ for any i so that a process of the program P starts or stops or a synchronization signal is executed at the time t. Obviously, all processes in the interval (t_{i-1}, t_i) are completed before any part of the processes corresponding to the interval (t_i, t_{i+1}) start when using this allocation, since this is so without the discretization. For instance, a process which is active from time t_{i-1} to time t_{i+1} completes the processing in the interval (t_{i-1}, t_i) before any processing corresponding to the interval (t_i, t_{i+1}) starts, and when the process starts processing the time interval (t_i, t_{i+1}), all processing corresponding to the interval (t_{i-1}, t_i) is completed.

Such a discretization is possible if all synchronizing signals and process terminations occur at rational time points, which we can assume. Therefore the discretization involves no approximation; exactly the same program is represented in a discrete way. Observe that m might be very large even if the program P is small and has a simple structure.

From the program P we next construct another program P' which will prolong or not affect the makespan using cluster allocation $T_c(P, k, u) \leq T_c(P', k, u)$ but leave the makespan using dynamic allocation unchanged, $T_d(P, q) = T_d(P', q)$.

The construction of P' is obtained by two changes of the program P: we introduce new synchronizing signals and prolong certain processes. At every time point t_i we introduce all possible synchronization between the processes. This means that the synchronization structure now requires that all processes in the interval (t_{i-1}, t_i) have to be completed before any part of the processes corresponding to the interval (t_i, t_{i+1}) may start. Since the execution time of synchronizing signals is neglected, this does not change the makespan with the fixed optimal dynamic allocation, which is $T_d(P, q)$. Further, all processors are made to be busy at all time intervals. This is achieved by, if necessary, prolonging some processes. However, no process is prolonged beyond $T_d(P, q)$; hence $T_d(P, q) = T_d(P', q)$. It is of no importance that the prolonging of processes can be made in many ways; many programs can play the role of P' to a specific program P.

By the construction we thus have $T_d(P, q) = T_d(P', q)$. However, since introducing more synchronization and prolonging processes never shortens the makespan, for other allocations the makespan is either increased or unchanged. In particular, for optimal cluster allocation we therefore have $T_c(P, k, u) \le T_c(P', k, u)$. Consequently,

$$\frac{T_c(P, k, u)}{T_d(P, q)} \le \frac{T_c(P', k, u)}{T_d(P', q)}.$$

Certainly there are programs P which are left unchanged by the above transformation: programs such that $P = P'$. Since these programs constitute a subset of the parallel programs we consider, we actually have

$$g(n, k, u, q) = \max_{P} \frac{T_c(P, k, u)}{T_d(P, q)} = \max_{P'} \frac{T_c(P', k, u)}{T_d(P', q)}.$$

Therefore, in order to calculate the maximum, only programs of the type P' need to be considered.

We next represent a program P' by an $m \times n$ matrix of zeros and ones only. Here each process is represented by a column, and each time period is represented by a row. The entry at the position (i, j) of the matrix is one if the jth process is active between t_{i-1} and t_i; if it is inactive the entry is zero. Each row contains exactly q ones, since each processor is constantly busy. Because of the complete synchronization, each row has to be completed before the next row may start.

Our next objective is to compute the makespan for cluster allocation of the program P'.

Since we consider a time ratio, the choice of time unit is immaterial. Then we can choose the time unit so that $T_d(P', q) = m$; hence the processing time for each time period using the optimal dynamic allocation is 1.

To compute the makespan for cluster allocations we need to decide how the n processes are to be allocated to the k clusters. In this case every process is to be executed within one cluster, so each cluster allocation corresponds to a way of grouping the n processes onto the k clusters. In the matrix formulation each cluster allocation corresponds to a way of grouping the n columns of the matrix together in k sets. Here each set represents one cluster.

Consider one such cluster allocation A. Assume that l of the processes which are allocated to a specific cluster are executable at a specific time interval. Within a cluster there are u processors and the processes are allocated dynamically. Since there is no synchronization within a time interval, the programs are independent and we will next see that the completion time of this cluster in this time interval is $\max(l/u, 1)$. The completion time clearly cannot be lower than l/u. Further, the completion time cannot be lower than 1 since no part of a process can be executed in parallel with another part of the same process. In the case $l \ge u$ the limit l/u is reached, which has been proved by McNaughton (see Theorem 3.1 in [17]).

Because of the complete synchronization at the time points t_i, processing in the next time interval cannot start before completion of the slowest cluster in the current time interval. Since the clusters are identical, the slowest cluster is the cluster with a maximal number of active processes at that time interval. Hence, we obtain the completion time of a time interval as the maximum of $\max(l_i/u, 1)$ over the k clusters, where $l_i, i = 1, \ldots, k$ is the number of active processes in cluster i during the time interval. The makespan with the cluster allocation A is the sum of these maxima. If we have found an allocation of the n columns together in k sets which minimizes the makespan using cluster allocation, this is an *optimal* cluster allocation, and the makespan is denoted $T_c(P', k, u)$.

5. The matrix problem. We have seen in section 4 that for the sake of computing $T_c(P', k, u)$ and $T_d(P', q)$, the program P' can be replaced by a certain $m \times n$ matrix P' of zeros and ones only. In this matrix each row has exactly q ones, and thus $n - q$ zeros, $1 \le q \le n$. We say that a matrix P' of this type is an n, q*-matrix.*

We call an n, q-matrix *complete* if all $\binom{n}{q}$ permutations of the q ones occur equally frequently as rows. For a complete matrix, the number of rows is thus necessarily divisible by $\binom{n}{q}$. When considering n and q fixed, complete n, q-matrices differ from each other only in that they may have different number of copies of each row, and that the rows may occur in different order. Here are examples of two complete $4, 2$-matrices ($n = 4$ and $q = 2$):

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}.$$

THEOREM 5.1. *If P' and Q' are complete n, q-matrices, then*

$$\frac{T_c(P', k, u)}{T_d(P', q)} = \frac{T_c(Q', k, u)}{T_d(Q', q)}.$$

Proof. The order of the rows does not affect the ratio $T_c(P', k, u)/T_d(P', q)$, since each row must be completed before the execution of the next row may start. Having x ($x > 1$) copies of each row will multiply both quantities $T_c(P', k, u)$ and $T_d(P', q)$ by x; hence this does not affect the ratio $T_c(P', k, u)/T_d(P', q)$. Consequently, the ratio $T_c(P', k, u)/T_d(P', q)$ is the same for all complete n, q-matrices. The theorem is proved. ∎

Consider a complete n, q-matrix P' and a partition A of the n column vectors into k sets. Each set is allocated to a cluster. Denote the makespan for partition A with $T_{c,A}(P', k, u)$. Denote the number of ones in cluster l at row j by $c(l, j)$. As described in the previous section, the makespan with cluster allocation using the partition A is then

$$T_{c,A}(P', k, u) = \sum_{j=1}^{m} \max_{l=1,\dots,k} (c(l, j)/u, 1).$$

THEOREM 5.2.

$$g(n, k, u, q) = \max_{\textit{all } n,q\textit{-matrices } P'} \frac{T_c(P', k, u)}{T_d(P', q)} = \frac{T_c(P'', k, u)}{T_d(P'', q)},$$

where P'' is a complete n, q-matrix.

Proof. Consider an arbitrary n, q-matrix P' of m rows. From P' we will produce a complete n, q-matrix P'' where

$$\frac{T_c(P', k, u)}{T_d(P', q)} \le \frac{T_c(P'', k, u)}{T_d(P'', q)}.$$

Since this can be done for any n, q-matrix P', and since P' itself may be complete, the theorem then follows by Theorem 5.1.

We create $n!$ copies P'_i of P', and in each copy we permute the columns according to one of the $n!$ possible permutations of n columns. Next we form an n, q-matrix P'' of $n!m$ rows by concatenating the $n!$ copies. It is easy to see that the matrix P'' is complete. We obviously have $T_d(P'', q) = T_d(P', q)n!$.

Now we will show that the makespan of each copy cannot be less than $T_c(P', k, u)$, using cluster allocation. Let A be an optimal partition for P''. Consider one of the parts P'_i of P'', and consider the quantity $T_{c,A}(P'_i, k, u)$. Since P'_i is produced from P' by permuting columns, and A is a specific partition of the columns of P'_i, we can find a partition of P', denoted A_i, containing exactly the same columns as the partition A of P'_i. This is an essential observation for this proof. Hence, $T_{c,A_i}(P', k, u) = T_{c,A}(P'_i, k, u)$. We can produce A_i by letting the permutation corresponding to P_i act on the partition A.

Therefore we obtain $T_c(P', k, u) \le T_{c,A}(P'_i, k, u)$ for all $i = 1, \dots, n!$. Then we also have $T_c(P', k, u) \le \sum_{i=1}^{n!} T_{c,A}(P'_i, k, u)/n!$.

Furthermore, $T_c(P'', k, u) = \sum_{i=1}^{n!} T_{c,A}(P'_i, k, u)$. Consequently, $T_c(P', k, u) \le T_c(P'', k, u)/n!$. Hence

$$\frac{T_c(P', k, u)}{T_d(P', q)} \le \frac{T_c(P'', k, u)}{T_d(P'', q)}.$$

Can there exist a noncomplete matrix Q' with a ratio $\frac{T_c(Q',k,u)}{T_d(Q',q)}$ larger than the complete matrices? No, since by duplicating the columns of Q' as above, we get a complete matrix Q'' with

$$\frac{T_c(Q', k, u)}{T_d(Q', q)} \le \frac{T_c(Q'', k, u)}{T_d(Q'', q)}.$$

The theorem is proved. ∎

In order to compute the functions g and G representing the rightmost side of the theorem, we therefore only need to consider complete n, q-matrices.

Consider a partition A, such that column vector v_1 is in set X and column vector v_2 is in set Y. Consider also another partition A' which is identical to A, with the exception that vector v_1 is in set Y and vector v_2 is in set X. In that case the symmetry of a complete n, q-matrix P' guarantees that $T_{c,A}(P', k, u) = T_{c,A'}(P', k, u)$. Consequently, $T_{c,A}(P', k, u)$ is only affected by the number of vectors in each set.

We want to choose the partition A so that $T_{c,A}(P', k, u)$ is minimal. The execution time with optimal cluster allocation is thus

$$T_c(P', k, u) = \min_{\text{all partitions } A} T_{c,A}(P', k, u).$$

We refer to partitions where the sizes of the sets in the partition differ as little as possible as *even partitions*.

If n/k is an integer w, every set in an even partition has w members. Denote the integer part of n/k, the floor function, by $\lfloor n/k \rfloor$, and the smallest integer greater than or equal to n/k, the ceiling function, by $\lceil n/k \rceil$. If n/k is not an integer, the sets in an even partition have $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ members.

Theorem 5.3. *All even partitions are optimal partitions for a complete n, q-matrix.*

Proof. Consider a complete matrix P and an arbitrary partition A. Suppose A is not even. Then there are two partition sets where one set contains at least two columns more than the other one. In this proof we will modify the partition A into a partition A' by moving one of the columns in the larger partition set to the smaller. We will prove that

$$T_{c,A}(P', k, u) \geq T_{c,A'}(P', k, u).$$

By repeating this argument, we finally obtain an even partition $\tilde{A}$ where

$$T_{c,A}(P', k, u) \geq T_{c,\tilde{A}}(P', k, u).$$

Since this can be done for any partition A, it follows that even partitions are optimal.

Consider a partition A where there are x column vectors allocated to set X, there are y column vectors allocated to set Y, and $x - y > 1$. Consider another partition A' which is identical to A, with the exception that one vector is moved from set X to set Y, resulting in the sets X' and Y'.

We order the vectors in such a way that vectors 1 to x are in set X and vectors $x + 1$ to $x + y$ are in set Y in partition A, and vectors 1 to $x - 1$ are in set X' and vectors x to $x + y$ are in set Y' in partition A'. Hence the xth column is moved from the first set to the second. The example below shows how this looks for one row ($x = 5$ and $y = 3$):

$$\underbrace{01011}_{X}\underbrace{111}_{Y}\cdots$$

$$\underbrace{0101}_{X'}\underbrace{1111}_{Y'}\cdots .$$

The rows in a complete n, q-matrix P contain all permutations. By Theorem 5.1 we may consider a complete n, q-matrix P where each permutation occurs exactly once: P has $\binom{n}{q}$ rows. We next consider a row r in the matrix P. We denote by $r(i)$ the ith element in the row r. Then, for each row r such that $\sum_{x+1}^{x+y} r(i) > \sum_{1}^{x} r(i)$, there is another row r' which is obtained by switching the entry at index i with the entry at index $x+y+1-i$ for $i = 1, \ldots, y$: $r'(i) = r(x+y+1-i)$ for $i = 1, \ldots, y$ and for $i = y + 1, \ldots, x + y$. For all other i we have $r'(i) = r(i)$. Note that two different such rows are never mapped to the same row. Below we have an example of a row r and the corresponding row r' for the case when $x = 5$ and $y = 3$:

$$r = \underbrace{01011}_{X}\underbrace{111}_{Y}\cdots$$

$$r' = \underbrace{11111}_{X}\underbrace{010}_{Y}\cdots .$$

We will show that if the shift from A to A' gives a problem at row r in that the maximum here increases, then we can find another row where the maximum necessarily decreases. Therefore the sum of maxima cannot increase, and we have $T_{c,A}(P, k, u) \geq T_{c,A'}(P, k, u)$.

Suppose that $\max_{l=1,\ldots,k}(c(l, r)/u, 1)$ is larger with partition A' than with partition A for row r. Then necessarily $r(x) = 1$, and the set Y' has more ones than the set X: $\sum_x^{x+y} r(i) > \sum_1^x r(i)$. Also, $\sum_x^{x+y} r(i)$ is larger than the sum corresponding to any other partition set.

Then, at the row r' the maximum $\max_{l=1,\ldots,k}(c(l, r')/u, 1)$ will necessarily be smaller. We have $r'(x) = r(x) = 1$, and from $\sum_x^{x+y} r(i) > \sum_1^x r(i)$ and the mapping from r to r' we get $\sum_1^x r'(i) \geq \sum_1^y r'(i) + 1 = \sum_x^{x+y} r(i) > \sum_1^x r(i) \geq \sum_x^{x+y} r'(i)$. The sum corresponding to any other partition set is unchanged since here $r'(i) = r(i)$ and $A' = A$.

Hence the sum of ones in X at row r' is larger than the sum in Y'. The sum in Y' is also larger than the sum in any other set at row r, from which it follows that the sum in X at row r' is larger than the sum corresponding to any other partition set at row r'. Hence the maximum at row r' decreases.

Consequently, $T_{c,A}(P, k, u) \geq T_{c,A'}(P, k, u)$.

The theorem is proved. $\square$

Referring to the definition of $g(n, k, u, q)$ in section 2, we may now conclude that it holds for the performance function $g(n, k, u, q)$ that

$$g(n, k, u, q) = \frac{T_c(P', k, u)}{T_d(P', q)} = \frac{T_{c,A}(P', k, u)}{T_d(P', q)},$$

where P' is any complete n, q-matrix and A is any even partition.

Then g can be obtained by explicitly calculating the ratio $T_{c,A}(P', k, u)/T_d(P', q)$ for some arbitrary complete n, q-matrix P' and an even partition A. Unfortunately, this is extremely inefficient, making it impossible to handle reasonable parameters n, k, u, and q. In the next section we present a formula for g which makes it possible to handle realistic values on n, k, u, and q.

The function g is defined only for $q \leq n$. This is due to the matrix formulation, since there cannot be more than n ones in each row of the matrix: n is the number of columns. From the parallel program application it is clear that the function g should be extended by $g(n, k, u, q) = g(n, k, u, n)$ when $q > n$. We here have more processors than processes in the dynamic case. If $q > n$ we have $q - n$ unnecessary processors, and we can safely reduce the number of processors from q to n.

6. Process dependent performance function.

For the next theorem we need the following three combinatorial functions. Let $I = \{i_1, \ldots, i_{k-1}\}$ be a decreasing finite sequence of nonnegative integers. Then we define the following:

$b(l, I)$ = the number of distinct integers in $\{l, i_1, \ldots, i_{k-1}\}$;

$a(l, I, j)$ = the number of occurrences of the jth distinct integer in $\{l, i_1, \ldots, i_{k-1}\}$, enumerated in size order, $1 \leq j \leq b(l, I)$;

$\pi(k, w, q, l)$ = the number of permutations of q ones distributed in kw slots, which are divided in k sets with w slots in each, such that the set with maximum number of ones has exactly l ones. Note that all parameters to π are integers.

We can now state the formula for $g(n, k, u, q)$, from which an explicit formula for the function $G(k, u, q)$ follows.

Theorem 6.1. *Given positive integers $n, k, q,$ and u where $n \geq q$, in the case where $w = n/k$ is an integer, we have*

$$g(n,k,u,q) = \frac{1}{u\binom{n}{q}} \sum_{l=1}^{\min(w,q)} \max(l,u)\pi(k,w,q,l).$$

If n/k is not an integer we let $w = \lfloor n/k \rfloor$ and denote the remainder of n divided by k by n_k. Then we have

$$g(n,k,u,q) = \frac{1}{u\binom{n}{q}} \sum_{l_1=\max(0,\lceil \frac{q-(k-n_k)w}{n_k} \rceil)}^{\min(w+1,q)} \sum_{l_2=\max(0,\lceil \frac{q-l_1 n_k}{k-n_k} \rceil)}^{\min(w,q-l_1)} \max(l_1,l_2,u)$$
$$\times \sum_{i=\max(l_1,q-l_2(k-n_k))}^{\min(l_1 n_k, q-l_2)} \pi(n_k, w+1, i, l_1)\pi(k-n_k, w, q-i, l_2).$$

The formula for π is given in the following lemma.

Lemma 6.2. *In the special cases $\min(q,w) < l$ and $q > kl$, $\pi(k,w,q,l) = 0$. Otherwise, if $k = 1$ and $q = l$, $\pi(1,w,q,l) = \binom{w}{l}$. In all other cases we have*

$$\pi(k,w,q,l) = \binom{w}{l} \sum_{I \in \mathcal{I}} \binom{w}{i_1} \cdot \ldots \cdot \binom{w}{i_{k-1}} \frac{k!}{\Pi_{j=1}^{b(l,I)} a(l,I,j)!}.$$

The set $\mathcal{I}$ is the set of all sequences of nonnegative integers $I = \{i_1, \ldots, i_{k-1}\}$ which are decreasing: $i_j \geq i_{j+1}$ for all $j = 1, \ldots, k-2$, bounded by $l : i_1 \leq l$, and have sum $q - l$: $\sum_{j=1}^{k-1} i_j = q - l$.

Proof of Theorem 6.1. We consider the smallest complete n, q-matrix P' possible, i.e., the complete n, q-matrix with $\binom{n}{q}$ rows.

We start by considering the case when n/k is an integer. In this case, the n column vectors can be mapped to k sets containing w vectors each. Choose the time unit so that $T_d(P', q) = \binom{n}{q}$. The processing time of each row is then $\max(l, u)/u$, where l is the maximum number of ones in one set for the row. Obviously, l is an integer in the interval 1 to $\min(w, q)$.

From the definition of the function π we know that for each specific l in the interval 1 to $\min(w, q)$, $\pi(k, w, q, l)$ denotes the number of rows with completion time $\max(l, u)/u$. Consequently,

$$T_c(P', k, u) = \frac{1}{u} \sum_{l=1}^{\min(w,q)} \max(l,u)\pi(k,w,q,l).$$

We get

$$g(n,k,u,q) = T_c(P',k,u)/T_d(P',q) = \frac{1}{u\binom{n}{q}} \sum_{l=1}^{\min(w,q)} \max(l,u)\pi(k,w,q,l).$$

The formula for the case when n/k is not an integer follows by considering n_k sets of size $w + 1$ and $k - n_k$ sets of size w. In this case the first sum runs over the number of ones the n_k first sets. The theorem is proved.

Proof of Lemma 6.2. We take care of the trivial cases first. We have $\pi(k, w, q, l) = 0$ if $w < l$ (l is too large for the partition sets), $q < l$ (there are not ones enough to get l ones in one set), or if $q > kl$ (there are at least $l + 1$ ones in some set). If $k = 1$ we have one partition set only, and we have $n = w$, so one of the conditions $q < l$ and $q > kl$ applies unless $q = l$. If $k = 1$ and $q = l$ we get $\pi(k, w, q, l) = \pi(k, n, q, q) = \binom{n}{q}$, so $g(n, 1, u, q) = \max(l/u, 1) = \max(q/u, 1)$.

The formula for π is obtained in the following way. Consider a complete matrix and an even partition of the columns in k sets. Each row corresponds to a sequence $I = \{i_1, \ldots, i_k\}$ of nonnegative integers, where each integer i_j denotes the number of ones in set j. The sequence clearly has sum equal to q. We assume that the sequence is decreasing; hence we may rearrange the order of the i_j's derived from the matrix to obtain a decreasing sequence. In this way the set of possible rows of n, q-matrices is grouped into subsets, where each subset corresponds to a certain sequence $I = \{i_1, \ldots, i_k\}$.

Note that for all rows in the subset represented by the sequence $I = \{i_1, \ldots, i_k\}$, the maximal number of ones in one set is i_1. To compute the sum of maxima for a complete matrix where each row occurs exactly once, we then only need to compute the number of rows in each subset.

The number of rows corresponding to a sequence I can be obtained by permuting the number of ones allocated to each set and by permuting the sets. Permuting within the sets causes a product of binomial coefficients: $\binom{w}{i_1}\binom{w}{i_2}\cdots\binom{w}{i_k}$. Since we know that $i_1 = l$ is the maximal integer, we can extract the factor $\binom{w}{l}$ and consider decreasing sequences of length $k - 1$ and with sum $q - l$. Permuting the sets gives a factor $k!$; however, since no new rows are obtained when permuting sets with equal number of ones, we have to divide by the number of permutations of equal sets. We therefore get the factor $k!/(\Pi_{j=1}^{b(l,I)} a(l, I, j)!)$. (The functions $a(l, I, j)$ and $b(l, I)$ have been defined previously.)

The summation in the formula for π sums the number of rows where the maximum number of ones in a set is l. The lemma is proved.

As described earlier, $g(n, k, u, q)$ is defined also in the case $q > n$; here we have $g(n, k, u, q) = g(n, k, u, n) = \max(\lceil \frac{n}{k} \rceil/u, 1)$. A consequence of this and of Theorem 6.1 is that for $k = 1$ we have $g(n, 1, u, q) = \max(\min(n, q)/u, 1)$.

In section 8, the 256 first values of $g(n, k, u, q)$, for $1 \le n, k, u, q \le 4$, are presented in Figure 8.2. Figure 8.3 shows three plots of g representing three cases of increasing degrees of cluster organization using the same number of processors.

The following lemma provides an algorithm which generates all decreasing sequences appearing in Theorems 6.1 and 7.1.

We say that the least decreasing sequence of length μ and sum σ is the sequence $\{\lceil \frac{\sigma}{\mu} \rceil, \ldots, \lceil \frac{\sigma}{\mu} \rceil, \lfloor \frac{\sigma}{\mu} \rfloor, \ldots, \lfloor \frac{\sigma}{\mu} \rfloor\}$. If σ_μ is the remainder when σ is divided by μ, the number of $\lceil \frac{\sigma}{\mu} \rceil$'s is σ_μ, and the number of $\lfloor \frac{\sigma}{\mu} \rfloor$'s is $\mu - \sigma_\mu$, making the sum of the sequence σ.

LEMMA 6.3. *Let λ and σ be nonnegative integers and μ be a positive integer such that $\lambda \le \sigma \le \lambda\mu$.*

Every sequence of μ integers in the interval $0 \le i \le \lambda$ which is decreasing, bounded by λ, and has sum σ is generated exactly once by the following algorithm:

(1) *Take I as the least decreasing sequence of length μ and sum σ.*
(2) *Find the* rightmost *position in I, say, j, which fulfills*
  (a) $i_j < l$,
  (b) $i_j < i_{j-1}$ *or* $j = 1$,

(c) $i_{j+1} > 0$.

The algorithm terminates if no such j can be found.

(3) *The next sequence is obtained from I by increasing the entry in position j by one and replacing the subsequence $\{i_{j+1}, \ldots, i_\mu\}$ with the least decreasing subsequence of length $\mu - j$ and sum $\sum_{k=j+1}^{\mu} i_k - 1$.*

(4) *Go to step* 2.

Proof. The lemma is proved on page 758 in [4].

Observe that if n/k is an integer, $\sum_{l=1}^{\min(w,q)} \pi(k, w, q, l) = \binom{n}{q}$. Otherwise we have

$$\sum_{l_1=0}^{\min(w+1,q)} \sum_{l_2=0}^{\min(w,q-l_1)} \sum_{i=\max(l_1, q-l_2(k-n_k))}^{\min(l_1 n_k, q-l_2)} \pi(n_k, w+1, i, l_1)\pi(k-n_k, w, q-i, l_2) = \binom{n}{q},$$

since we are only counting all permutations of the q ones in the n slots in different ways. Thus the formula can be regarded as a weighted average of the numbers $\{\max(l/u, 1) : l = 1, 2, \ldots, \min(\lceil n/k \rceil, q)\}$. In the case $u \geq \min(\lceil n/k \rceil, q)$ all these numbers are 1; hence we in this case obtain $g(n, k, u, q) = 1$. In multiprocessor terms this corresponds to a collapse of cluster allocation into dynamic allocation since then one cluster alone is equally large as the first multiprocessor.

For the sake of the following theorem, we remark that we use the term "increasing" in the nonstrict sense: $f(k)$ is increasing if $f(k) \leq f(k+1)$ for all k. We do not require $f(k) < f(k+1)$.

THEOREM 6.4. *The function $g(n, k, u, q)$ has the following properties:*

(1) *$g(n, k, u, q)$ is increasing in the variables n and q and decreasing in the variables u and k.*

(2) *Given positive integers ν, w, and u, we have*

$$\begin{aligned}
\max\left(\frac{w+1}{u}, 1\right) - \left(\max\left(\frac{w+1}{u}, 1\right) - \max\left(\frac{w}{u}, 1\right)\right)(1 - w^{-(w+1)})\nu \\
\geq \lim_{k\to\infty} g(wk + \nu, k, u, k) \\
\geq \max\Big(\max\left(\frac{w}{u}, 1\right), \max\left(\frac{w+1}{u}, 1\right) - \Big(\max\left(\frac{w+1}{u}, 1\right) \\
- 1\Big)(1^{-(w+1)})^\nu\Big).
\end{aligned}$$

(3) *The function $g(n, k, u, q)$ is unbounded.*

Note further that $g(n, k, u, q) = 1$ if $u \geq \min(\lceil n/k \rceil, q)$ or if $k \geq n$, and $g(n, k, u, q) = g(n, k, u, n) = \max(\lceil n/k \rceil/u, 1)$ for all $q \geq n$.

If $w > u$, (2) reduces to

$$\begin{aligned}
\frac{w+1}{u} - \left(\frac{w+1}{u} - 1\right)(1 - w^{-(w+1)})^\nu \\
\geq \lim_{k\to\infty} g(wk + \nu, k, u, k) \\
\geq \max\left(\frac{w}{u}, \frac{w+1}{u} - \frac{1}{u}(1 - w^{-(w+1)})^\nu\right).
\end{aligned}$$

Proof. (1) That g is increasing as a function of n follows by considering an extremal program P with n processes, i.e., $T_c(P, k, u)/T_d(P, q) = g(n, k, u, q)$. Consider also a program P' which is identical to P with the exception that a new process has been added. This new process is inactive during the entire execution of P'; i.e., there are $n+1$ processes in P'. Obviously, $T_c(P,k,u)/T_d(P,q) = T_c(P',k,u)/T_d(P',q)$. From the definition of g we know that $T_c(P',k,u)/T_d(P',q) \leq g(n+1,k,u,q)$. Consequently, $g(n,k,u,q) = T_c(P',k,u)/T_d(P',q) \leq g(n+1,k,u,q)$.

Consider $q < n$. That g is increasing as a function of q follows by considering a complete n,q-matrix P, i.e., $T_c(P,k,u)/T_d(P,q) = g(n,k,u,q)$. If we replace the first zero in each row with a one we get an $n,q+1$-matrix P'. Obviously, $T_d(P,q) = T_d(P',q+1)$ and $T_c(P,k,u) \leq T_c(P',k,u)$, i.e., $g(n,k,u,q) = T_c(P,k,u)/T_d(P,q) \leq T_c(P',k,u)/T_d(P',q+1) \leq g(n,k,u,q+1)$.

Since $\max(1,l/u)$ is decreasing as a function of u, so is $g(n,k,u,q)$.

$T_c(P,k,u)$ is obviously a decreasing function of k for any program P, and $T_d(P,q)$ is not affected by k. Consequently, $g(n,k,u,q) = \max_{\text{all } n\text{-programs } P} \frac{T_c(P,k,u)}{T_d(P,q)} \geq \max_{\text{all } n\text{-programs } P} \frac{T_c(P,k+1,u)}{T_d(P,q)} = g(n,k+1,u,q)$. Here n-programs denotes parallel programs with n processes.

(2) The idea of the proof of (2) is to exploit the fact that g is a weighted average of the numbers $\{\max(l/u,1) : l = 1,2,\ldots,\min(\lceil n/k\rceil,k)\}$. We calculate the weight W_k corresponding to the number $\max((w+1)/u,1)$ in the weighted average. We then let $k \to \infty$, in which case $W_k \to W$. Since k is large we may assume $\min(\lceil n/k\rceil,k) = \lceil n/k\rceil = w+1$. Then the estimate follows from $W\max((w+1)/u,1) + (1-W)1 \leq \lim_{k\to\infty} g(wk+\nu,k,u,k) \leq W\max((w+1)/u,1) + (1-W)\max(w/u,1)$. On page 763 in [4] we prove that $W = 1-(1-w^{-(w+1)})^{\nu}$.

The lower bound on $\lim_{k\to\infty} g(wk+\nu,k,u,k)$ for a certain w and ν is bounded from below by $\frac{w}{u}$. This follows from the fact that g is increasing in n.

If we let $\nu \to \infty$, the upper and lower bounds coincide, so

$$\lim_{\nu\to\infty}\left(\lim_{k\to\infty} g(wk+\nu,k,u,k)\right) = \frac{w+1}{u}.$$

Furthermore, for each k and ν, there is a ν_0 so that

$$g(wk+\nu,k,u,k) \geq g((w-1)k+\nu_1,k,u,k)$$

for all $\nu_1 > \nu_0$. Hence,

$$\lim_{k\to\infty} g(wk+\nu,k,u,k) \geq \frac{w}{u}.$$

(3) follows immediately from (2). We remark that $g(n,k,u,q)$ being unbounded means that for any real M there exist positive integers n,k,u,q so that $g(n,k,u,q) > M$. $\square$

Analogously to the function $g(n,k)$ described in [4], the existence of plateaus for the graph of $g(n,k,u,k)$ follows. From

$$\lim_{\nu\to\infty}\left(\lim_{k\to\infty} g(wk+\nu,k,u,k)\right) = \frac{w+1}{u},$$

and the fact that $g(n,k,u,k)$ is increasing in n, it follows that there exists an unbounded plateau at level $(w+1)/u$ for each integer w. For example, for each $\epsilon > 0$ there is a number N so that for all (n,k) in the domain $\{(n,k) : wk+N < n < (w+1)k\}$ we have $|\,g(n,k,u,k) - w/u\,| < \epsilon$. For a given ϵ, N appears to increase very rapidly with w.

The function $g(n,k) = g(n,k,1,k)$, which compares static with dynamic allocation, is neither increasing nor decreasing as a function of k (see [4]). A plot of the local extreme values of $g(n, k)$ as a function of k, for each fixed n, is shown in section 8. Detail structure otherwise not visible is revealed in this plot.

The optimal performance function $g(n, k, u, k)$ thus has a shape resembling a winding staircase with constant step height $1/u$ and an infinite number of steps, where each step is narrower, less sharp edged, and much farther from the origin than the previous step.

7. The process independent performance function.

It turns out that it is also possible to compute explicitly the optimal performance function $G(k, u, q)$.

THEOREM 7.1. *For any positive integers $k, u,$ and q,*

$$G(k,u,q) = \lim_{n\to\infty} g(n,k,u,q) = \sup_{n\in\mathbf{N}} g(n,k,u,q),$$

where $G(1, u, q) = \max(q/u, 1)$, and if $k > 1$,

$$G(k,u,q) = \frac{k!q!}{k^q u}\sum_{l=1}^{l=q}\frac{\max(l,u)}{l!}\sum_{I\in\mathcal{I}}\left(\Pi_{j=1}^{k-1} i_j!\,\Pi_{j=1}^{b(l,I)} a(l,I,j)!\right)^{-1}.$$

The set $\mathcal{I}$ is the set of all sequences $I = \{i_1, \dots, i_{k-1}\}$ of nonnegative integers which are decreasing; $i_j \geq i_{j+1}$ for all $j = 1, \dots, k-2$, bounded by $l; i_1 \leq l$, and have sum $q - l$; $\sum_{j=1}^{k-1} i_j = q - l$.

Further, the function $G(k, u, q)$ is decreasing as a function of k and as a function of u, and increasing as a function of q. $G(k) = G(k, 1, k)$ is an increasing function.

Proof. By Theorem 6.4 the function $g(n, k, u, q)$ is increasing as a function of n; hence the limit and the supremum in the theorem coincide. Furthermore it follows that $\lim_{n\to\infty} g(n, k, u, q) = \lim_{w\to\infty} g(wk, k, u, q)$. Note also that exactly the same decreasing sequences are generated in the function $\pi(k, w, k, l)$ for all $w \geq k$, if l and k are fixed. This fact makes it possible to explicitly compute the function $G(k, u, q)$. When letting $w \to \infty$, it is enough to study the behavior of the w-dependent part of each term, which is

$$\frac{\binom{w}{l}\binom{w}{i_1}\cdot\dots\cdot\binom{w}{i_c}}{\binom{wk}{q}}.$$

We will next prove that

$$\lim_{w\to\infty}\frac{\binom{w}{l}\binom{w}{i_1}\cdot\dots\cdot\binom{w}{i_c}}{\binom{wk}{q}} = \frac{q!k^{-q}}{l!i_1!\cdot\dots\cdot i_c!}.$$

Here c is the number of nonzero entries in the sequence I. In the following we will use the fact that $l + i_1 + \cdots + i_c = q$. From the Stirling formula $n! \approx (\frac{n}{e})^n\sqrt{2\pi n}$, we get for large n:

$$\binom{n}{k} \approx \frac{e^{-k}}{k!}\sqrt{\frac{n}{n-k}}\frac{n^n}{(n-k)^{n-k}}.$$

Thus the quotient approximately equals

$$\frac{e^{-l}}{l!}\sqrt{\frac{w}{w-l}}\frac{w^w}{(w-l)^{w-l}}\frac{e^{-i_1}}{i_1!}\sqrt{\frac{w}{w-i_1}}\frac{w^w}{(w-i_1)^{w-i_1}}\cdot\ldots$$

$$\cdot\frac{e^{-i_c}}{i_c!}\sqrt{\frac{w}{w-i_c}}\frac{w^w}{(w-i_c)^{w-i_c}}\frac{q!}{e^{-q}}\sqrt{\frac{kw-q}{kw}}\frac{(kw-q)^{kw-q}}{(kw)^{kw}}$$

$$=\frac{q!}{l!i_1!\cdot\ldots\cdot i_c!}$$

$$\cdot\sqrt{\frac{w}{w-l}\frac{w}{w-i_1}\cdot\ldots\cdot\frac{w}{w-i_c}\frac{kw-q}{kw}}$$

$$\cdot\Big(1-\frac{q}{kw}\Big)^{kw}\frac{w^{w-l}w^{w-i_1}\cdot\ldots\cdot w^{w-i_c}}{(w-l)^{w-l}(w-i_1)^{w-i_1}\cdot\ldots\cdot(w-i_c)^{w-i_c}}w^q(kw-q)^{-q}.$$

The first line after the last equality clearly is independent of w. The second line tends to 1. The first factor of the third line tends to e^{-q}. The second factor consists of $c+1$ factors, where each can be written as

$$\Big(1+\frac{i}{w-i}\Big)^w\cdot\Big(\frac{w}{w-i}\Big)^{-i}$$

for $i=l,i_1,\ldots,i_c$. The factors $(\frac{w}{w-i})^{-i}$ together with the third factor on the third line gives

$$\Big(\frac{kw-q}{w-l}\Big)^{-l}\Big(\frac{kw-q}{w-i_1}\Big)^{-i_1}\cdot\ldots\cdot\Big(\frac{kw-q}{w-i_c}\Big)^{-i_c}\to k^{-l}k^{-i_1}\cdot\ldots\cdot k^{-i_c}=k^{-q}.$$

Now,

$$\Big(1+\frac{i}{w-i}\Big)^w\to e^i\text{ as }w\to\infty,$$

so the remaining factors contributes with

$$e^le^{i_1}\cdot\ldots\cdot e^{i_c}=e^q$$

as $w\to\infty$. This leaves

$$\frac{\binom{w}{l}\binom{w}{i_1}\cdot\ldots\cdot\binom{w}{i_c}}{\binom{wk}{q}}\to\frac{q!}{l!i_1!\cdot\ldots\cdot i_c!}\cdot e^{-q}\cdot e^q\cdot k^{-q}=\frac{q!}{l!i_1!\cdot\ldots\cdot i_c!}k^{-q}.$$

Since $g(n,1,u,q)=\max(\min(n,q)/u,1)$, it immediately follows that $G(1,u,q)=\max(q/u,1)$. Note that this case represents the worst case ratio of dynamic versus dynamic allocation, with u and q processors, respectively. This completes the proof of the formula for $G(k,u,q)$.

The increasing and decreasing properties for $G(k,u,q)$ follow from (1) in Theorem 6.4. In Theorem 2 part (2) on page 762 in paper [4] we show that $g(wk,k,1,k)$ is increasing as a function of k for any positive integer w. From this we immediately get that $G(k) = G(k,1,k)$ is increasing. The theorem is proved. $\square$

In the case of static allocation and $k = q$, the case considered in [4], we get $G(k) = G(k,1,k)$, where

$$G(k) = \frac{(k!)^2}{k^k} \sum_{l=1}^{l=k} \frac{1}{(l-1)!} \sum_I \left(\Pi_{j=1}^{k-1} i_j! \, \Pi_{j=1}^{b(\{l\}+I)} a(\{l\}+I,j)! \right)^{-1}.$$

The last sum is taken over the same sequences I as in the previous formula, with $q = k$.

Note that the number of terms in the sum increases rapidly with k. However, being unit fractions with products of factorials as denominators, all terms are very small if k is large. From the Stirling formula it indeed follows that $(k!)^2 \gg k^k \gg k!$ for large k.

We give the first values of $G(k)$ as maximally reduced rational numbers, prime factorizations, and decimal expansions:

$G(1) = 1$,
$G(2) = \frac{3}{2} = 1.5$,
$G(3) = \frac{17}{9} = \frac{17}{3^2} = 1.888\ldots$,
$G(4) = \frac{17}{8} = \frac{17}{2^3} = 2.125$,
$G(5) = \frac{1429}{625} = \frac{1429}{5^4} = 2.2864$,
$G(6) = \frac{3121}{1296} = \frac{3121}{2^4 3^4} = 2.40818\ldots$,
$G(7) = \frac{295189}{117649} = \frac{211 \cdot 1399}{7^6} = 2.50907\ldots$,
$G(8) = \frac{680849}{262144} = \frac{13 \cdot 83 \cdot 631}{2^{18}} = 2.597232\ldots$,
$G(9) = \frac{38404547}{14348907} = \frac{43 \cdot 107 \cdot 491 \cdot 1717}{3^{15}} = 2.67648\ldots$,
$G(10) = \frac{274868911}{100000000} = \frac{3929 \cdot 69959}{2^8 5^8} = 2.74868911.$

We conclude with some notes on the computability of the functions.

The variable u does not affect the number of floating point operations necessary for computing the value of the function $g(n,k,u,q)$. This number increases slightly with q and n, if n is a multiple of k, because of the larger values in the binomials and factorials to be computed. However, the number of operations increases rapidly with the variable k since the number of decreasing sequences increase rapidly. This is also true for the variable n if it is not a multiple of k.

The approximation $h(n,k,u,q) = g(k\lceil n/k \rceil, k, u, q)$ is much faster to compute, since only a fraction of the values are needed. Furthermore, by Theorem 6.1 it is evident that the values which are calculated involve a simple sum in the function π only; hence they are the computationally lightest. The function h is close to optimal upper bound since $h(n,k,u,q) \geq g(n,k,u,q)$ for all n,k,u, and q, with equality whenever n/k is an integer.

8. Graphics.

The graphics and tables in this section refer to and illustrate functions and concepts of the report. One aim is to describe the performance functions quantitatively and geometrically. Another is to give examples of how the functions can be used to provide optimal trade-off estimates, aiding in the choices between multiprocessor design alternatives.

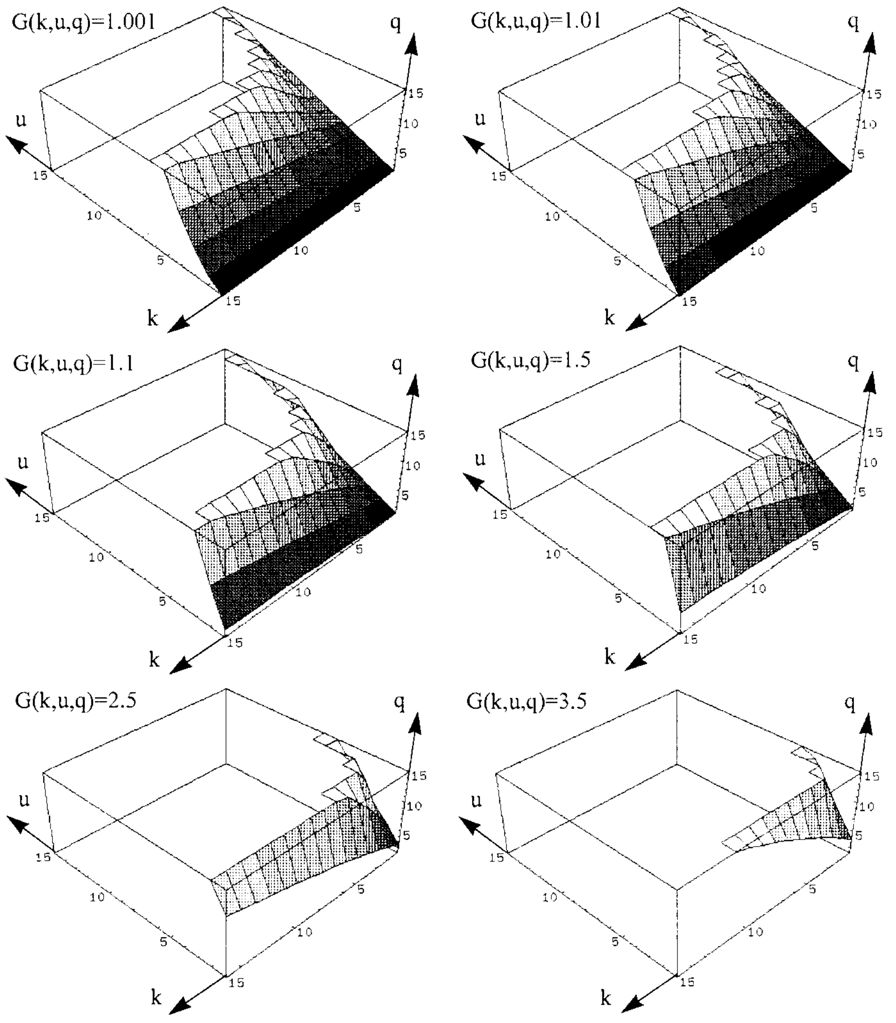


FIG. 8.1. *Level surfaces $G(k, u, q) = C$.*

In Figure 8.1 we plot a version of $G(k, u, q)$ where the discrete variable q is turned into a continuous variable Q by linear interpolation between neighboring values. Thus, given k, u, and C, all values Q such that $(\lceil Q\rceil - Q)G(k, u, \lfloor Q\rfloor) + (Q - \lfloor Q\rfloor)G(k, u, \lceil Q\rceil) = C$ are plotted, if any exist. Since $G(k, u, q) \geq 1$ is a sharp bound of G, every $C \geq 1$ gives a plot. The grayscale in the plots emphasizes the value of q: black is $q = 1$, white is $q = 15$. $G(k, u, q) = 1$ if $u \geq q$, so there are no level surfaces here for $C > 1$. $G(k, u, q) > 1$ if $q > u$; however, G clearly increases very slowly with q when $q - u$ is small.

Most of the values of $g(n, k, u, q)$ are 1. Figure 8.2 indicates in which directions $g(n, k, u, q)$ is not 1 and indicates the initial growth here. $g(n, k, u, q) = 1$ if $u \geq \lceil n/k\rceil$, $u \geq q$, or $k \geq n$. Further, $g(n, k, u, q) = g(n, k, u, n)$ if $q > n$. In all these cases there are more processors than processes in some sense. $g(n, 1, 1, n) = n$ is the direction of fastest growth. For example, here we have the only value where $g(n, k, u, q) > 1$ if

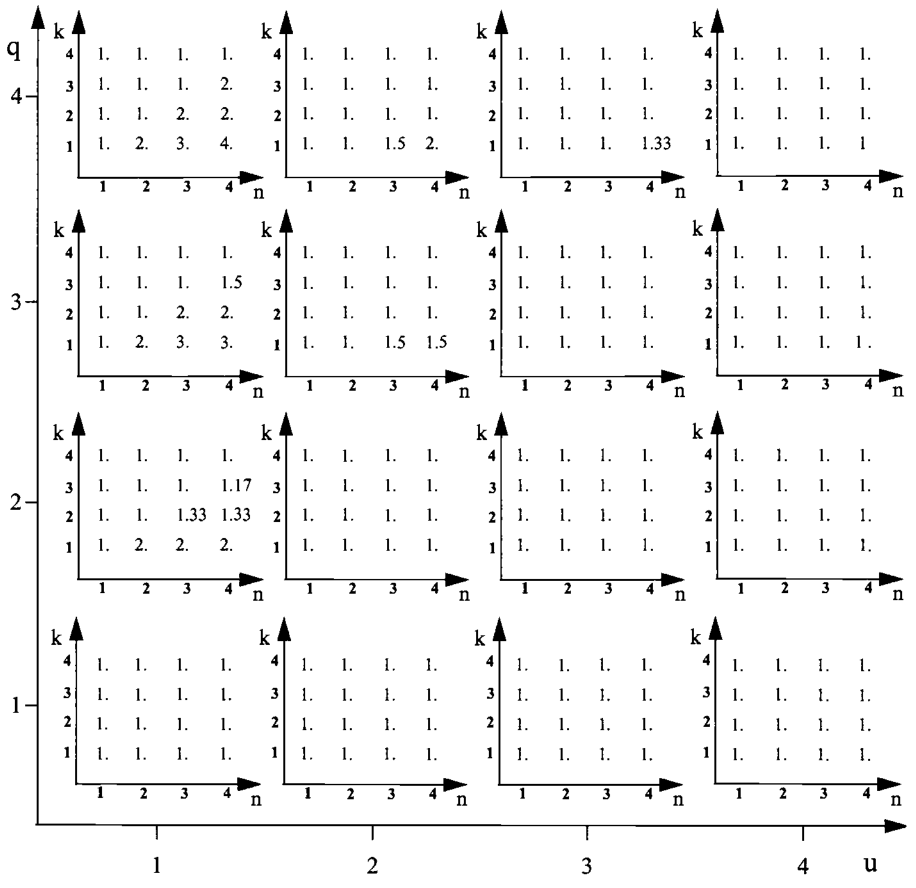


FIG. 8.2. *The* 256 *first values of* $g(n, k, u, q)$.

all arguments are one or two. In multicomputing terms this corresponds to the worst case ratio of one processor compared to n processors, running a multiprogram with n subprocesses.

The plot in Figure 8.3 shows the worst case execution time for three degrees of cluster allocation, compared to dynamic allocation. The number of processors in corresponding points is equal. Note that when there are $u > 1$ processors in each cluster, the plateaus for the case $u = 1$ split into several, where plateau number i have asymptotic height $g = 1 + (i - 1)/u$.

The function $G(k, u, ku)$ expresses the worst case ratio of cluster allocation with k clusters and u processors in each cluster, compared to dynamic allocation with the same processor quantity: ku processors. See Figure 8.4.

Note the following: $G(k, 1, k)$ is static versus dynamic allocation (k-axis). $G(1, u, u)$ $= 1$; this is dynamic versus dynamic allocation (u-axis). The function G exists only for integer arguments; in the plot, linear interpolation is done between these points. The grayscale in the figure indicates the value linearly, where black is minimum and white is maximum.

A measure of the trade-off between processor organization and processor power is

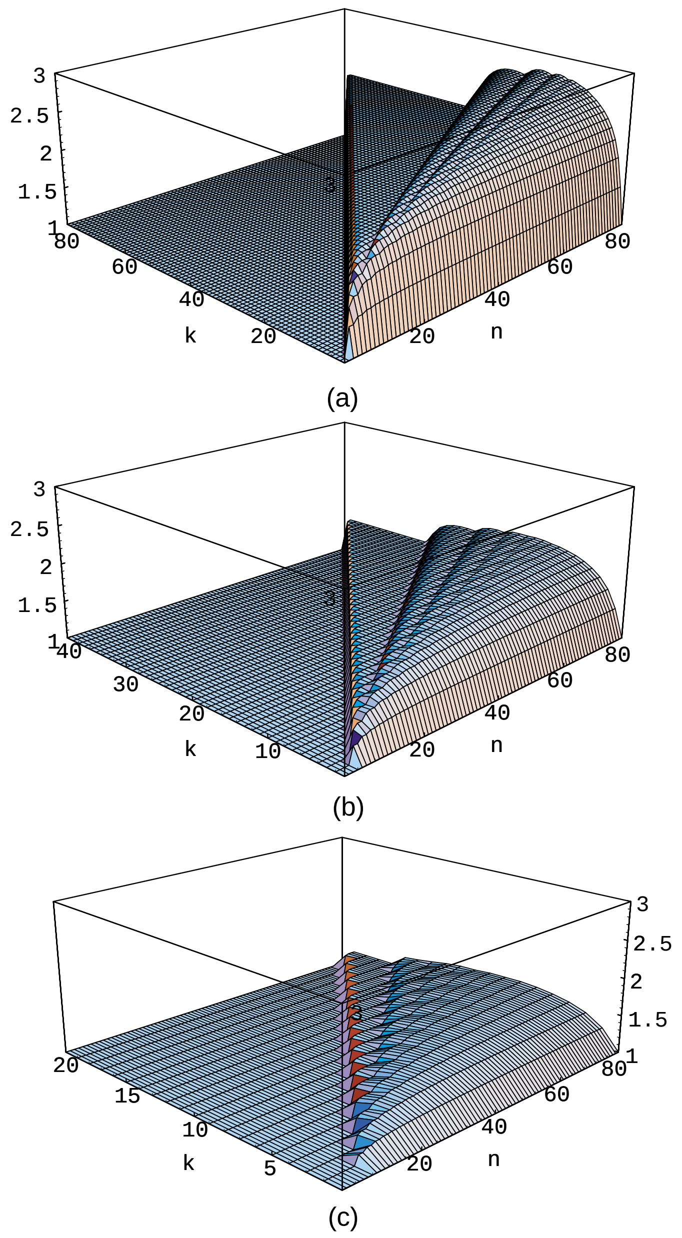


FIG. 8.3. *Increasing degree of cluster allocation.* (a) $g(n, k, 1, k)$, $n \leq 80$, $k \leq 80$; (b) $g(n, k, 2, 2k)$, $n \leq 80$, $k \leq 40$; (c) $g(n, k, 4, 4k)$, $n \leq 80$, $k \leq 20$.

provided by the graphs in Figure 8.5. The gain in execution time $G(k, u, q)$ for each possible cluster allocations using a fixed number of processors (ku) is compared to dynamic allocation with various numbers of processors (q). For example, if we assume $ku = 6$, constant worst case efficiency implies that the four possible cluster allocation strategies correspond to dynamic allocation with 7, 11, 13, and 16 processors, in increasing degree of cluster organization of the processors. Optimal bounds comparing

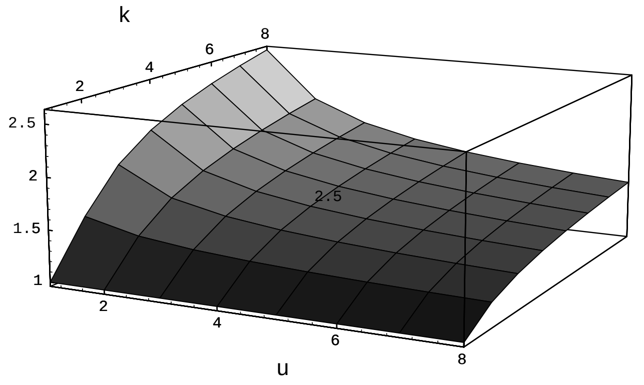


FIG. 8.4. *Cluster versus dynamic allocations on the same multiprocessor.*

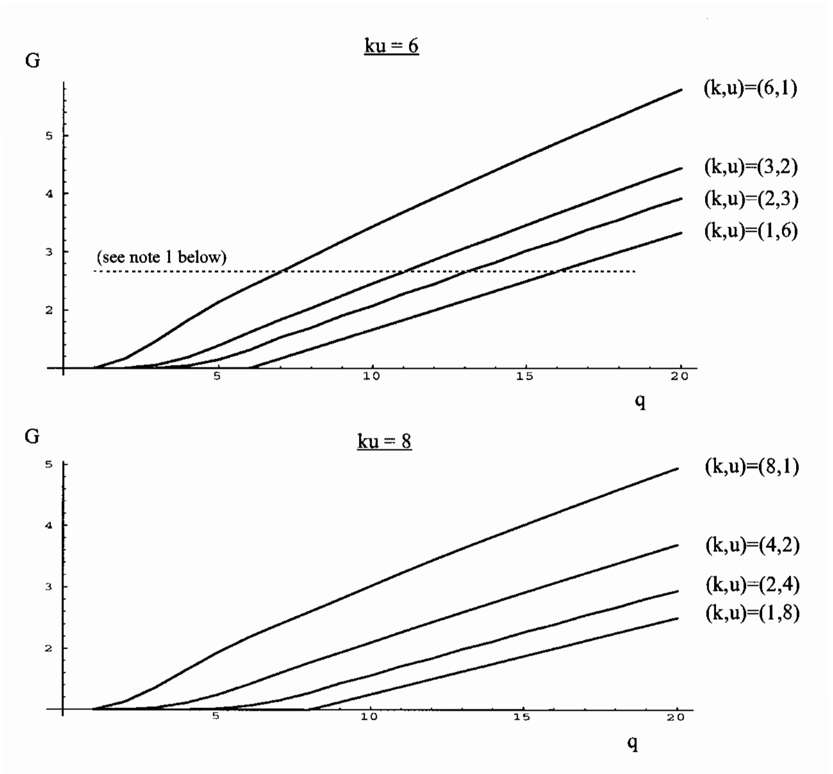


FIG. 8.5. *Degree of clustering versus processor quantity.*

any two cluster allocations, generalizing the main results of the article, cannot be concluded. The worst case estimate of dynamic (u) versus dynamic (q) allocations, $G(1, u, q) = \max(q/u, 1)$, appears in the figure.

Functions giving upper and lower bounds for the diagonal limit function

$$D(w, \nu) = \lim_{k\to\infty} g(wk + \nu, k, 1, k)$$

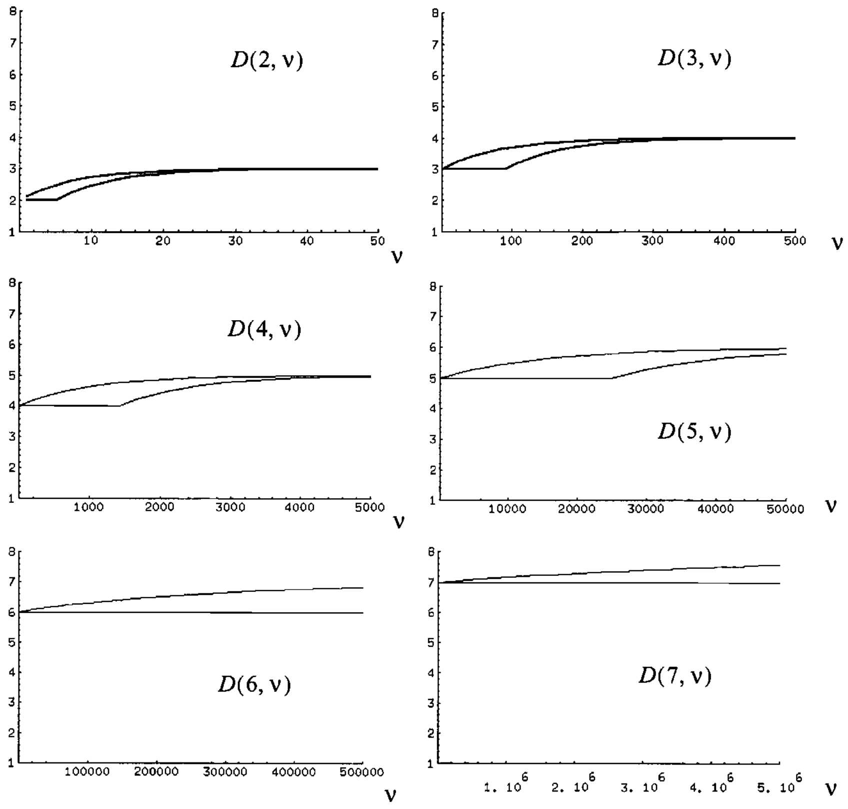


FIG. 8.6. *Bounds on diagonal limit functions.*

are plotted in Figure 8.6 given by the estimate of Theorem 6.4(2) in the case $u = \alpha = 1$:

$$\max(w, w + 1 - w(1 - w^{-w-1})^{\nu}) \leq D(w, \nu) \leq w + 1 - (1 - w^{-w-1})^{\nu}.$$

Observe the increasing scale of the ν-axis. The plots illustrate the fact that the distance from the wth plateau to the next increases very rapidly with w.

REFERENCES

[1] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman, San Francisco, CA, 1979.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA, 1994.

[3] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM. J. Appl. Math., 17 (1969), pp. 416–429.

[4] H. Lennerstad and L. Lundberg, *An optimal execution time estimate of static versus dynamic allocation in multiprocessor systems*, SIAM J. Comput., 24 (1995), pp. 751–764.

[5] H. Lennerstad and L. Lundberg, *Optimal scheduling results for parallel computing*, in Applications on Advanced Architecture Computers, Greg Astfalk, ed., SIAM, Philadelphia, 1996, pp. 155–163.

[6] H. Lennerstad and L. Lundberg, *Combinatorics for multiprocessor scheduling optimization and other contexts in computer architecture*, in Proceedings of the Conference of Com-

binatorics and Computer Science, Lecture Notes in Comput. Sci. 1120, Springer-Verlag, New York, 1996.

[7] H. Lennerstad and L. Lundberg, *Combinatorial formulas for optimal cache memory efficiency*, SIAM News, July/August 1996, pp. 341–347.

[8] L. Lundberg and H. Lennerstad, *An Optimal Performance Bound on the Gain of Using Dynamic versus Static Allocation in Multiprocessors with Clusters*, Technical Report, University of Karlskrona/Ronneby, Sweden, 1993.

[9] L. Lundberg and H. Lennerstad, *An optimal lower bound on the maximal speedup in multiprocessors with clusters*, in Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing, Australia, 1995, pp. 640–649.

[10] L. Lundberg and H. Lennerstad, *An optimal bound on the gain of using one large processor cluster instead of a number of small clusters*, in Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, Orlando, FL, 1995, pp. 576–582.

[11] H. Lennerstad and L. Lundberg, *Optimal worst case formulas comparing cache memory associativity*, SIAM J. Comput., to appear.

[12] L. Lundberg and H. Lennerstad, *Combinatorics for scheduling optimization*, in Combinatorics and Computer Science, Lecture Notes in Comput. Sci., Springer-Verlag, New York, 1996.

[13] L. Lundberg and H. Lennerstad, *Bounding the maximum gain of changing the number of memory modules in multiprocessor computers*, Nordic J. Comput., 4 (1997), pp. 233–258.

[14] L. Lundberg and H. Lennerstad, *Optimal bound on the gain of permitting dynamical allocation of communication channels in distributed processing*, Acta Inform., 36 (1999), pp. 425–446.

[15] L. Lundberg and H. Lennerstad, *Using recorded values for bounding the minimum completion time in multiprocessors*, IEEE Trans. Parallel and Distributed Systems., 9 (1998), pp. 346–358.

[16] MPI Forum, *MPI: A message-passing interface standard*, Internat. J. Supercomput. Appl., 8 (1994), pp. 165–416.

[17] R. McNaughton, *Scheduling with deadlines and loss functions*, Management Sci., 6 (1959), pp. 1–12.

SIAM J. Comput. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 1839–1857

THE CREW PRAM COMPLEXITY OF MODULAR INVERSION*

JOACHIM VON ZUR GATHEN† and IGOR E. SHPARLINSKI‡

Abstract. One of the long-standing open questions in the theory of parallel computation is the parallel complexity of the integer gcd and related problems, such as modular inversion. We present a lower bound $\Omega(\log n)$ for the parallel time on a concurrent-read exclusive-write parallel random access machine (CREW PRAM) computing the inverse modulo certain n-bit integers, including all such primes. For infinitely many moduli, our lower bound matches asymptotically the known upper bound. We obtain a similar lower bound for computing a specified bit in a large power of an integer. Our main tools are certain estimates for exponential sums in finite fields.

Key words. modular inversion, parallel computation, CREW PRAM complexity, exponential sums

AMS subject classifications. 68Q25, 11T23, 11Y16

PII. S0097539797328070

1. Introduction.

In this paper we address the problem of parallel computation of the inverse of integers modulo an integer M. That is, given positive integers $M \geq 3$ and $x < M$, with $\gcd(x, M) = 1$, we want to compute its modular inverse $\mathrm{inv}_M(x) \in \mathbb{N}$ defined by the conditions

$$x \cdot \mathrm{inv}_M(x) \equiv 1 \bmod M, \qquad 1 \leq \mathrm{inv}_M(x) < M. \tag{1.1}$$

Since $\mathrm{inv}_M(x) \equiv x^{\varphi(M)-1} \bmod M$, where φ is the Euler function, inversion can be considered as a special case of the more general question of modular exponentiation. Both these problems can also be considered over finite fields and other algebraic domains.

For inversion, exponentiation, and gcd, several parallel algorithms are in the literature [1, 2, 3, 9, 10, 11, 12, 13, 14, 15, 17, 19, 20, 22, 27, 29]. The question of obtaining a general parallel algorithm running in polylogarithmic time $(\log n)^{O(1)}$ for n-bit integers M is wide open [11, 12].

Some lower bounds on the depth of arithmetic circuits are known [11, 15]. On the other hand, some examples indicate that for this kind of problem the Boolean model of computation may be more powerful than the arithmetic model; see discussions of these phenomena in [9, 11, 15].

In this paper we show that the method of [5, 25] can be adapted to derive nontrivial lower bounds on Boolean concurrent-read exclusive-write parallel random access machines (CREW PRAMs). It is based on estimates of exponential sums.

Our bounds are derived from lower bounds for the *sensitivity* $\sigma(f)$ (or *critical complexity*) of a Boolean function $f(X_1, \ldots, X_n)$ with binary inputs $X_1, \ldots, X_n$. It is defined as the largest integer $m \leq n$ such that there is a binary vector $x = (x_1, \ldots, x_n)$ for which $f(x) \neq f(x^{(i)})$ for m values of $i \leq n$, where $x^{(i)}$ is the vector obtained from x by flipping its ith coordinate. In other words, $\sigma(f)$ is the maximum, over all input

*Received by the editors October 1, 1997; accepted for publication (in revised form) March 9, 1999; published electronically April 4, 2000.
http://www.siam.org/journals/sicomp/29-6/32807.html

†Fachbereich Mathematik-Informatik, Universität Paderborn, 33095 Paderborn, Germany (gathen@uni-paderborn.de).

‡School of MPCE, Macquarie University, NSW 2109, Sydney, Australia (igor@mpce.mq.edu.au).

vectors x, of the number of points y on the unit Hamming sphere around x with $f(y) \neq f(x)$; see, e.g., [30].

Since [4], the sensitivity has been used as an effective tool for obtaining lower bounds of the CREW PRAM complexity, i.e., the *time complexity* on a parallel random access machine with an unlimited number of all-powerful processors, where each machine can read from and write to one memory cell at each step, but where no write conflicts are allowed: each memory cell may be written into by only one processor, at each time step.

By [21], $0.5 \log_2(\sigma(f)/3)$ is a lower bound on the parallel time for computing f on such machines; see also [6, 7, 8, 30]. This yields immediately the lower bound $\Omega(\log n)$ for the OR and the AND of n input bits. It should be contrasted with the common concurrent-read concurrent-write (CRCW) PRAM, where write conflicts are allowed, provided every processor writes the same result, and where all Boolean functions can be computed in constant time (with a large number of processors).

The contents of the paper are as follows. In section 2, we prove some auxiliary results on exponential sums. We apply these in section 3 to obtain a lower bound on the sensitivity of the least bit of the inverse modulo a prime. In section 4, we use the same approach to obtain a lower bound on the sensitivity of the least bit of the inverse modulo an odd squarefree integer M. The bound is somewhat weaker, and the proof becomes more involved due to zero-divisors in the residue ring modulo M, but for some such moduli we are able to match the known upper and the new lower bounds. Namely, we obtain the lower bound $\Omega(\log n)$ on the CREW PRAM complexity of inversion modulo, an n-bit odd squarefree M with not “too many” prime divisors, and we exhibit infinite sequences of M for which this bound matches the upper bound $O(\log n)$ from [11] on the depth of P-uniform Boolean circuits for inversion modulo a “smooth” M with only “small” prime divisors; see (4.6) and (4.7). For example, the bounds coincide for moduli $M = p_1 \cdots p_s$, where $p_1, \ldots, p_s$ are any $\lceil s/\log s \rceil$ prime numbers between s^3 and $2s^3$.

We apply our method in section 5 to the following problem posed by Allan Borodin (see Open Question 7.2 of [11]): given n-bit positive integers m, x, e, compute the mth bit of x^e.

Generally speaking, a parallel lower bound $\Omega(\log n)$ for a problem with n inputs is not a big surprise. Our interest in these bounds comes from their following features:

- some of these questions have been around for over a decade;
- no similar lower bounds are known for the gcd;
- on the common CRCW PRAM, the problems can be solved in constant time;
- for some types of inputs, our bounds are asymptotically optimal;
- the powerful tools we use from the theory of finite fields might prove helpful for other problems in this area.

2. Exponential sums.

The main tool for our bounds are estimates of exponential sums. For positive integers M and z, we write $\mathbf{e}_M(z) = \exp(2\pi i z/M) \in \mathbb{C}$. Thus $\mathbf{e}_M(z_1 + z_2) = \mathbf{e}_M(z_1) + \mathbf{e}_M(z_2)$ for any z_1, z_2.

The following identity follows from the formula for a geometric sum.

LEMMA 2.1. *For any integer a,*

$$\sum_{0 \le a < M} \mathbf{e}_M(au) = \begin{cases} 0, & \textit{if } u \not\equiv 0 \bmod M, \\ M, & \textit{if } u \equiv 0 \bmod M. \end{cases}$$

LEMMA 2.2. *For positive integers M and H, we have*

$$\sum_{0\le a<M}\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right| = H^2+(r+1)(M-r-1),$$

where $r\equiv H-1 \bmod M$ with $0\le r<M$ is the remainder of $H-1$ modulo M.

Proof. We note that

$$\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right) = \left|\sum_{0\le x<H}\mathbf{e}_M(ax)\right|^2 > 0.$$

Thus

$$\begin{aligned}\sum_{0\le a<M}\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right| &= \sum_{0\le a<M}\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\\ &= \sum_{0\le x,y<H}\sum_{0\le a<M}\mathbf{e}_M\left(a(y-x)\right).\end{aligned}$$

From Lemma 2.1 we see that the last sum is equal to MW, where W is the number of (x,y) with $x\equiv y \bmod M$ and $0\le x,y<H$. It is easy to see that

$$W=\sum_{0\le i<M}\left(\left\lfloor\frac{H-1-i}{M}\right\rfloor+1\right)^2.$$

Let $s=r+1$ and $q=\lfloor (H-1)/M\rfloor$, thus $q=(H-s)/M$. Then,

$$\begin{aligned}W&=(r+1)(q+1)^2+(M-r-1)q^2=Mq^2+s(2q+1)\\ &=(H-s)q+2sq+s=(H+s)q+s\\ &=\frac{H^2-s^2}{M}+s=\frac{1}{M}(H^2+sM-s^2),\end{aligned}$$

and the result follows. □

Taking into account that $(r+1)(M-r-1)\le M^2/4$, we derive from Lemma 2.2 that the bound

$$\sum_{0\le a<M}\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right|\le H^2+M^2/4 \tag{2.1}$$

holds for any H and M.

Also, it is easy to see that for $H\le M$, we have $r=H-1$, and the identity of Lemma 2.2 takes the form

$$\sum_{0\le a<M}\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right| = MH, \qquad 0\le H\le M. \tag{2.2}$$

Finally, we have

$$\sum_{1\le a<M}\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right| = (r+1)(M-r-1)\le M^2/4. \tag{2.3}$$

Indeed, this sum is smaller by the term corresponding to $a = 0$, which equals H^2.

In what follows, we consider several sums over values of rational functions in residue rings, which may not be defined for all values. We use the symbol $\sum^*$ to express that the summation is extended over those arguments for which the rational function is well defined, so that its denominator is relatively prime to the modulus. We give an explicit definition only in the example of the following statement, which is known as the *Weil bound*; see [18, 24, 31].

LEMMA 2.3. *Let $f, g \in \mathbb{Z}[X]$ be two polynomials of degrees n, m, respectively, and p a prime number such that the rational function f/g is defined and not constant modulo p. Then*

$$\left|\sum_{0\le x<p}{}^{*}\,\mathbf{e}_p\left(f(x)/g(x)\right)\right| = \left|\sum_{\substack{0\le x<p\\ \gcd(g(x),p)=1}} \mathbf{e}_p\left(f(x)/g(x)\right)\right| \le (n+m-1)p^{1/2}.$$

Let $\omega(k)$ denote the number of distinct prime divisors of an integer k. The following statement is a combination of the Chinese remainder theorem and the Weil bound.

LEMMA 2.4. *Let $M \in \mathbb{N}$ be squarefree with $M \ge 2$, d a divisor of M, and $f, g \in \mathbb{Z}[X]$ of degrees n, m, respectively, such that the rational function f/g is defined and not constant modulo each prime divisor $p > \max\{n, m\}$ of M. Then*

$$\left|\sum_{0\le x<M}{}^{*}\,\mathbf{e}_M\left(d\,f(x)/g(x)\right)\right| \le (n+m-1)^{\omega(M)}M^{1/2}d^{1/2}.$$

Proof. In the following, p stands for a prime divisor of M. We define $M_p \in \mathbb{N}$ by the conditions

$$M_p \equiv 0 \bmod M/p, \qquad M_p \equiv 1 \bmod p, \qquad 1 \le M_p \le M.$$

Then, one easily verifies the identity

$$\sum_{0\le x<M}{}^{*}\,\mathbf{e}_M\left(d\,f(x)/g(x)\right) = \prod_{p|M}\sum_{0\le x<p}{}^{*}\,\mathbf{e}_p\left(d\,f(M_p x)/g(M_p x)\right).$$

We use the estimate of Lemma 2.3 for those p for which $p\nmid d$ and $p > \max\{n, m\}$, and estimate trivially by p the sum for each other p. Then

$$\begin{aligned}\left|\sum_{0\le x<M}{}^{*}\,\mathbf{e}_M\left(d\,f(x)/g(x)\right)\right| &\le \prod_{p\nmid d}(n+m-1)p^{1/2}\prod_{p|d}p \\ &= (n+m-1)^{\omega(M/d)}(Md)^{1/2}.\end{aligned}$$

Since $\omega(M/d) \le \omega(M)$, we obtain the desired estimate. □

LEMMA 2.5. *Let $M \ge 2$ be a squarefree integer, $f, g \in \mathbb{Z}[X]$ of degrees n, m, respectively, such that f/g is defined and neither constant nor a linear function modulo each prime divisor p of M. Then for any $N, H, d \in \mathbb{N}$ with $H \le M$ and $d|M$, we have*

$$\left|\sum_{0\le x,y<H}{}^{*}\,\mathbf{e}_M\left(d\,\frac{f(N+x-y)}{g(N+x-y)}\right)\right| \le (n+m-1)^{\omega(M)}HM^{1/2}d^{1/2}.$$

Proof. From Lemma 2.1 we obtain

$$\begin{aligned}
&\left|\sum_{0\le x,y<H}{}^{*}\,\mathbf{e}_M\left(d\,\frac{f(N+x-y)}{g(N+x-y)}\right)\right| \\
&= \left|\sum_{0\le u<M}{}^{*}\,\mathbf{e}_M\left(d\,f(u)/g(u)\right)\sum_{0\le x,y<H}\frac{1}{M}\sum_{0\le a<M}\mathbf{e}_M\left(a(u-N-x+y)\right)\right| \\
&= \frac{1}{M}\left|\sum_{0\le u<M}{}^{*}\,\mathbf{e}_M\left(d\,f(u)/g(u)\right)\sum_{0\le a<M}\sum_{0\le x,y<H}\mathbf{e}_M\left(a(u-N-x+y)\right)\right| \\
&= \frac{1}{M}\left|\sum_{0\le a<M}\mathbf{e}_M(-aN)\sum_{0\le u<M}{}^{*}\,\mathbf{e}_M\left(d\frac{f(u)}{g(u)}+au\right)\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right| \\
&\le \frac{1}{M}\sum_{0\le a<M}\left|\sum_{0\le u<M}{}^{*}\,\mathbf{e}_M\left(d\frac{f(u)}{g(u)}+au\right)\right|\cdot\left|\sum_{0\le x,y<H}\mathbf{e}_M\left(a(y-x)\right)\right|.
\end{aligned}$$

From Lemma 2.4 we see that for each $a < M$ the sum over u can be estimated as

$$\left|\sum_{0\le u<M}{}^{*}\,\mathbf{e}_M\left(d\frac{f(u)}{g(u)}+au\right)\right| \le \left(\max\{n+m-1,2m\}\right)^{\omega(M)}M^{1/2}\delta^{1/2},$$

where $\delta = \gcd(d,a) \le d$. Applying the estimate (2.2), we obtain the result. ∎

The following result is the particular case $p = 2$ of Theorem 1 of [28].

LEMMA 2.6. *There exists a constant c such that for all polynomials $f = a_tX^t + \cdots + a_1X + a_0 \in \mathbb{Z}[X]$ with $\gcd(a_t,\ldots,a_1,2) = 1$ and all integers $m \ge 1$ we have*

$$\left|\sum_{0\le x<2^m}\mathbf{e}_{2^m}\left(f(x)\right)\right| \le c\cdot 2^{m(1-1/t)}.$$

For $a_0,\ldots,a_{k-1} \in \mathbb{Z}$, not all zero, we define $\mu(a_0,\ldots,a_{k-1})$ to be the largest exponent e for which 2^e divides $a_0,\ldots,a_{k-1}$.

LEMMA 2.7. *Let $a_0,\ldots,a_{k-1} \in \mathbb{Z}$ not be all zero, and*

$$b_j = \sum_{0\le i<k} a_i 2^{ij}$$

for $0 \le j < k$. Then $\mu(b_0,\ldots,b_{k-1}) \le \mu(a_0,\ldots,a_{k-1}) + (k-1)(k-2)/2$.

Proof. We extend μ to $\mathbb{Q}$ by $\mu(a/b) = \mu(a) - \mu(b)$ and to nonzero matrices in $\mathbb{Q}^{k\times k}$ by taking the minimum value at all nonzero entries. Then $\mu(U\cdot v) \ge \mu(U) + \mu(v)$ for a matrix U and a vector v such that $Uv \ne 0$.

Let $C_k = (2^{ij})_{0\le i,j<k}$. The determinant of this Vandermonde matrix has value

$$\mu(\det C_k) = \mu\left(\prod_{0\le i<j<k}(2^j-2^i)\right) = \sum_{0\le i<j<k} i = \frac{1}{6}k(k-1)(k-2).$$

We consider an entry of the adjoint $\mathrm{ad}C_k$ of C_k. Each of the summands contributing to the determinant expansion of that entry is divisible by

$$2^{(k-3)+2(k-4)+\cdots+(k-3)},$$

so that

$$\mu(\mathrm{ad}C_k) \geq \sum_{1\leq i<k-2} i\cdot(k-2-i) = \frac{1}{6}(k-1)(k-2)(k-3).$$

(In fact, we have equality, since $\det C_{k-1}$ has the right-hand side as its μ-value and is one entry of $\mathrm{ad}C_k$.) Therefore

$$\begin{aligned}\mu(C_k^{-1}) &\geq \mu(\mathrm{ad}C_k) - \mu(\det C_k)\\ &\geq \frac{1}{6}(k-1)(k-2)(k-3) - \frac{1}{6}k(k-1)(k-2)\\ &= -\frac{1}{2}(k-1)(k-2).\end{aligned}$$

Now from the inequality $\mu(a) = \mu(C_k^{-1}b) \geq \mu(C_k^{-1}) + \mu(b)$ the result follows. ◻

We also need an estimate on the number of terms in the sum of Lemma 2.5. For a polynomial $g \in \mathbb{Z}[X]$ and $M, H \in \mathbb{Z}$, we denote by $T_g(M,H)$ the number of $x \in \mathbb{Z}$ for which $0 \leq x < H$ and $\gcd(g(x), M) = 1$. The following result is, probably, not new and can be improved via more sophisticated sieve methods.

LEMMA 2.8. *Let $M > 1$ be squarefree and $g \in \mathbb{Z}[x]$ of degree m such that* $\gcd(g(x), M) = 1$ *for some $x \in \mathbb{Z}$. Then for all integers $H \leq M$, we have*

$$T_g(M,H) \geq H\prod_{p|M}\left(1 - \frac{\min\{m, p-1\}}{p}\right) - (m+1)^{\omega(M)}.$$

Proof. We denote by $\rho(M,H)$ the number of $x \in \{0,\ldots,H-1\}$ such that

$$g(x) \equiv 0 \bmod M,$$

and set $\rho(M) = \rho(M,M)$. Since M is squarefree, the inclusion–exclusion principle yields

$$T_g(M,H) = H + \sum_{1\leq k\leq\omega(M)} (-1)^k \sum_{\substack{d|M\\ \omega(d)=k}} \rho(d,H).$$

For any divisor d of M we have

$$\left|\rho(d,H) - \rho(d)\frac{H}{d}\right| \leq \rho(d) = \prod_{p|d}\rho(p).$$

Therefore,

$$\begin{aligned}T_g(M,H) &\geq H + H\sum_{1\leq k\leq\omega(M)} (-1)^k \sum_{\substack{d|M\\ \omega(d)=k}} \frac{\rho(d)}{d} - \sum_{d|M}\rho(d)\\ &= H\prod_{p|M}\left(1 - \frac{\rho(p)}{p}\right) - \prod_{p|M}\left(1+\rho(p)\right).\end{aligned}$$

By assumption, g takes a nonzero value modulo every prime divisor p of M. Thus $\rho(p) \le \min\{m, p-1\}$, and the claim follows. $\square$

Throughout this paper, $\log z$ means the logarithm of z in base 2, $\ln z$ means the natural logarithm, and

$$\operatorname{Ln} z = \begin{cases} \ln z, & \text{if } z > 1, \\ 1, & \text{if } z \le 1. \end{cases}$$

LEMMA 2.9. *For positive integers m and M, with $M > 1$ squarefree, we have*

$$\prod_{p \mid M} \left(1 - \frac{\min\{m, p-1\}}{p}\right) \ge \exp\left(-2m \operatorname{Lnln} \omega(M) - 7m\right).$$

Proof. We split the logarithm of the product as follows:

$$\ln \prod_{p \mid M} \left(1 - \frac{\min\{m, p-1\}}{p}\right) \ge \sum_{\substack{p \mid M \\ p \le 2m}} \ln\left(\frac{1}{p}\right) + \sum_{\substack{p \mid M \\ p > 2m}} \ln\left(1 - \frac{m}{p}\right); \tag{2.4}$$

and we prove a lower bound on each summand. For the first one, we use that

$$\sum_{p \le x} \ln p \le x\left(1 + \frac{1}{2\ln x}\right) \text{ for } x > 1$$

by [24, (3.15)]. Thus, for $m > 1$,

$$\sum_{\substack{p \mid M \\ p \le 2m}} \ln p \le \sum_{p \le 2m} \ln p \le 2m\left(1 + \frac{1}{2\ln 2m}\right) \le 3m. \tag{2.5}$$

It is easy to verify that for $m = 1$ the sum on the left-hand side does not exceed $3m$ as well.

For the second summand, we use that $(1 + 2\delta)(1 - \delta) = 1 + \delta(1 - 2\delta) \ge 1$ for $0 \le \delta < 1/2$, so that $\exp(2\delta) > 1 + 2\delta \ge (1 - \delta)^{-1}$ and $\ln(1 - \delta) > -2\delta$. This implies that

$$\sum_{\substack{p \mid M \\ p > 2m}} \ln\left(1 - \frac{m}{p}\right) \ge -2m \sum_{\substack{p \mid M \\ p > 2m}} \frac{1}{p}.$$

From [24, (3.20)], we know that

$$\sum_{p \le x} \frac{1}{p} \le \operatorname{Lnln} x + B + \frac{1}{\ln^2 x},$$

where $B < 0.262$ is a constant. Let $s = \omega(M)$ and p_s be the sth prime number, so that $p_s \le s^2$ for $s \ge 2$. Thus for $s \ge 2$ we have

$$\sum_{\substack{p \mid M \\ p > 2m}} \frac{1}{p} \le \sum_{p \le p_s} \frac{1}{p} \le \sum_{p \le s^2} \frac{1}{p} \le \operatorname{Lnln}(s^2) + B + (\ln s^2)^{-2} \le \operatorname{Lnln}(s) + 2. \tag{2.6}$$

The inequality between the first and last term is also valid for $s = 1$. Now (2.4), (2.5), and (2.6) imply the claim. $\square$

3. PRAM complexity of the least bit of the inverse modulo a prime number.

In this section, we prove a lower bound on the sensitivity of the Boolean function representing the least bit of the inverse modulo p, for an n-bit prime p. For $x \in \mathbb{N}$ with $\gcd(x, p) = 1$, we recall the definition of $\mathrm{inv}_p\,(x) \in \mathbb{N}$ in (1.1). Furthermore, for $x_0, \dots, x_{n-2} \in \{0, 1\}$, we let

$$\mathrm{num}(x_0, \dots, x_{n-2}) = \sum_{0 \le i \le n-2} x_i 2^i. \tag{3.1}$$

We consider Boolean functions f with $n-1$ inputs which satisfy the congruence

$$f(x_0, \dots, x_{n-2}) \equiv \mathrm{inv}_p\,(\mathrm{num}(x_0, \dots, x_{n-2})) \bmod 2 \tag{3.2}$$

for all $x_0, \dots, x_{n-2} \in \{0, 1\}$ with $(x_0, \dots, x_{n-2}) \neq (0, \dots, 0)$. Thus no condition is imposed for the value of $f(0, \dots, 0)$.

Finally we recall the sensitivity σ from the introduction.

THEOREM 3.1. *Let p be a sufficiently large n-bit prime. Suppose that a Boolean function $f(x_0, \dots, x_{n-2})$ satisfies the congruence* (3.2). *Then*

$$\sigma(f) \ge \frac{1}{6}n - \frac{1}{2}\log n - 1.$$

Proof. We let k be an integer parameter to be determined later, with $2 \le k \le n-3$, and show that $\sigma(f) \ge k$ for p large enough. For this, we prove that there is some integer z with $1 \le z \le 2^{n-k-1}$ and

$$\mathrm{inv}_p\,(2^k z) \equiv 1 \bmod 2, \qquad \mathrm{inv}_p\,(2^k z + 2^{i-1}) \equiv 0 \bmod 2 \quad \text{for } 1 \le i \le k,$$

provided that p is large enough. We note that all these $2^k z$ and $2^k z + 2^{i-1}$ are indeed invertible modulo p.

We set $e_0 = 0$, $\delta_0 = 1$, and $e_i = 2^{i-1}$, $\delta_i = 0$ for $1 \le i \le k$. Then it is sufficient to show that there exist integers $z, w_0, \dots, w_k$ with

$$\begin{gathered}(2^k z + e_i)^{-1} \equiv 2w_i + \delta_i \bmod p,\\ 1 \le z \le 2^{n-k-1}, \quad 0 \le w_i \le (p-3)/2 \quad \text{for } 0 \le i \le k.\end{gathered} \tag{3.3}$$

Next we set $A = 2^k$, $H = 2^{n-k-2}$, $K = \lfloor (p-3)/4 \rfloor$, and $\Delta_i = 2K + \delta_i$ for $0 \le i \le k$. Then it is sufficient to find integers $x, y, u_0, \dots, u_k, v_0, \dots, v_k$ satisfying

$$\begin{gathered}\left(A(H + x - y) + e_i\right)^{-1} \equiv 2(u_i - v_i) + \Delta_i \bmod p,\\ 0 \le x, y < H, \qquad 0 \le u_0, \dots, u_k, v_0, \dots, v_k < K.\end{gathered} \tag{3.4}$$

Indeed from each solution of the system (3.4) we obtain a solution of the system (3.3) by putting $z = H + x - y$ and $w_i = K + u_i - v_i$, $i = 0, \dots, k$. On the other hand, the system (3.4) contains more variables and is somewhat easier to study. A typical application of character sum estimates to systems of equations proceeds as follows. One expresses the number of solutions as a sum over $a \in \mathbb{Z}_p$, using Lemma 2.1, then isolates the term corresponding to $a = 0$, and (hopefully) finds that the remaining sum is less than the isolated term. Usually, the challenge is to verify the last part. In the task at hand, Lemma 2.1 expresses the number of solutions of (3.4) as

$$
\begin{aligned}
&p^{-(k+1)} \sum_{0\le x,y<H}{}^{*} \sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \\
&\cdot \sum_{0\le a_0,\dots,a_k<p} \mathbf{e}_p\left(\sum_{0\le i\le k} a_i\left((A(H+x-y)+e_i)^{-1} - 2(u_i-v_i) - \Delta_i\right)\right)\\
&\quad = p^{-(k+1)} \sum_{0\le a_0,\dots,a_k<p} \mathbf{e}_p\left(-\sum_{0\le i\le k} a_i\Delta_i\right)\\
&\qquad \cdot \sum_{0\le x,y<H}{}^{*}\, \mathbf{e}_p\left(\sum_{0\le i\le k} a_i\,(A(H+x-y)+e_i)^{-1}\right)\\
&\qquad\quad \cdot \sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \mathbf{e}_p\left(\sum_{0\le i\le k} 2a_i(v_i-u_i)\right)\\
&\quad = p^{-(k+1)}(H^2K^{2(k+1)} + R),
\end{aligned}
$$

where the first summand corresponds to $a_0 = \cdots = a_k = 0$ and R to the remaining sum; we also used (2.2). For other $k+1$ tuples $(a_0,\dots,a_k)$, the sum over x,y satisfies the conditions of Lemma 2.5, with $n=k$ and $m=k+1$; indeed, we have

$$
\sum_{0\le i\le k} a_i\,(A(H+x-y)+e_i)^{-1} = \frac{f(H+x-y)}{g(H+x-y)},
$$

where

$$
g = \prod_{0\le i\le k}(AX+e_i),\;\; f = \sum_{0\le i\le k} a_i\frac{g}{AX+e_i} \in \mathbb{Z}[X].
$$

Therefore f/g is neither constant nor linear modulo p. Thus,

$$
\begin{aligned}
|R| &\le 2(k+1)Hp^{1/2} \sum_{0\le a_0,\dots,a_k<p} \left| \sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \mathbf{e}_p\left(\sum_{0\le i\le k} 2a_i(v_i-u_i)\right)\right|\\
&= 2(k+1)Hp^{1/2} \prod_{0\le i\le k}\sum_{0\le a_i<p}\left|\sum_{0\le u_i,v_i<K} \mathbf{e}_p\left(a_i(v_i-u_i)\right)\right|\\
&\le 2(k+1)Hp^{1/2}(pK)^{k+1}.
\end{aligned}
$$

We have left out the factors $|\mathbf{e}_p(-a_i\Delta_i)|$, which equal 1, transformed the summation index $2a_i$ into a_i, and used the identity (2.2).

It is sufficient to show that $H^2K^{2(k+1)}$ is larger than $|R|$, or that

$$
HK^{k+1} > 2(k+1)p^{k+3/2}. \tag{3.5}
$$

Since $K \ge (p-6)/4$, it is sufficient that

$$
2^{n-k-2} > 2(k+1)\left(\frac{p}{p-6}\right)^{k+1} p^{1/2}4^{k+1}. \tag{3.6}
$$

We now set $k = \lfloor (n - 3\log n)/6 \rfloor$, so that $6(k+1) \le n \le 2^{n-2}\ln 2 < (p-6)\ln 2$. Now $(1+z^{-1})^z < e$ for real $z > 0$, and

$$\left(\frac{p}{p-6}\right)^{k+1} < e^{6(k+1)/(p-6)} < 2.$$

Furthermore, $p^{1/2} < 2^{n/2}$ and $32n/3 < n^{3/2}$, and (3.6) follows from

$$2^{n/2} > 2^{n/2} \cdot \frac{32}{3} n \cdot 2^{-\frac{3}{2}\log n} = 64 \cdot \frac{n}{6} \cdot 2^{n/2 - \frac{3}{2}\log n} \ge 64(k+1) \cdot 2^{3k}.$$

Hence the inequality (3.5) holds, and we obtain $\sigma(f) \ge k \ge n/6 - 0.5\log n - 1$. □

From [21] we know that the CREW PRAM complexity of any Boolean function f is at least $0.5\log(\sigma(f)/3)$, and we have the following consequence.

COROLLARY 3.2. *Any CREW PRAM computing the least bit of the inverse modulo a sufficiently large n-bit prime needs at least* $0.5\log n - 3$ *steps.*

4. PRAM complexity of inversion modulo an odd squarefree integer.

In this section, we prove a lower bound on the PRAM complexity of finding the least bit of the inverse modulo an odd squarefree integer.

To avoid complications with gcd computations, we make the following (generous) definition. Let M be an odd squarefree n-bit integer and f a Boolean function with n inputs. Then f *computes the least bit of the inverse modulo M* if and only if

$$\mathrm{inv}_M\,(\mathrm{num}(x)) \equiv f(x) \bmod 2$$

for all $x \in \{0,1\}^{n-1}$ with $\gcd(\mathrm{num}(x), M) = 1$, where $\mathrm{num}(x)$ is the nonnegative integer with binary representation x, similar to (3.1). Thus no condition is imposed for integers $x \ge 2^n$ or for integers that have a nontrivial common factor with M.

THEOREM 4.1. *Let $M > 2$ be an odd squarefree integer with $\omega(M)$ distinct prime divisors, and f the Boolean function representing the least bit of the inverse modulo M, as above. Then*

$$\sigma(f) \ge \frac{\ln M - 2\omega(M)\mathrm{Lnln}\, M}{4\mathrm{Lnln}\,\omega(M) + O(1)}.$$

Proof. We let $n = \lfloor \log_2 M \rfloor$ and k be an integer parameter to be determined later. We want to show that there is some integer z with $1 \le z \le 2^{n-k-1}$ for which

$$\mathrm{inv}_M\,(2^k z) \equiv 1 \bmod 2, \qquad \mathrm{inv}_M\,(2^k z + 2^{i-1}) \equiv 0 \bmod 2 \quad \text{for } 1 \le i \le k.$$

As in the proof of Theorem 3.1, we see that in this case $\sigma(f) \ge k$.

We put $e_0 = 0$, $\delta_0 = 1$, and $e_i = 2^{i-1}$, $\delta_i = 0$ for $1 \le i \le k$. It is sufficient to show that there exist integers $z, w_0, \ldots, w_k$ such that

$$\begin{gathered}(2^k z + e_i)^{-1} \equiv 2w_i + \delta_i \bmod M,\\ 1 \le z \le 2^{n-k-1}, \quad 0 \le w_i \le (M-3)/2 \quad \text{for } 0 \le i \le k.\end{gathered}$$

Next, we set $A = 2^k$, $H = 2^{n-k-2}$, $K = \lfloor (M-3)/4 \rfloor$, and $\Delta_i = 2K + \delta_i$ for $0 \le i < k$. As in the proof of Theorem 3.1 we see that it is sufficient to find integers $x, y, u_0, \ldots, u_k, v_0, \ldots, v_k$ satisfying the following conditions for $0 \le i \le k$:

$$\begin{gathered}(A(H + x - y) + e_i)^{-1} \equiv 2(u_i - v_i) + \Delta_i \bmod M,\\ 0 \le x, y < H, \qquad 0 \le u_0, \ldots, u_k, v_0, \ldots, v_k < K.\end{gathered}$$

Lemma 2.1 expresses the number of solutions as

$$\begin{aligned}
&M^{-(k+1)} \sum_{0\le x,y<H}{}^{*} \sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \\
&\quad\cdot \sum_{0\le a_0,\dots,a_k<M} \mathbf{e}_M\left(\sum_{0\le i\le k} a_i\left(\left(A(H+x-y)+e_i\right)^{-1} - 2(u_i-v_i) - \Delta_i\right)\right)\\
&= M^{-(k+1)} \sum_{0\le a_0,\dots,a_k<M} \mathbf{e}_M\left(-\sum_{0\le i\le k} a_i\Delta_i\right)\\
&\quad\cdot \sum_{0\le x,y<H}{}^{*}\, \mathbf{e}_M\left(\sum_{0\le i\le k} a_i\left(A(H+x-y)+e_i\right)^{-1}\right)\\
&\quad\cdot \sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \mathbf{e}_M\left(2\sum_{0\le i\le k} a_i(v_i-u_i)\right)\\
&= M^{-(k+1)} \sum_{d\mid M} S_d,
\end{aligned}$$

where S_d is the subsum over those $0\le a_0,\dots,a_k<M$ for which

$$\gcd(a_0,\dots,a_k,M)=d.$$

It is sufficient to show that

$$S_M > \sum_{\substack{d\mid M\\ d<M}} |S_d|. \tag{4.1}$$

First we note that S_M consists of only one summand corresponding to $a_0=\cdots=a_k=0$. Since all values to be added equal 1, we only have to estimate the number of terms for which the argument of $\sum^*$ is defined. For each y with $0\le y<H$, we apply Lemma 2.8 to the polynomial

$$g=\prod_{0\le i\le k}\left(A(H+X-y)+e_i\right)\in\mathbb{Z}[X]$$

of degree $k+1$. We set $s=\omega(M)$, and using Lemmas 2.8 and 2.9, we deduce that

$$S_M \ge H\Big(H\exp\left(-2(k+1)\mathrm{Lnln}\, s - 7(k+1)\right) - (k+2)^s\Big)K^{2(k+1)}. \tag{4.2}$$

The other $|S_d|$ are bounded from above by

$$\begin{aligned}
|S_d| \le &\sum_{\substack{0\le a_0,\dots,a_k<M\\ \gcd(a_0,\dots,a_k,M)=d}} \left|\sum_{0\le x,y<H}{}^{*}\, \mathbf{e}_M\left(\sum_{0\le i\le k} a_i\left(A(H+x-y)+e_i\right)^{-1}\right)\right|\\
&\cdot\left|\sum_{\substack{0\le u_0,\dots,u_k,\\ v_0,\dots,v_k<K}} \mathbf{e}_M\left(2\sum_{0\le i\le k} a_i(v_i-u_i)\right)\right|.
\end{aligned}$$

Now let $d = \gcd(a_0, \dots, a_k, M)$ and

$$g = \prod_{0 \le i \le k} (AX + e_i), \quad f = \sum_{0 \le i \le k} \frac{a_i}{d} \frac{g}{AX + e_i} \in \mathbb{Z}[X].$$

Then

$$\sum_{0 \le i \le k} \frac{a_i}{d} \left(A(H + x - y) + e_i\right)^{-1} = \frac{f(H + x - y)}{g(H + x - y)},$$

and f/g is neither constant nor linear modulo any prime divisor $p \ge k+1$ of M. Thus we can apply Lemma 2.5 and find that

$$\left| \sum_{0 \le x, y < H}{}^{*} \mathbf{e}_M \left(d \sum_{0 \le i \le k} a_i/d \left(A(H + x - y) + e_i\right)^{-1} \right) \right| \le (2k+2)^s H M^{1/2} d^{1/2};$$

the hypothesis of the lemma is satisfied because M is squarefree. If $d < M$, then $a_i = db_i$ for some $0 \le b_0, \dots, b_k < M/d$, with at least one $b_i \ne 0$. Then

$$\begin{aligned}
&\sum_{\substack{0 \le a_0, \dots, a_k < M \\ \gcd(a_0, \dots, a_k, M) = d}} \left| \sum_{\substack{0 \le u_0, \dots, u_k, \\ v_0, \dots, v_k < K}} \mathbf{e}_{M/d} \left(\sum_{0 \le i \le k} 2a_i(v_i - u_i) \right) \right| \\
&\le (k+1) \sum_{\substack{1 \le b_0 < M/d \\ 0 \le b_1, \dots, b_k < M/d}} \left| \sum_{\substack{0 \le u_0, \dots, u_k, \\ v_0, \dots, v_k < K}} \mathbf{e}_{M/d} \left(\sum_{0 \le i \le k} 2b_i(v_i - u_i) \right) \right| \\
&= (k+1) \sum_{1 \le b_0 < M/d} \left| \sum_{0 \le u_0, v_0 < K} \mathbf{e}_{M/d} \left(2b_0(v_0 - u_0)\right) \right| \\
&\quad \cdot \prod_{1 \le i \le k} \sum_{0 \le b_i < M/d} \left| \sum_{0 \le u_i, v_i < K} \mathbf{e}_{M/d} \left(2b_i(v_i - u_i)\right) \right|.
\end{aligned}$$

Since M/d is odd, we may replace the summation index $2b_i$ by b_i. From the inequalities (2.3) and (2.1) we find

$$\sum_{1 \le b_0 < M/d} \left| \sum_{0 \le u_0, v_0 < K} \mathbf{e}_{M/d} \left(b_0(v_0 - u_0)\right) \right| \le \frac{M^2}{4d^2},$$

$$\sum_{0 \le b_i < M/d} \left| \sum_{0 \le u_i, v_i < K} \mathbf{e}_{M/d} \left(b_i(v_i - u_i)\right) \right| \le K^2 + \frac{M^2}{4d^2} \le \frac{5}{16} M^2 \le M^2.$$

Combining these inequalities, we obtain

$$|S_d| \le (k+1)(2k+2)^s H M^{2k+5/2} d^{-3/2};$$

therefore

$$\sum_{\substack{d|M\\ d<M}} |S_d| \le (k+1)(2k+2)^s H M^{2k+5/2} \sum_{d|M} d^{-3/2}$$
$$< \zeta(3/2)(k+1)^{s+1}2^s H M^{2k+5/2},$$

where

$$\zeta(3/2) = \sum_{h\ge 1} h^{-3/2} = 2.61\ldots.$$

Using (4.1) and (4.2) it is now sufficient to prove that

$$H\Big(H\exp\left(-2(k+1)\mathrm{Lnln}\, s - 7(k+1)\right) - (k+2)^s\Big)K^{2(k+1)}$$
$$> \zeta(3/2)(k+1)^{s+1}2^s H M^{2k+5/2}$$

for some

$$k \ge \frac{\ln M - 2s\mathrm{Lnln}\, M}{4\mathrm{Lnln}\, s + O(1)}. \tag{4.3}$$

To do so we suppose that

$$\begin{aligned}&\left(H\exp\left(-2(k+1)\mathrm{Lnln}\, s - 7(k+1)\right) - (k+2)^s\right)K^{2(k+1)}\\ &\le \zeta(3/2)(k+1)^{s+1}2^s M^{2k+5/2}\end{aligned} \tag{4.4}$$

and will show that k satisfies the opposite inequality. Obviously, we may assume that

$$k \le 0.5\ln M - 1.$$

We also recall that $K \ge (M-6)/4$ and $H = 2^{n-k-2} \ge M2^{-k-3}$. Now if

$$(k+2)^s \le 0.5H\exp\left(-2(k+1)\mathrm{Lnln}\, s - 7(k+1)\right),$$

then, because $s \le \log_2 M$, we immediately obtain (4.3). Otherwise, we derive from (4.4) that

$$\exp\left(-2(k+1)\mathrm{Lnln}\, s + O(k)\right) \le (2k+2)^s M^{-1/2} \le M^{-1/2}\exp(s\mathrm{Lnln}\, M).$$

Comparing this inequality with the inequality (4.3), we are able to obtain the desired statement. □

Our bound takes the form

$$\sigma(f) = \Omega(n/\mathrm{Lnln}\, n) \tag{4.5}$$

for an odd squarefree n-bit M with $\omega(M) \le \beta\ln M/\mathrm{Lnln}\, M$ for some constant $\beta < 0.5$. We recall that $\omega(M) \le (1+o(1))\ln M/\mathrm{Lnln}\, M$ for any $M > 1$, and that $\omega(M) = O(\mathrm{Lnln}\, M)$ for almost all odd squarefree numbers M.

We denote by $i_{\mathrm{PRAM}}(M)$ and $i_{\mathrm{BC}}(M)$ the CREW PRAM complexity and the Boolean circuit complexity, respectively, of inversion modulo M. We know from [11, 20] that

$$i_{\mathrm{PRAM}}(M) \le i_{\mathrm{BC}}(M) = O(n) \tag{4.6}$$

for any n-bit integer M. The *smoothness* $\gamma(M)$ of an integer M is defined as its largest prime divisor, and M is b-smooth if and only if $\gamma(M) \le b$. Then

$$i_{\mathrm{PRAM}}(M) \le i_{\mathrm{BC}}(M) = O(\log(n\gamma(M))). \tag{4.7}$$

Since we are mainly interested in lower bounds in this paper, we do not discuss the issue of uniformity.

COROLLARY 4.2.

$$i_{\mathrm{BC}}(M) \ge i_{\mathrm{PRAM}}(M) \ge (0.5 + o(1)) \log n \tag{4.8}$$

for any odd squarefree n-bit integer M with $\omega(M) \le 0.49 \ln M/\mathrm{Lnln}\, M$.

THEOREM 4.3. *There is an infinite sequence of moduli M such that the CREW PRAM complexity and the Boolean circuit complexity of computing the least bit of the inverse modulo M are both $\Theta(\log n)$, where n is the bit length of M.*

Proof. We construct infinitely many odd squarefree integers M with $\omega(M) \le 0.34 \ln M/\mathrm{Lnln}\, M$, thus satisfying the lower bound (4.8); and with smoothness $\gamma(M) = O(\log^3 M)$, thus satisfying the upper bound $O(\ln\ln M) = O(\log n)$ of [11] on the depth of Boolean circuits for inversion modulo such M.

For each integer $s > 1$ we select $\lfloor s/\ln s \rfloor$ primes between s^3 and $2s^3$ and let M be the product of these primes. Then, $M \ge s^{3s/\ln s} = \exp(3s)$, and thus $\omega(M) \le s/\ln s \le 0.34 \ln M/\ln\ln M$, provided that s is large enough. ∎

5. Complexity of one bit of an integer power.

For nonnegative integers u and m, we let $\mathrm{Bt}_m(u)$ be the mth lower bit of u, i.e., $\mathrm{Bt}_m(u) = u_m$ if $u = \sum_{i\ge 0} u_i 2^i$ with each $u_i \in \{0,1\}$. If $u < 2^m$, then $\mathrm{Bt}_m(u) = 0$.

In this section, we obtain a lower bound on the CREW PRAM complexity of computing $\mathrm{Bt}_m(x^e)$. For small m, this function is simple; for example, $\mathrm{Bt}_0(x^e) = \mathrm{Bt}_0(x)$ can be computed in one step. However, we show that for larger m this is not the case, and the PRAM complexity is $\Omega(\log n)$ for n-bit data.

Exponential sums modulo M are easiest to use when M is a prime, as in section 3. In section 4 we had the more difficult case of a squarefree M, and now we have the extreme case $M = 2^m$.

THEOREM 5.1. *Let m and n be positive integers with $n \ge m + m^{1/2}$, and let f be the Boolean function with $2n$ inputs and*

$$f(x_0, \ldots, x_{n-1}, e_0, \ldots, e_{n-1}) = \mathrm{Bt}_{m-1}(x^e),$$

where $x = \mathrm{num}(x_0, \ldots, x_{n-1})$ and $e = \mathrm{num}(e_0, \ldots, e_{n-1})$; see (3.1). *Then*

$$\sigma(f) \ge \gamma m^{1/2} + O(m^{1/3}),$$

where $\gamma = 3 - 7^{1/2} = 0.3542\ldots$.

Proof. We set $e = \lceil m^{1/2} \rceil$ and consider $g(x) = f(x, e)$, so that $\sigma(f) \ge \sigma(g)$. Furthermore, k is an integer parameter with $e \ge k \ge 2$ to be determined later.

To prove that $\sigma(g) \ge k$, it is sufficient to show that there exists an integer x with $0 \le x < 2^{n-e}$, $\mathrm{Bt}_{m-1}((2^e x)^e) = 0$, and $\mathrm{Bt}_{m-1}\left((2^e x + 2^i)^e\right) = 1$ for $0 \le i < k$.

The first equality holds for any such x because $e^2 \ge m$, and thus the conditions are equivalent to the existence of integers $x, u_0, \ldots, u_{k-1}$ such that

$$\begin{gathered}(2^e x + 2^i)^e \equiv 2^{m-1} + u_i \bmod 2^m, \\ 0 \le x < 2^{n-e}, \qquad 0 \le u_0, \ldots, u_{k-1} < 2^{m-1} \quad \text{for } 0 \le i < k,\end{gathered}$$

which is implied by the existence of $x, u_0 \ldots, u_{k-1}, v_0, \ldots, v_{k-1}$ with

$$\begin{gathered}(2^e x + 2^i)^e \equiv 2^{m-1} + 2^{m-2} + u_i - v_i \bmod 2^m, \\ 0 \le x < 2^{n-e}, \qquad 0 \le u_i, v_i < 2^{m-2} \quad \text{for } 0 \le i < k.\end{gathered} \tag{5.1}$$

We set $H = 2^{m-2}$ and $K = 2^{m-1} + 2^{m-2}$.

Lemma 2.1 expresses the number of solutions of (5.1) as

$$\begin{aligned}
&2^{-mk} \sum_{0 \le x < 2^{n-e}} \sum_{\substack{0 \le u_0, \ldots, u_{k-1} \\ v_0, \ldots, v_{k-1} < H}} \\
&\quad \cdot \sum_{0 \le a_0, \ldots, a_{k-1} < 2^m} \mathbf{e}_{2^m}\left(\sum_{0 \le i < k} a_i \left((2^e x + 2^i)^e - (K + u_i - v_i) \right) \right) \\
&= 2^{-mk} \sum_{0 \le a_0, \ldots, a_{k-1} < 2^m} \mathbf{e}_{2^m}(-K \sum_{0 \le i < k} a_i) \sum_{0 \le x < 2^{n-e}} \mathbf{e}_{2^m}\left(\sum_{0 \le i < k} a_i (2^e x + 2^i)^e \right) \\
&\quad \cdot \sum_{\substack{0 \le u_0, \ldots, u_{k-1}, \\ v_0, \ldots, v_{k-1} < H}} \mathbf{e}_{2^m}(\sum_{0 \le i < k} a_i (v_i - u_i)) \\
&= 2^{-mk} \sum_{0 \le \delta \le m} S_\delta,
\end{aligned}$$

where S_δ is the subsum over all integers $0 \le a_0, \ldots, a_{k-1} < 2^m$ with

$$\gcd(a_0, \ldots, a_{k-1}, 2^m) = 2^\delta.$$

It is sufficient to show that

$$S_m > \sum_{0 \le \delta < m} |S_\delta|. \tag{5.2}$$

S_m contains only one summand, for $a_0 = \cdots = a_{k-1} = 0$, and equals

$$S_m = 2^{n-e} H^{2k} = 2^{n+2mk-4k-e}. \tag{5.3}$$

Using the function μ from section 2, we have for $\delta < m$ that

$$\begin{aligned}
|S_\delta| \le \sum_{\substack{0 \le a_0, \ldots, a_{k-1} < 2^m \\ \mu(a_0, \ldots, a_{k-1}) = \delta}} &\left| \sum_{0 \le x < 2^{n-e}} \mathbf{e}_{2^m}\left(\sum_{0 \le i < k} a_i (2^e x + 2^i)^e \right) \right| \\
&\cdot \left| \sum_{\substack{0 \le u_0, \ldots, u_{k-1}, \\ v_0, \ldots, v_{k-1} < H}} \mathbf{e}_{2^m}\left(\sum_{0 \le i < k} a_i (v_i - u_i) \right) \right|.
\end{aligned}$$

Now let $a_0, \ldots, a_{k-1} < 2^m$. We set

$$h(X) = \sum_{0 \le i < k} a_i (2^e X + 2^i)^e = \sum_{0 \le j \le e} A_j X^j \in \mathbb{Z}[X], \tag{5.4}$$

so that

$$A_j = 2^{ej}\binom{e}{j}\sum_{0\le i<k} a_i 2^{i(e-j)} \qquad \text{for } 0 \le j \le e.$$

We put

$$\Delta = \mu(A_1, \ldots, A_e).$$

If $\Delta < m$, then h is periodic modulo 2^m with period $2^{m-\Delta}$:

$$h(X + 2^{m-\Delta}) \equiv h(X) \bmod 2^m.$$

Since $n - e \ge m$ and $\mathbf{e}_M(z)$ is periodic with period M, then

$$\begin{aligned}(5.5)\qquad & \left|\sum_{0\le x<2^{n-e}} \mathbf{e}_{2^m}\left(\textstyle\sum_{0\le i<k} a_i(2^e x + 2^i)^e\right)\right| \\ &= 2^{n-e-m+\Delta}\left|\sum_{0\le x<2^{m-\Delta}} \mathbf{e}_{2^{m-\Delta}}\left(2^{-\Delta}h(x)\right)\right| \\ &\le 2^{n-e-m+\Delta}\cdot c\cdot 2^{m-\Delta-(m-\Delta)/e} = c\cdot 2^{n-e-(m-\Delta)/e},\end{aligned}$$

where c is the constant from Lemma 2.6. This bound also holds for $\Delta \ge m$, because the sum contains 2^{n-e} terms with absolute value 1. Using the (crude) estimate

$$\mu\left(\binom{e}{j}\right) \le \log_2\binom{e}{j} \le \log_2 2^e \le e,$$

and noting that

$$A_{k-j} = 2^{e(k-j)}\binom{e}{k-j}\sum_{0\le i<k}\left(a_i 2^{i(e-k)}\right)2^{ij},$$

from Lemma 2.7 we derive that for tuples with $\mu(a_0, \ldots, a_{k-1}) = \delta$,

$$\begin{aligned}\Delta \le \mu(A_1, \ldots, A_k) &\le ek + e + \delta + (k-1)(e-k) + (k-1)(k-2)/2 \\ &= 2ek + \delta - (k-1)(k+2)/2 \le 2ek + \delta - k^2/2,\end{aligned}$$

provided that $k \ge 2$. Substituting this bound in (5.5), we obtain

$$|S_\delta| \le c\cdot 2^{n-e-(m-2ek-\delta+k^2/2)/e}T_\delta = c\cdot 2^{n-e-m/e+\delta/e+2k-k^2/2e}T_\delta,$$

where

$$T_\delta = \sum_{\substack{0\le a_0,\ldots,a_{k-1}<2^m \\ \mu(a_0,\ldots,a_k)=\delta}} \left|\sum_{\substack{0\le u_0,\ldots,u_{k-1}, \\ v_0,\ldots,v_{k-1}<H}} \mathbf{e}_{2^m}\left(\sum_{0\le i<k} a_i(v_i - u_i)\right)\right|.$$

We set

$$U_\delta = \begin{cases} 2^{2(m-\delta)} + 2^{2(m-2)} & \text{if } \delta \ge 3, \\ 2^{2m-\delta-2} & \text{if } 0 \le \delta \le 2.\end{cases}$$

Then $U_\delta \le 2^{2m-3}$ for all $\delta \ge 0$, and as in the proof of Theorem 4.1, from Lemma 2.2 we find

$$\begin{aligned} T_\delta &\le k \cdot \sum_{1 \le b_0 < 2^{m-\delta}} \sum_{0 \le b_1,\dots,b_{k-1} < 2^{m-\delta}} \left| \sum_{\substack{0 \le u_0,\dots,u_{k-1},\\ v_0,\dots,v_{k-1} < H}} \mathbf{e}_{2^{m-\delta}} \left(\sum_{0 \le i < k} b_i(v_i - u_i) \right) \right| \\ &\le k \cdot 2^{2(m-\delta)} U_\delta^{k-1} \le k \cdot 2^{2mk-3k-2\delta+3}. \end{aligned}$$

Next, we obtain

$$\begin{aligned} \sum_{0 \le \delta < m} |S_\delta| &\le c \cdot \sum_{0 \le \delta < m} 2^{n-e-m/e+\delta/e+2k-k^2/2e} \cdot k \cdot 2^{2mk-3k-2\delta+3} \\ &= ck \cdot 2^{n+2mk-k-e-m/e-k^2/2e+3} \sum_{0 \le \delta < m} 2^{-\delta(2-1/e)} \\ &< ck \cdot 2^{n+2mk-k-e-m/e-k^2/2e+4}. \end{aligned}$$

We set

$$k = \left\lfloor \gamma m^{1/2} - m^{1/3} \right\rfloor,$$

where $\gamma = 3 - 7^{1/2} = 0.3542\dots$ satisfies $-\gamma - 1 - \gamma^2/2 = -4\gamma$. It easy to verify that the inequality (5.2) holds for this choice of k, provided that m is large enough. ☐

COROLLARY 5.2. *Let $n \ge m + m^{1/2}$. The CREW PRAM complexity of finding the mth bit of an n-bit power of an n-bit integer is at least $0.25 \log m - o(\log m)$. In particular, for $m = \lceil n/2 \rceil$ it is $\Omega(\log n)$.*

6. Conclusion and open problems.

Inversion in arbitrary residue rings can be considered along these lines. There are two main obstacles for obtaining similar results. Instead of the powerful Weil estimate of Lemma 2.3, only essentially weaker (and unimprovable) estimates are available [16, 26, 28]. Also, we need a good explicit estimate, while the bounds of [16, 26] contain nonspecified constants depending on the degree of the rational function in the exponential sum. The paper [28] deals with polynomials rather than with rational functions, and its generalization has not been worked out yet.

OPEN QUESTION 6.1. *Extend Theorem 4.1 to arbitrary moduli M.*

Moduli of the form $M = p^m$, where p is a small prime number, are of special interest because Hensel's lifting allows to design efficient parallel algorithms for them [2, 11, 15]. Theorem 5.1 and its proof demonstrate how to deal with such moduli and what kind of result should be expected.

Each Boolean function $f(X_1, \dots, X_n)$ can be uniquely represented as a multilinear polynomial of degree n over $\mathbb{F}_2$ of the form

$$f(X_1, \dots, X_n) = \sum_{0 \le k \le d} \sum_{1 \le i_1 < \dots < i_k \le r} A_{i_1 \dots i_k} X_{i_1} \dots X_{i_k} \in \mathbb{F}_2[X_1, \dots, X_n].$$

We define its weight as the number of nonzero coefficients in this representation. Both the weight and the degree can be considered as measures of complexity of f. In [5, 25], the same method was applied to obtain good lower bounds on these characteristics of the Boolean function f deciding whether x is a quadratic residue modulo p. However, for the Boolean functions of this paper, the same approach produces rather poor results.

OPEN QUESTION 6.2. *Obtain lower bounds on the weight and the degree of the Boolean function f of Theorem* 4.1.

It is well known that the modular inversion problem is closely related to the GCD-problem.

OPEN QUESTION 6.3. *Obtain a lower bound on the PRAM complexity of computing integers u, v such that $Mu + Nv = 1$ for given relatively prime integers $M \geq N > 1$.*

In the previous question we assume that $\gcd(N, M) = 1$ is guaranteed. Otherwise one can easily obtain the lower bound $\sigma(f) \geq \Omega(n)$ on the *sensitivity* of the Boolean function f which on input of two n-bit integers M and N, returns 1 if they are relatively prime, and 0 otherwise. Indeed, if $M = p$ is an n bit integer, then the function returns 0 for $N = p$ and 1 for all other n bit integers. That is, the PRAM complexity of this Boolean function is at least $0.5 \log n + O(1)$.

Acknowledgment. This paper was essentially written during a sabbatical visit by the second author to the University of Paderborn, and he gratefully acknowledges its hospitality and excellent working conditions.

REFERENCES

[1] L. M. ADLEMAN AND K. KOMPELLA, *Using smoothness to achieve parallelism*, in Proceedings of the 20th ACM Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 528–538.

[2] P. W. BEAME, S. A. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[3] G. CESARI, *Parallel implementation of Schönhage's integer GCD algorithm*, in Proceedings of ANTS-III, Lecture Notes in Comput. Sci. 1423, Springer-Verlag, Berlin, 1998, pp. 64–76.

[4] S. A. COOK, C. DWORK, AND R. REISCHUK, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[5] D. COPPERSMITH AND I. E. SHPARLINSKI, *On polynomial approximation of the discrete logarithm and the Diffie–Hellman mapping*, J. Cryptology, to appear.

[6] M. DIETZFELBINGER, M. KUTYŁOWSKI, AND R. REISCHUK, *Exact time bounds for computing Boolean functions on PRAMs without simultaneous writes*, J. Comput. System Sci., 48 (1994), pp. 231–254.

[7] M. DIETZFELBINGER, M. KUTYŁOWSKI, AND R. REISCHUK, *Feasible time-optimal algorithms for Boolean functions on exclusive-write parallel random-access machine*, SIAM J. Comput., 25 (1996), pp. 1196–1230.

[8] F. E. FICH, *The complexity of computation on the parallel random access machine*, in Synthesis of Parallel Algorithms, J. H. Reif, ed., Morgan Kaufmann, San Mateo, CA, 1993, pp. 843–899.

[9] F. E. FICH AND M. TOMPA, *The parallel complexity of exponentiating polynomials over finite fields*, J. ACM, 35 (1988), pp. 651–667.

[10] S. GAO, J. VON ZUR GATHEN, AND D. PANARIO, *Gauss periods and fast exponentiation in finite fields*, in Proceedings of LATIN'95, Lecture Notes in Comput. Sci. 911, Springer-Verlag, Berlin, 1995, pp. 311–322.

[11] J. VON ZUR GATHEN, *Computing powers in parallel*, SIAM J. Comput., 16 (1987), pp. 930–945.

[12] J. VON ZUR GATHEN, *Inversion in finite fields using logarithmic depth*, J. Symbolic Comput., 9 (1990), pp. 175–183.

[13] J. VON ZUR GATHEN, *Efficient and optimal exponentiation in finite fields*, Comput. Complexity, 1 (1991), pp. 360–394.

[14] J. VON ZUR GATHEN, *Processor–efficient exponentiation in finite fields*, Inform. Process Lett., 41 (1992), pp. 81–86.

[15] J. VON ZUR GATHEN AND G. SEROUSSI, *Boolean circuits versus arithmetic circuits*, Inform.

and Comput., 91 (1991), pp. 142–154.

[16] D. Ismoilov, *On a method of Hua Loo-Keng of estimating complete trigonometric sums*, Adv. Math. (Beijing), 23 (1994), pp. 31–49.

[17] R. Kannan, G. Miller, and L. Rudolph, *Sublinear parallel algorithm for computing the greatest common divisor of two integers*, SIAM J. Comput., 16 (1987), pp. 7–16.

[18] R. Lidl and H. Niederreiter, *Finite Fields*, Addison-Wesley, Reading, MA, 1983.

[19] B. E. Litow and G. I. Davida, $O(\log(n))$ *parallel time finite field inversion*, in VLSI Algorithms and Architectures, Lecture Notes in Comput. Sci. 319, Springer-Verlag, New York, 1988, pp. 74–80.

[20] M. Mňuk, *A div*(n) *depth Boolean circuit for smooth modular inverse*, Inform. Process Lett., 38 (1991), pp. 153–156.

[21] I. Parberry and P. Y. Yan, *Improved upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 20 (1991), pp. 88–99.

[22] G. B. Purdy, *A carry-free algorithm for finding the greatest common divisor of two integers*, Comput. Math. Appl., 9 (1983), pp. 311–316.

[23] J. B. Rosser and L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Ilinois J. Math., 6 (1962), pp. 64–94.

[24] I. E. Shparlinski, *Finite fields: Theory and Computation*, Kluwer Academic Publishers, Dordrecht, 1999.

[25] I. E. Shparlinski, *Number Theoretic Methods in Cryptography: Complexity Lower Bounds*, Birkhäuser, Cambridge, MA, 1999.

[26] I. E. Shparlinski and S. A. Stepanov, *Estimates of exponential sums with rational and algebraic functions*, in Automorphic Functions and Number Theory, Vladivostok, 1989, pp. 5–18 (in Russian).

[27] J. Sorenson, *Two fast GCD algorithms*, J. Algorithms, 16 (1994), pp. 110–144.

[28] S. B. Stečkin, *An estimate of a complete rational exponential sum*, Proc. Math. Inst. Acad. Sci. USSR, 143 (1977), pp. 188–207 (in Russian).

[29] K. Weber, *Parallel implementation of the accelerated integer GCD algorithm*, J. Symbolic Comput., 21 (1996), pp. 457–466.

[30] I. Wegener, *The Complexity of Boolean Functions*, Wiley Interscience , New York, 1987.

[31] A. Weil, *Basic Number Theory*, Springer-Verlag, New York, 1974.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1858–1877

DYNAMIC MAINTENANCE OF MAXIMA OF 2-D POINT SETS[*]

SANJIV KAPOOR[†]

Abstract. This paper describes an efficient scheme for the dynamic maintenance of the set of maxima of a 2-d set of points. Using the fact that the maxima can be stored in a staircase structure, we use a technique in which we maintain approximations to the staircase structure. We first describe how to maintain the maxima in $O(\log n)$ time per insertion and deletion when there are n insertions and deletions. $O(\log n)$ is charged per change for reporting changes to the staircase structure which stores the maxima. $O(n)$ space is used. We also show another scheme which requires a total of $O(n \log n + r)$ time when r maximal points are listed. We finally consider extensions to higher dimensions.

Key words. maximal points, dynamic maintenance, balanced trees

AMS subject classifications. 68Q25, 68U05, 68P05

PII. S0097539798348365

1. Introduction. Given a set S of points in the x-y plane, $p_i = (x_i, y_i) \in S$ is a maximal point iff it is not dominated by another point p_j, where a point p_j dominates p_i iff $x_i < x_j$ and $y_i < y_j$. The set of maximal points of S form a structure which monotonically decreases in the y direction as the x-coordinate of the points increases. Such a structure is called a staircase structure. The problem of computing the maxima occurs in a large number of applications in statistics, economics, operations research, etc. The reader is referred to the book by Preparata and Shamos for [FP] further details.

In this paper we describe an efficient data structure for the dynamic maintenance of the maxima of a point set. We maintain approximations to the maxima set in a structure called the approximate staircase structure.

For the static case it has been shown that the staircase structure can be computed in $O(n \log n)$ time, eliminating dominated points [KLP]. In the dynamic case, however, one needs to keep track of the dominated points. In this case Overmars and Van Leeuwen [OL] have designed a data structure which requires splitting and merging balanced trees when points are inserted and deleted. In their scheme $O(\log^2 n)$ operations are required for each insertion and deletion. A scheme by Willard and Lueker [WL] gives a bound of $O(\log n)$ for updates but the set of maxima is not maintained. Frederickson and Rodger [GR] and Janardan [J] have designed schemes which maintain the staircase structure of the set of maxima and allow for insertions and deletions in $O(\log n)$ and $O(\log^2 n)$ operations, respectively. These are the most asymptotically efficient schemes known. Their data structure requires $O(n)$ space. Also, Hershberger and Suri [HS] give a scheme for off-line maintenance.

We first present an improved data structure that maintains an approximate staircase structure in $O(\log n)$ time per insertion and deletion. A point can be tested

[*]Received by the editors November 24, 1998; accepted for publication June 29, 1999; published electronically April 4, 2000. Part of this work was done while the author was visiting the Max-Planck-Institute für Informatik, Saarbrücken, Germany. This work was also supported in part by grant SR/OY/E13/92 from the Department of Science and Technology, India, and by the Max-Planck-Institute für Informatik, Germany

http://www.siam.org/journals/sicomp/29-6/34836.html

[†]Department of Computer Science and Engineering, Indian Institute of Technology, Hauz Khas, New Delhi 110016, India (skapoor@cse.iitd.ernet.in).

for maximality in $O(\log n)$ time in our representation. The staircase structure representing the set of maxima can be listed in linear time but $O(\log n)$ operations have to be spent on removal of every approximation present in the current semistaircase structure to be reported. Our data structure is simple and requires $O(n)$ space.

We further show that the data structure can be modified to give a total bound of $O(n \log n + m \log n + r)$ when there are n insertions, m deletions, and r maximal points are reported.

The methodology that we have presented here may also be applicable to dynamic maintenance of other geometric structures such as convex hulls, intersections of half planes, and kernels of simple polygons. Since the optimal dynamic maintenance of these geometric structures has been an open problem for some time, the solution in this paper may be of interest in terms of technique.

We also outline an extension of our scheme to higher dimensions but only for the case of insertions. Maintaining the maxima efficiently, under both deletions and insertions, in higher dimensions is a challenging problem.

2. Preliminaries and outline of the paper.

2. Preliminaries and outline of the paper. In this paper we first describe the structural changes required for maintaining the maxima under insertions and deletions. We then show how to implement the scheme in a data structure requiring $O(n)$ space.

The underlying data structure that we use for storing the points is the red-black tree [T]. We call it the maxima balanced tree. The set of points, S, are stored at the leaves of a red-black tree, which we denote by $MT(S)$, in order of increasing x-coordinate values. At each node we store the set of maxima of the subset of points stored in the subtree rooted at that node. This subset is not stored explicitly as we will see. The initial tree can be constructed in $O(n \log n)$ time easily. Note that red-black trees can be maintained in $O(\log n)$ steps at each update.

We let $ST(q)$ be the set of points in the subtree rooted at q and $MAX(q)$ the set of maxima of $ST(q)$ at node q in the tree. Furthermore, we assume that at each node the maxima are specified in a sorted order with points having decreasing y-coordinate and increasing x-coordinate. We let $M(q)$ represent this ordered set. It is known that the set $M(q)$ has a staircase structure. $Top(M(q))$ represents the point with the maximum y-coordinate in $M(q)$. We let $y(p)$ refer to the y-coordinate of point p. We use $y(TOP(M(q)))$ and $TOP(M(q))$ interchangeably when making comparisons for ease of presentation. Finally, we let $right(v)$ and $left(v)$ refer to the left and right children of an internal node v in the binary tree, $MT(S)$ storing the maxima.

The first scheme that we propose uses the fact that during insertions binary search is performed only once after which the inserted point is dominated. This fact has also been used in [GR], [J] in their data structure. We thus obtain a solution with $O(\log n)$ insertion time and $O(\log^2 n)$ deletion time. We also isolate the reason for the $O(\log^2 n)$ behavior of the scheme. We consider only *structural* changes, i.e., changes to the staircase structure. This scheme is briefly described in section 3 to motivate the final improved method.

Then, in section 4, we refine the algorithm so that both insertions and deletions can be done in $O(\log n)$ time. We do so by maintaining approximations to the staircase structure. This allows us to use only a constant number of binary searches to update the balanced tree structure storing the maxima. We describe the improvements in time complexity for making structural changes without a description of the secondary structures required to store the maxima at the nodes of the balanced binary tree.

Finally, in section 5, the secondary structures required to store the maxima are described. The space requirements are shown to be $O(n)$. In section 6 we show how to modify the scheme to list the maximal points in linear amortized time. In section 7 we consider insertions of points in d-dimensions.

3. An $O(\log n)$ insertion and $O(\log^2 n)$ deletion scheme.

In this section we describe a scheme for insertions and deletions which is suboptimal. The scheme is described without details of secondary structures which we consider in detail for the improved scheme in a later section.

The following definitions will be required.

The *staircase* structure is a sequence of points, simultaneously sorted in increasing x order and in decreasing y order, such that two adjacent points are connected by a horizontal and vertical Manhattan path.

A *stairstep* is the horizontal and vertical Manhattan path that connects two adjacent points in a staircase structure.

A point p is *located* onto a point q in $M(v), v \in MT(S)$ when $q \in M(v)$ is the point with the least y-coordinate such that $y(q) > y(p)$. The point p is also said to be located onto the stairstep formed by q.

We will use the following characterization [GR, FP].

LEMMA 1. *The maximal elements of a set of points in the plane form a staircase structure.*

We will refer to the staircase structure, comprising maximal elements of the set of points in the subtree rooted at a node v, as the staircase structure at the node. This structure is also referred to by $M(v)$.

We first consider insertions. Let p be the point to be inserted into the tree $MT(S)$ storing the set of maxima of the point set S. Since the leaves are arranged in x-sorted order a binary search gives us the leaf at which the point p is to be inserted. Let P be the path from the root to the leaf. Suppose, on the path P, we encounter a triple of nodes (u, v, w), where v is the common parent of u and w. u is the left child and is referred to by $left(v)$, and w is the right child and referred to by $right(v)$. P uses v and one of u and w.

First suppose the path P uses w. Assume that the maxima of the points in the subtree at w has been recomputed because of the insertion of the new point p. $M(v)$ has to be recomputed from $M(u)$ and $M(w)$ by a merger. The merger, in general, involves locating $Top(M(w))$ in $M(u)$ and then replacing the portion of $M(u)$ dominated by $M(w)$ by $M(w)$ itself. However when a single point, p, is inserted, $M(w)$ changes only locally. The merger operation can now be accomplished by changes to the current staircase structure instead of complete recomputation. There are a number of cases depending on the topmost point of $M'(w)$, $Top(M'(w))$, where $M'(w)$ is the updated staircase structure at w. We assume that $Top(M'(w))$ has changed. If it has not, then no change in the merged staircase structure, $M(v)$, is required.

Case R.1. $Top(M'(w))$ is above $Top(M(v))$. In this case the merger requires constant time since $M'(v)$ is actually $M'(w)$ (Figure 1(i)).

Case R.2. $Top(M'(w))$ is not above $Top(M(v))$. In this case the merger is done by first locating $Top(M'(w))$ by a binary search on $M(u)$ to obtain the first point, q, in $M(u)$ such that $y(q) > Top(M'(w))$. The portion of $M(u)$ below q is now replaced by $M'(w)$ (Figure 1(ii)). This requires $O(\log n)$ time. Note that from this node upwards to the root the topmost point of any staircase structure does not change since the inserted point is now within a staircase structure and new merger points need not be recomputed.

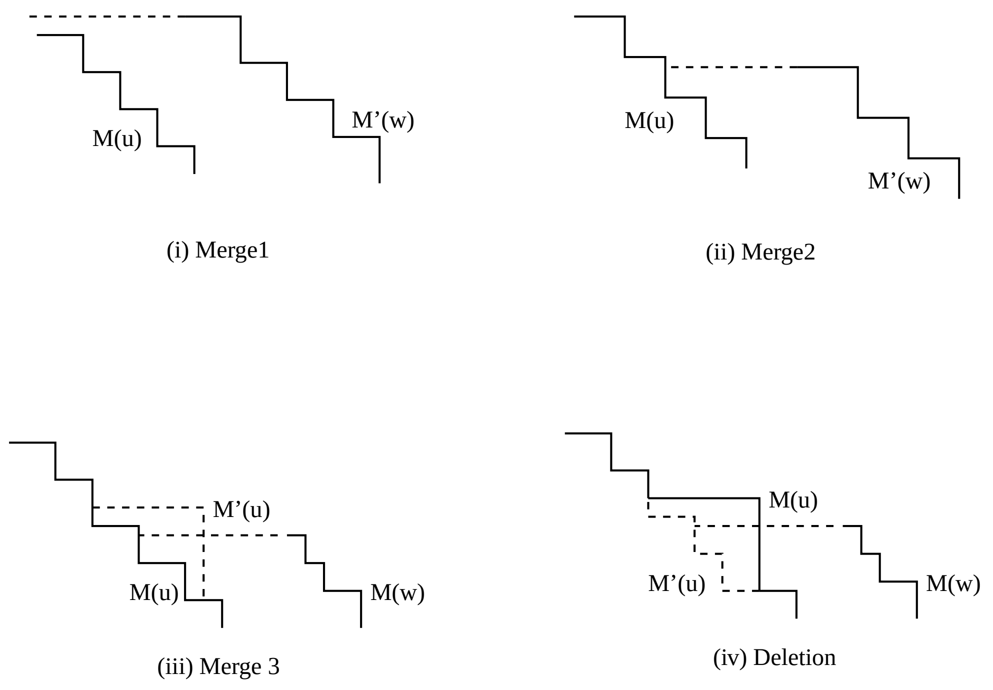


FIG. 1. *Insertion and deletion cases.*

Next consider the case when P uses u. Again, suppose $M(u)$ has been modified and let the modified structure be $M'(u)$. There are two cases:

Case L.1. $Top(M(u))$ has changed. There are two subcases. If $Top(M(w))< Top(M'(u))$, then locate $Top(M(w))$ in $M'(u)$. This is done by comparing $Top(M(w))$ with the stairstep formed by the inserted point. If $Top(M(w))$ is located below this stairstep, then the previous location of $Top(M(w))$ is valid, and $M(u)$ is updated using this merge point. Otherwise, $Top(M(w))$ is located onto the stairstep, and $M'(v)$ is obtained by adding $Top(M'(u))$ to the staircase structure $M(w)$.

Alternatively, suppose $Top(M(w)) \geq Top(M'(u))$. Then $M'(v)$ is $M(w)$.

Case L.2. The topmost point of $M(u)$ has not changed. In this case we have to determine the changes that are to be made to $M(v)$ since the top of $M(w)$ has to be positioned with respect to the changes in $M(u)$. As in the previous case this can be done in constant time since only a comparison of $Top(M(w))$ with the stairstep formed by the point inserted into $M(u)$ has to be made, provided that point is in the set of maxima (Figure 1(iii)). This information is easily maintained as we proceed up the path P.

We thus see that the above procedure requires $O(\log n)$ steps for an insertion since a binary search is performed once only in Case L.2.

We next consider the effect of deletions. Again, we consider a path P from leaf to root and a triple of nodes (u, v, w) on the path. First consider the case when the path P uses w. Suppose the maximal set has been recomputed at w and is $M'(w)$. There are two cases depending on $Top(M(w))$: Case R.1, where $Top(M(w))$ changes (this case has two subcases); and Case R.2, when $Top(M(w))$ does not change. We consider the cases.

Case R1.1. $Top(M'(w)) < Top(M(u))$. Then $M(v)$ can be computed in $O(\log n)$ time by locating $Top(M'(w))$ in $M(u)$ by a binary search.

Case R1.2. $Top(M'(w)) > Top(M(u))$. Update $Top(M(v))$ in $O(1)$ time since $M(v)$ is $M'(w)$.

Case R2. $Top(M(w))$ does not change. $M(v)$ changes exactly as $M(w)$ is changed in the portion of the staircase common to both $M(v)$ and $M(w)$. In this case the staircase structure at v need not be searched but only changed at $Top(M(w))$ to incorporate $M'(w)$.

Next consider the case when the path P uses u. $M(v)$ is obtained from $M'(u)$ and $M(w)$ by a merge operation and requires $O(\log n)$ operations (Figure 1(iv)).

Finally we consider the rebalancing operations. Rebalancing involves $O(1)$ single or double rotations. Consider single rotations. The update of $M(u)$, where u is a node with right and left child u_1 and u_2, respectively, and which participates in the rebalancing can be performed in $O(\log n)$ time since the update requires a location of $Top(M(u_1))$ in $M(u_2)$. Double rotations can be handled similarly.

From the above case description, we obtain the following theorem.

THEOREM 1. *The structural changes required to maintain the set of maxima require $O(\log n)$ steps for each insertion and $O(\log^2 n)$ steps for each deletion.*

It is the first subcase of the first case, Case R1.1, during deletions, when the path uses w and v, that creates the first problem. It gives a time bound of $O(\log^2 n)$ since the merge of $M(u)$ and $M'(w)$ requires $O(\log n)$ time, because of the search for $Top(M'(w))$ in $M(u)$, and there may be $O(\log n)$ such merges to be performed. The second problem occurs since $Top(M'(u))$ may be changed at $O(\log n)$ nodes along path P. This occurs when the path P uses u in the second case above. There may be $O(\log n)$ merges of this type requiring $O(\log n)$ operations each.

We next show how to improve the performance of the deletion operation to $O(\log n)$ per deletions also.

4. Improving the performance.

4.1. Outline. In order to improve the performance we attempt to remove the bottlenecks introduced by the repeated mergers that are required on path P when either $M'(w)$ and $M(u)$ or $M'(u)$ and $M(w)$ are merged after a deletion. This is done by performing a constant number of mergers as outlined below:

Let $Mset(P)$ be the set of nodes on the path P where the staircase structure is changed due to the deletion. Let w_i be a node on P with $left(w_i)$ its left child and $right(w_i)$ its right child. $Mset(P)$ consists of two types of nodes. One type, type 1, has the property that $Top(M(w_i)), w_i \in Mset(P)$ is changed due to the deletion. The other type, type 2, has the property that $M(right(w_i))$ is merged with $M(left(w_i))$ when $Top(right(M(w_i)))$ intersects the section of the staircase, $M(left(w_i))$, that is changed due to the deletion. Note that $|Mset(P)| = O(\log n)$. We only do two mergers requiring a binary search for each type of nodes in $Mset(P)$. One is the merger at $M(w_k)$, where w_k is the node in the binary tree such that $Top(M(w_k))$ exceeds the top, $Top(M(w_i))$, of all nodes w_i, of type 2 in the set $Mset(P)$. The other merger is due to the location of $Top(M(w_l))$, where $Top(M(w_l))$ exceeds the top, $TOP(M(w_j))$, of all nodes, w_j, of type 1. All the other mergers required are deferred. Instead, $M(right(w_i))$ is merged with $M(left(w_i))$ by joining $Top(M(right(w_i)))$ to one of the following points:

(i) The point in $M(left(w_i))$ with y-coordinate just less than that of $TOP(M(w_k))$ when w_i is a node of type 2.

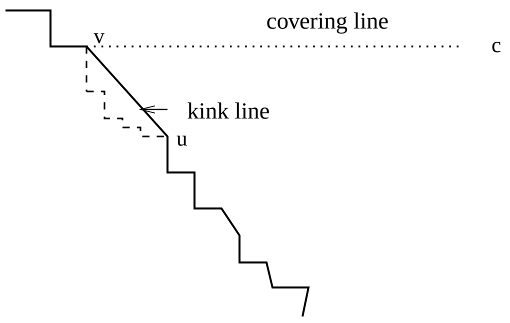


FIG. 2. *Semistaircase structure.*

(ii) $Top(M(left(w_i)))$, if $Top(M(left(w_i))) > Top(M(right(w_i)))$, when $right(w_i)$ is a node of type 1.

This destroys the staircase nature of the maximal set, creating nonhorizontal and vertical lines called "kinks," but the deletion now requires $O(\log n)$ steps.

The insertion also requires $O(\log n)$ steps. It may so happen that during insertion or deletion the top point of a set of "kinks" changes. The kinks thus shift. However, we show that there are only $O(\log n)$ kink edges to be shifted.

4.2. Detailed algorithm. First, we formally define the semistaircase structure.

DEFINITION 1. *The semistaircase structure is a sequence of points, simultaneously sorted in increasing x order and in decreasing y order, such that two adjacent points are either connected by a horizontal and vertical Manhattan path or by a straight line (called a kink line) (Figure 2).*

DEFINITION 2. *A stairstep is the horizontal and vertical Manhattan path that connects two adjacent points in a staircase or a semistaircase structure.*

DEFINITION 3. *A kink line (p, q) is an approximation to a staircase structure which connects the two points p and q with $x(p) > x(q)$ and $y(p) < y(q)$.*

DEFINITION 4. *A point p is said to be* located *onto a point q in $M(v), v \in MT(S)$ when $q \in MAX(v)$ is the point with the least y-coordinate such that $y(q) > y(p)$. The point p is also said to be located onto the stairstep formed by q.*

Note the difference in the definition of *located* with respect to the previous section.

At the nodes of the data structure $MT(S)$, we store approximations to sets of maxima. Each approximation will be a subset of the maximal elements, and at node v the approximation will be referred to by $M(v)$ for ease of notation. $M(v)$ will be represented by a semistaircase structure. The following invariant holds during the execution of the algorithm.

Property Semi. If $p \in M(v)$, then p is a maximal point in the set of points stored in the subtree rooted at V.

We next show how to dynamically maintain semistaircase structures under insertions and deletions.

4.3. Insertions. We first consider insertions. When inserting point p, let P be the path from the leaf containing p to the root. As we proceed up the path consider the triple of nodes (u, v, w) such that v is the parent of u and w.

Suppose the path P uses w. The cases are similar to the previous section and involve a binary search with $Top(M(w))$ onto a semistaircase structure. As in the previous section one location of $Top(M(w))$ in a semistaircase structure is required.

Before describing the updates we describe two generic procedures for the location of $Top(M(w))$ in $M(u)$. The procedure $LOCATE(p, z)$ locates a point p in a semistaircase at node z. And the procedure $RELOCATE(z_1, z_2)$ updates the semistaircase after locating $Top(M(z_1))$ in $M(z_2)$, where z_1 and z_2 are the right and left children of a node z.

Procedure $LOCATE(p, z)$.

The procedure is recursive. We start with node z. In the generic step, at any node a in the tree we check if the point is located in the part of the semistaircase at the left child or at the right child and we recur in the appropriate subtree. This check is done as follows: Suppose b and c are the right and left children at a and $M(b)$ is merged with $M(c)$ by a horizontal edge, h. Then the procedure branches to the left if $y(p)$ is greater than the y-coordinate of points on h; otherwise the procedure branches to the right. Alternatively, if at a there is a kink edge (r, s) merging the semistaircase at the right child with the one at the left child, then the procedure recurs to the left if $y(p) \geq y(r)$. Otherwise the procedure recurs to the right. The procedure stops at a leaf having discovered on the staircase a point with y-coordinate just greater than $y(p)$. This is the point adjacent to p in the staircase and defines the location of p. The procedure requires $O(\log n)$ steps. Also those $O(\log n)$ kink edges on the search path which intersect $y = y(p)$ are obtained. We summarize this below.

LEMMA 2. *$LOCATE(p, z)$ correctly locates point p in the semistaircase at node z in $O(\log n)$ steps.*

We next describe the second procedure.

Procedure $RELOCATE(z_1, z_2)$.

In this procedure, $Top(M(z_1))$, denoted by p', is first located in $M(z_2)$ using $LOCATE(Top(M(z_1)), z_2)$. To update the staircase structures after the location of p' is done, suppose the location of p' is q at node l. First, the stairstep joining $Top(M(z_1))$ to q is constructed, all points with the y-coordinate less than q in $M(z_2)$ are removed, and $M(z_1)$ is added to the semistaircase structure at z. Next all the kink edges intersected by $y = y(Top(M(z_1)))$ during the location procedure have their top point changed. The procedure for doing this is as follows: Let $lp(p')$ be the path followed by the location procedure for locating p'. Let d be a node on $lp(p')$ where a kink edge is used to merge $M(right(d))$ with $M(left(d))$. $Top(right(d))$ is merged using the point q', just below q, in the semistaircase $M(left(d))$, if such a point exists. Otherwise $Top(right(d))$ is located onto q. The point q' may be found as follows:

(i) Start at node l with q' being a point with y-coordinate $-\infty$.

(ii) At each node z, $z \in lp(p')$ such that $right(z) \notin lp(p')$, q' is updated as follows: If $y(Top(right(z))) > y(q')$, then $q' = Top(right(z))$.

Note that q' is defined not to exist if $y(q') = -\infty$.

The following result will be proved formally in a later section.

LEMMA 3. *$RELOCATE(z_1, z_2)$ correctly locates $TOP(M(z_1))$ in $M(z_2)$ and updates semistaircase structures in $O(\log n)$ steps.*

The above procedures are used to update the staircase structure when path $P(p)$ uses w and a location is required. The procedure called $RELOCATE(Top(M'(w)), u)$ suffices.

When path P uses u, the update after an insertion is again in constant time as described in the previous section.

Thus a point can be inserted and semistaircase structures updated in $O(\log n)$ steps.

4.4. Deletion. We consider the various cases that arise during a deletion. Again consider the triple (u, v, w). Suppose P, the path from the leaf deleted (corresponding to point q) to the root, uses w. We consider the cases depending on whether $M'(w)$ has its topmost point different from $M(w)$ or not.

Case L.1. $Top(M'(w))$ is changed, i.e., the topmost point of the previous maximal set has been deleted. In this case $Top(M'(w))$ is to be located in $M(u)$ to construct $M(v)$. There are two subcases: Either the new topmost point is the topmost point of $M(v)$ (this check can be done in $O(1)$ time); alternatively, the topmost point has to be relocated within $M(u)$ in $O(\log n)$ time. This can be achieved by the procedure $RELOCATE(w, u)$ which locates $Top(M'(w))$ in $M(u)$. However, we need to do this only when the set of maxima might be affected.

We first describe when the relocation of points in the semistaircases is necessary: Let $Wset$ be the set of nodes whose Top has changed due to the deletion of the particular node. The Top value of these nodes can be ordered by increasing y-coordinates. A relocation is done at the node, say t, with the highest Top value. Furthermore, consider a node $a, a \neq t$ and $a \in Wset$. Let b be the parent of a and c its sibling to the left. $Top(M(a))$ is not relocated but is joined to $Top(M(c))$ by a kink edge.

Case L.2. $Top(M'(w))$ has not changed. In this case , $M(u)$ changes in the section comprising points from $M'(w)$ but no new merger is required.

Next suppose the path P uses u. We assume by induction that $M'(u)$ has a semistaircase structure.

Case R.1. $M(w)$ intersects $M(u)$ at the stairstep associated with the removed point. Let $R(q)$ be the semistaircase that replaces the point q deleted. We let RP be the set of right children of nodes on the path P such that if $a \in RP$, then $Top(M(a))$ is to be located within the the stairstep associated with q and thus within $R(q)$. Let b be the node in RP such that $Top(M(b)) \geq Top(M(a)), a \in RP$. Then $Top(M(b))$ is located in $R(q)$ to give a point z which is used to form a stairstep onto which q is located, with z being the point just above $Top(M(b))$ in the semistaircase structure at the parent of b. For all $a \in RP, a \neq b$, a direct line (kink line) is drawn from $Top(M(a)), a \in RP$ to a point t in $R(q)$ where t is the point in $R(q)$ such that $y(t)$ is just less than $y(z)$ and $y(t) > Top(M(a))$. If such a t does not exist, say when $Top(M(a)) > y(t)$, then a normal staircase is constructed with z being the point next to $TOP(M(a))$ in the semistaircase structure at the parent of a. Note that this procedure involves delaying the construction of the straight lines until z is obtained. All the merges required can be done in $O(\log n)$ time (Figure 3).

Case R.2. $M(w)$ does not intersect $M(u)$ at the stairstep associated with the changed node. Then no change in the merger need be done.

The above cases complete the description of the changes required after a deletion. It is easily seen that $O(\log n)$ steps are required.

Furthermore, balancing requires $O(\log n)$ steps at each single or double rotation. Consider the case of a single rotation. Let z be a node affected by the rotation with z_1 and z_2 being the right and left child, respectively. The semistaircase structure at z is updated by locating $Top(M(z_1))$ in $M(z_2)$ using the procedure $RELOCATE(z_1, z_2)$. Since there are at most $O(1)$ nodes affected, maintenance of the semistaircase structure at a single rotation can be achieved in $O(\log n)$ steps. A similar analysis applies to the case of double rotations. Since there are at most $O(1)$ single and double rotations while balancing red-black trees, an overall bound of $O(\log n)$ on the balancing of the maxima balanced tree $MT(S)$ follows.

Note, however, that at the end of the deletion kinks may be left on the final staircase structure at the root.

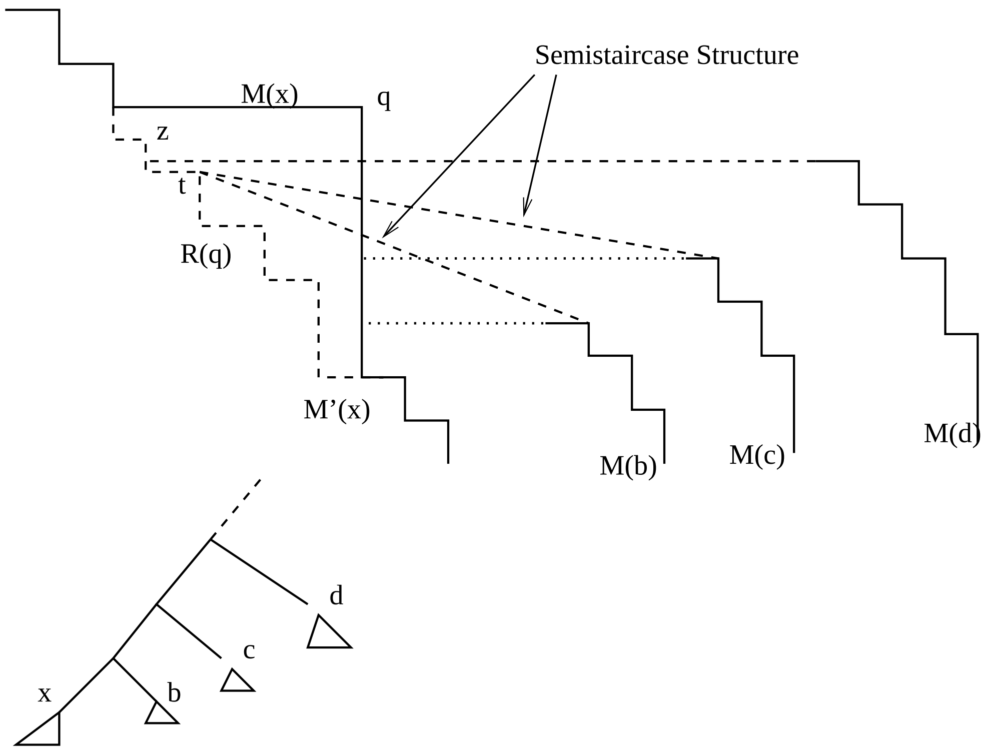


FIG. 3. *Deletion of point q.*

4.5. Correctness. We next show that the insertion and deletion procedures are correct. It suffices to show that a semistaircase is maintained at each of the nodes.

We first prove Lemma 3.

LEMMA 3. *$RELOCATE(z_1, z_2)$ correctly locates $TOP(M(z_1))$ in $M(z_2)$ and updates semistaircase structures in $O(\log n)$ steps.*

Proof. To prove the correctness first note that the $LOCATE$ procedure correctly computes the location of $p' = Top(M(z_1))$ in $M(z_2)$. Let $lp(p')$ be the set of nodes traversed during the location of p'. And let l be the node at which p' is located. We next show that kink edges intersected by $y = Top(M(z_1))$ are updated correctly. Let d be the node on $lp(p')$ where such a kink edge is present. It suffices to show that q', the point just below q in $M(left(d))$, is computed correctly. Note that q' is the point with the highest y-coordinate amongst points in $ST(d)$ located onto the stairstep formed by q. This point is a maximal point in $M(left(d))$. Consider the set of potentially maximal points in $ST(d)$ located onto the stairstep formed by q, $Mlp(d) = \{Top(M(right(z))), z \in Rlp(d)\}$, where $Rlp(d)$ is the set of nodes on $lp(p')$ lying between l and d with right children not on $lp(p')$. q' is an element of this set since for every point in $ST(d)$ located onto q there exists a point in $Mlp(d)$ with greater y-coordinate value. The set $Mlp(d)$ and thus q' is correctly computed by the procedure.

We next prove the correctness of the changes made to the semistaircase structures. Consider the path in MT, $lp(p')$, followed by $LOCATE$. The insertion procedure changes semistaircase structures on $lp(p')$ after the location of p'. We prove that semistaircases along $lp(p')$ are maintained correctly. The proof is by induction on the distance to the leaf node. The basis is easily established. Let q be the location of p'. And v be a node on $lp(p')$. Let q' be the point in $M(left(v))$ just below q. A merger is updated when the line $y = y(p')$ intersects a kink line joining $Top(right(v))$ to a point r with $y(r) > y(p')$. By induction $M(left(v))$ is a semistaircase structure. Thus all points s with $y(s) \geq y(q')$ in $M(left(v))$ are maximal in $ST(v)$. Moreover, all points in $M(right(v))$ are maximal in $ST(right(v))$. Since $Top(right(v))$ is connected to either q' or located onto q by a horizontal edge, all points in $M(v)$ are also maximal in $ST(v)$.

Finally, we bound the time complexity. First, $LOCATE$ requires $O(\log n)$ steps. Thus q is obtained in $O(\log n)$ steps. Next kink edges are changed along $lp(p')$. This requires computation of q' at each node on $lp(p')$. q' is maintained in constant time as we traverse nodes up $lp(p')$. Since there are $O(\log n)$ nodes on $lp(p')$, all the kink edge changes require $O(\log n)$ steps. □

We next prove the correctness of the insertion procedure.

LEMMA 4. *The insertion procedure maintains semistaircases, satisfying Property Semi, at each node of the maxima balanced tree.*

Proof. The proof is by induction. It is trivially true when there is only one point. Thus assume that there are n points and the data structure contains lists which are semistaircase structures.

On an insertion, let $P(p)$ be the path from the leaf containing the inserted point p to the root. Let c_p be the node at which the point p is located within a semistaircase structure. The semistaircase structures change at c_p and above and below c_p.

First, consider nodes below c_p. We prove the claim in the lemma by induction on the distance from the leaf node in $P(p)$. At a node v, $M(v)$ changes in two ways. Either $r = Top(M(right(v)))$ is located onto p or $M(v) = M(right(v))$. Since points in $M(right(v))$ are maximal in $ST(right(v))$ and since points in $M(left(v))$ with y-coordinates greater than p are maximal in $ST(v)$, points in $M(v)$ are maximal in $ST(v)$. At c_p, p is correctly located in $M(left((c_p)))$ by the procedure $RELOCATE$, as shown in Lemma 3.

Finally, consider a node, v, on $P(p)$ above c_p. The semistaircase structures at these nodes are updated when $Top(M(right(v))), v \in P(p)$ is located onto p. Again, it can be shown, by induction, that $M(v)$ contains only maximal points in $ST(v)$. □

Next consider deletions.

LEMMA 5. *The deletion procedure maintains semistaircases, satisfying Property Semi, at each node of the maxima balanced tree.*

Proof. Let p be the deleted point. Along the deletion path $P(p)$, at most two locations are done using procedure $RELOCATE$. Changes during this procedure correctly maintain the semistaircase structure as shown in Lemma 3. Let q be the point obtained on relocation and let c_p be the node at which a relocation takes place. We next consider changes along path $P(p)$ and show that these changes preserve semistaircase structures. We use induction on the distance from the leaf containing p. Consider any node $v, v \in P(p)$, which is below c_p. The semistaircase structure, $M(v)$, is changed in two ways. Either it is replaced by $M(right(v))$ or a merge is performed. This is done by either constructing a kink edge which connects $r = Top(M(right(v)))$ to a node $q', y(q') > y(r), q' \in M(left(v))$ or locating r onto q. Note that q' may be $Top(M(left(v)))$. By induction $M(left(v))$ and $M(right(v))$ are semistaircase structures. Thus $M(v)$ also comprises maximal points after the merger. Nodes above c_p on $P(p)$ are not affected during the deletion procedure. □

The arguments above, proving correctness under insertions and deletions, extend to proving that changes done during rebalancing, when relocations are performed, also correctly maintain semistaircases satisfying Property Semi.

To report the maximal elements of S, kinks in the semistaircase structure at the root of the tree MT are important since these kinks have to be removed to obtain the actual output. These kinks can be removed at the time of reporting along with other kinks that are discovered when points are located to remove kinks. This is done by the procedure $RELOCATE$ as in the case of insertions. Note that once the set of maxima is reported, no kinks remain in the staircase structure at the root. We thus define *chng* to be the number of changes in the semistaircase structure to be output as compared with the last staircase structure reported. The changes are kinks and new points added to the semistaircase structure representing the set of maxima. The time for output is now $O(m + chng. \log n)$. Note that *chng* can be $O(m)$. Details of the reporting will be discussed in the next section.

The following lemma follows from the discussion above and the insertion and deletion procedures.

LEMMA 6. *The structural changes carried out to maintain the semistaircase structures at each node of the maxima balanced tree under an insertion or deletion require $O(\log n)$ operations. Moreover, $O(\log n)$ operations are required for every change to be reported in the staircase representing the maximal set.*

4.6. Implementing the scheme in $O(n)$space.

In this section we show how to implement the scheme for insertion and deletion in $O(n)$ space. This is accomplished by maintaining $O(1)$ information at the nodes of the balanced binary search tree, MT.

First, at each node v are maintained left and right pointers, $left(v)$ and $right(v)$, to the children of the node, say u and w. We also maintain two variables $TOP(v)$ and $INTER(v)$ which contain points. $TOP(v)$ is the topmost point in the semistaircase structure at v (this point having been referred to by $Top(M(v))$ before). And $INTER(v)$ is the point at which $Top(M(w))$ is located in the semistaircase structure at u, i.e., $INTER(v)$ is the point in the staircase structure at v just above the point $TOP(w)$, in the staircase structure at v, if such a point exists. Otherwise $INTER(v)$ is $TOP(v)$. Note that $Top(M(w))$ may be attached to $INTER(v)$ by a kink edge. This can be detected by maintaining at each node a Boolean variable called $MARK$-$KINK$. In the description below the value of $INTER(v)$ for nodes where a kink line (p, q) is drawn from p to q is assumed to $y(q)$, the y-coordinate of the upper point of the kink line. To list the points more efficiently we need another variable $NextLeft$ which we detail below.

We first describe the listing procedure on the data structure without using $NextLeft$. We then show how to make the procedures more efficient using $NextLeft$.

4.7. Listing the maxima.

To list all the maxima we proceed as follows: We start at the root of the tree MT. At the root node TOP gives the leftmost point in the staircase structure. In general, to obtain points contributed by the left subtree at a node v in MT, we test if the topmost point of the right subtree, say w, is located strictly below the topmost point of the left subtree, say u. If so, then we list recursively those maximal elements in the subtree at u not yet reported and with y-coordinate greater than the y-coordinate of $TOP(w)$. We then list the maximal elements in the subtree rooted at w recursively.

This procedure is formalized as follows:

```
    Procedure $LIST1(v, stop)$.
(This procedure lists the maxima in the subtree at $v$ with
$y$-coordinate greater than or equal to $stop$. )
Begin
    If $y(TOP(v)) \geq stop$, then
    begin
        If $y(TOP(v)) = stop$ or $v$ is a leaf node, then
            output $TOP(v)$
        else
        begin
            if $y(TOP(v)) > y(TOP(right(v)))$, then
              begin
                $LIST1(left(v), \max(stop, y(TOP(right(v)))))$;
                $LIST1(right(v), stop)$
              end
            else $LIST1(right(v), stop)$
        end
    end
end.
```

The procedure $LIST1$ above is invoked at the root with *stop* being the y-coordinate of the point with the maximum x-coordinate.

LEMMA 7. *$LIST1$ correctly outputs the set of maxima of size m stored in tree MT in $O(m \log n)$ steps.*

Proof. At node v the procedure correctly outputs, in order of decreasing y-coordinate, all points in the staircase at v such that if p is to be reported, then $y(p) \geq stop$. It does so by first comparing $y(TOP(v))$ with *stop*. If $y(TOP(v)) \geq stop$, then there exist points to be reported in the staircase structure. There are two cases: If either $y(TOP(v)) = stop$ or v is a leaf node, then only the topmost point is to be reported. On the other hand, if $y(TOP(v)) > stop$, then the points to be reported are to be found in the left and right subtrees of v. Points in $M(v)$ are to be found in the left subtree iff $y(TOP(v)) > y(TOP(w))$, where $w = right(v)$. These points are output by the recursive call to the left child. Each point, p, which is to be output from the left subtree, lies on the staircase structure at v if $y(p) \geq y(TOP(right(v)))$ and $y(p) \geq stop$. Thus the recursive call to the left has $\max(stop, y(TOP(right(v))))$ as the new *stop* parameter. The remaining points are to be output from the right subtree. The parameter *stop* for the call to the right subtree is the same as the parameter *stop* at the current node, v, since all points in the staircase at the right subtree are in the staircase at v.

We next estimate the time required by $LIST1$. First note that a recursive call is made only when at least one point is to be reported. During a call, either a maximal point is reported or another recursive call is made. Furthermore, at each recursive call the procedure processes points at a node which is closer to a leaf node. In $O(\log n)$ steps a node will be reached where a point will be reported. The procedure thus requires time proportional to $O(m \log n)$, where m is the number of maximal points. ∎

To list the maximal points in linear time we make the following modification to the data structure: At each leaf node, say v, where point p is stored, we store a pointer called $NextLeft(p)$. For addressing $NextLeft$, v and p are used synonymously. $NextLeft(p)$ points to the node in the tree at which p is first located within a staircase structure. $NextLeft(p)$ is null if p is not located within any staircase structure.

Using these pointers we can recursively list the maxima as described in $LIST2(MT)$. In this procedure, $RELOCATE$ removes kinks in the staircase structure at the root node as well as other kinks along the location path in the tree MT. The time for listing the maxima is linear in addition to the time for removal of kinks.

```
    **Procedure** $LIST2(MT)$.
(This procedure lists the maxima in a tree $MT$.)
Begin
    Let $p$ be the point with maximum $x$-coordinate in the set of points in $MT$;
    $v = NextLeft(p)$;
    output($p$);
     while $v \neq$ null do
          begin
            if ($M(right(v))$ is merged with $M(left(v))$ by a kink edge), then
              RELOCATE($right(v), left(v)$) and update $NextLeft(q)$ for all $q$
              whose location is updated during the process;
            output($INTER(v)$);
            $v := NextLeft(INTER(v))$;
          end;
end.
```

Note that a kink is removed by locating $TOP(right(v))$. This requires $O(\log n)$ steps for every kink removal.

We next prove the correctness of the algorithm $LIST2(MT)$. We need the following property.

LEMMA 8. **Property Next:** *If a point, p, is maximal and p is not located by a kink edge then $INTER(NextLeft(p))$ is also maximal.*

Proof. Since p is maximal there does not exist any point with greater x and greater y-coordinate. Consider the location of p inside a staircase structure in MT. The node at which this location is performed is given by $w = NextLeft(p)$. $q = INTER(w)$ gives the point just above p in the staircase structure, $M(w)$. This point is maximal with respect to the set of points in the subtree rooted at w since it is on the staircase structure at w. Moreover, at tree nodes which are ancestors of w, there does not exist any point in the staircase structures at these nodes which has greater x-coordinate and greater y-coordinate. This is because such a point would also dominate p which is maximal and stored in the subtree rooted at w. Thus q is maximal. ☐

LEMMA 9. *LIST2(MT) correctly reports the set of maxima of S stored in MT.*

Proof. The proof is by induction on the number of maximal elements. For the base case note that the first element reported is in the set of maxima. To prove the lemma consider the report of the kth element of the set of maxima when $k-1$ elements have been reported. The kth point is reported after locating $Top(right(v))$, if necessary, where $v = NextLeft(q)$. After the location q is not located by a kink edge at v, Property Next is applicable to q and thus $INTER(v)$ is maximal. ☐

The complexity of the procedure is $O(m + chng. \log n)$ where m is the number of maxima and *chng* is bounded by the number of changes in the semistaircase structure representing the set of maxima since the last report of the maxima. Note that, at that report, no kink edges were left in the staircase structure.

It will be shown later that the pointers $NextLeft(p)$ can be maintained in $O(\log n)$ time per insertion and deletion. $NextLeft(p)$ changes only at locations of points within staircase structures and only one location is required during an insertion, deletion, or a rotation.

4.8. Searching for dominance. In this section we show how to determine if a given point, p, is dominated by the set of maxima or not. We do so by a search for the y-coordinate of the given point in the set of points with greater x-coordinates. Let P be the path from the root to the leaf where point p is stored. Let R be the set of roots of the subtrees to the right of path P. It suffices to check if $y(p) \geq y(TOP(v))$ for all $v \in R$. If it is, then the point is not dominated; otherwise it is dominated.

4.9. Insertions and deletions. To bound the time for insertions and deletions in the $O(n)$ space data structure we note that the crucial step while performing insertions and deletions is that of locating the topmost point of a semistaircase structure in another semistaircase structure. An $O(\log n)$ solution to this step has been presented above.

Moreover, the $O(1)$ rotations performed at every step can also be done in $O(\log n)$ steps since they require $O(1)$ mergers, each requiring $O(\log n)$ steps. The variables TOP and $INTER$ stored at the nodes at which relocations are performed can also be modified in $O(\log n)$ steps. Note that the value of $INTER$ is obtained by using $LOCATE$. The value of TOP is immediately available.

We would also like to update the value of $NextLeft(p)$ during the insertion, deletion, and rotation process. We describe the updated steps below.

First consider insertions. Let $P(p)$ be the insertion path in MT for point p. Let c_p be the node at which p is located within a staircase structure. $NextLeft(p)$ is defined at this point. Below that node, p is at the top of the staircase structures. For nodes below c_p on $P(p)$, $NextLeft(q)$ changes for points q which are located onto the stairstep defined by p. For each such point, say q, $NextLeft(q)$ is updated in $O(1)$ time. And $NextLeft(q)$ is made null for each point, say q, which was replaced by p as top of the staircase structures since these points are not located onto any staircase structures. Also let c_u be the node at which p is removed from the maxima set by a point that dominates it. Above this node the point p is no longer on the staircase and will not affect the value of $NextLeft(q)$ for any point q located at a node above c_u in the tree. For nodes between c_p and c_u, both included, the value of $NextLeft(q)$ changes and is updated when the point q is located onto the stairstep defined by p. Note that during the location of p a number of kink edges would have their top points changed. If a kink edge (r, s) is constructed from point r to point s, $NextLeft(r)$ is set to s. At most $O(\log n)$ such changes are made.

The modification of the $NextLeft$ pointers thus requires $O(\log n)$ steps since at most $O(\log n)$ values are changed along $P(p)$. The entire insertion procedure thus requires $O(\log n)$ steps.

A similar procedure to update $NextLeft$ values is used when a point p is deleted. Consider the path $P(p)$. During a deletion there are at most two nodes at which a relocation of a point, say q, is performed. Let the node at which a location takes place be called c_q. $NextLeft(q)$ is thus to be updated. Below node c_q, merges are performed using kink edges. Let r be the topmost point of the staircase structure which is to be merged by a kink edge. $NextLeft(r)$ is updated in $O(1)$ time. Above node c_q no changes in the location of any point occur. Note that during the relocation at node c_q, $O(\log n)$ steps are required for changes of $NextLeft$ values for points connected by new kink edges formed during the relocation procedure as already described in the case of insertions.

Finally, consider rebalancing operations. We consider single rotations only. The procedure is similar for double rotations. Let v be the node at which a rebalancing operation is performed. Consider the nodes affected by the rotation. These include v, the roots of the subtrees involved in the rotation, and the additional node created. At each of these nodes a relocation is performed and $NextLeft$ values are changed as described above. And no point, located at nodes not involved in the rotation, has its $NextLeft$ value affected. The updates require $O(\log n)$ operations.

We can thus state the following theorem.

THEOREM 2. *The maxima of a set of points S, of size n, can be maintained in a data structure which uses $O(n)$ space and which requires $O(\log n)$ operations per update when $O(n)$ points are deleted or inserted. The complexity of reporting the maxima is $O(m + chng. \log n)$ when there are m maximal points reported and chng changes in the semistaircase structure storing the maxima since the last report.*

5. An improved amortized bound. In this section we show how the maxima can be reported in linear amortized time.

To improve the reporting bound we note that the $O(\log n)$ factor in the reporting time bound essentially arises because a kink edge (p, q) repeatedly reoccurs in the changes in the semistaircase representing the maximal set, i.e., it occurs, is removed from the maxima semistaircase structure by, say, an insertion, and then reoccurs in the semistaircase structure representing the maxima after a deletion. Thus its location in the staircase is to be determined at each report. To remedy this we keep the location of p and update it whenever necessary.

Let p be a point and let v be a node in the tree where the point p first occurs within a semistaircase, i.e., is not at the top of the semistaircase at v. At this node the point has to be located within a staircase structure. This location, $LOC(p)$, is important since it may need to be redetermined during deletions. So we keep this location in a variable $SLOC(p)$. However, this location is not updated at every insertion or deletion since only the top points in any staircase are manipulated. Thus when this point is to be output, the value of $SLOC(p)$ may need to be updated. The update is necessary when the stairstep onto which the point is located is changed. If the location requires a binary search, the time for this search is charged to a point, say q, that is inserted or deleted and changes the location of p. However point q may already have been charged. This happens when a set of points are located on the stairstep with q as its corner point. Fortunately, this set of points can be ordered by domination and there is one point in this set which dominates the others. This point must be deleted before other points become maximal. A location onto a stairstep whose corner point is already charged by the location of another point, r, is charged to the deletion of this point r, an event which must occur for the current location to be computed.

We prove this formally below. We need the following definitions.

We let $P(p)$ be the path from the leaf containing p to the root and $LS(p)$ be the set of points such that for each point q, $x(q) < x(p), q \in S$.

$LOC(p)$ is the maximal point in $LS(p)$ with y-coordinate just greater than $y(p)$.

$CLOC(p) = \{q|$ such that $SLOC(q) = LOC(q)$ prior to the insertion or deletion of point p but not after $\}$. This is the set of points for which the $LOC(p)$ has changed due to insertion and deletion of p.

$MCLOC(p) = \{q|y(q) = \max_{q' \in CLOC(p)}\{y(q')\}\}$, $OTHER(p) = \{r|r \& p \in CLOC(q)$ and r is dominated by $p\}$. Note that $OTHER(p)$ is the set of points which are dominated by p and required to be located onto $R(q)$, when q is a deleted point and p is located onto q. $R(q)$ is the semistaircase structure that replaces the deleted point q at nodes on $P(q)$, the path from the leaf containing q to the root. Alternatively, $OTHER(p)$ is the set of points dominated by p which are to be located onto q at the insertion of q.

The dynamic maintenance scheme, described in the section before, is now modified so that for every point we maintain $LOC(p)$ in the variable $SLOC(p)$. This value may be precomputed initially and may be updated when the point is to be output as a maximal point. To reduce the time for relocations consider the following additional steps in the insertion and deletion procedures where certain points are marked.

Insertions. On an insertion the following additional step is done:

1. Mark $MCLOC(p)$.

Deletions. On a deletion the additional steps are the following.

1. Mark $MCLOC(p)$.
2. If $p = MCLOC(q)$, mark updated $MCLOC(q)$ obtained after deleting p.
3. Relocate maximal point op in $OTHER(p)$. Also let $OTHER(op) = OTHER(p) - \{op\}$.

Reporting. Finally, on the report, do the following step.

1. Locate all marked nodes encountered and unmark them while reporting the maximal points.

No marks are changed during rotations. The correctness of the reporting procedure is shown next.

LEMMA 10. *Let q be a maximal point which is unmarked. Then $LOC(q)$ is correctly maintained.*

Proof. The proof is by induction on the number of insertions and deletions. For the base case, the lemma is trivially true. Assume that it is true when k insertions or deletions have been made. Consider the $k+1$st operation.

Consider an insertion. Suppose $LOC(q)$ is modified due to insertion of p'. Then $q' = MCLOC(p)$ is marked. If $q \neq q'$, then q' dominates q and thus q is not maximal.

Next, consider deletions. Suppose a point q is affected. This occurs in two ways:

(i) q becomes maximal, and
(ii) $LOC(q)$ is changed.

Consider the following cases: q is already in some set $CLOC$ or not in any such set. In the first case, if $q \in OTHER(p)$, then q may become maximal and is marked. Otherwise it is in $OTHER(op)$, where op is the maximal point in $OTHER(p)$. Alternatively, $q \in CLOC(p)$ at the current deletion. Then either q is marked or some other point $q' = MCLOC(p)$ is marked. If q' dominates q, then q is not maximal but q is in $OTHER(q')$. Otherwise $x(q) > x(q')$ and thus $LOC(q) \neq p$ since q would be located onto either q' or some other point. No other point is affected by p. □

Since marking points requires maintaining sets $CLOC$ and $OTHER$, we omit the marking in our actual procedure. It suffices to simply determine if $SLOC(p) = LOC(p)$, i.e., if $LOC(p)$ is correctly maintained for a maximal point p. If not, then p is located. Let v be a node in the tree where a point p is located or to be located. To determine if $LOC(p)$ is correctly maintained we first check if $y(p) > y(TOP(v))$. If not, then $Top(v)$ is in the maximal set along with other points in $M(v)$. The points with y-coordinate greater than $y(LOC(p))$ are determined by traversing $M(v)$ in decreasing order of y-coordinate values, starting from $Top(v)$, until the last node, $q, q \in M(v)$, such that $y(q) > y(p)$ is reached. At this stage p is required to be located if either a kink edge connects p to q or if $SLOC(p)$ is not q. Let $L(M(v))$ be the list of points traversed, in order of decreasing y-coordinate values, until q is reached. $L(M(v))$ satisfies the following property.

Lemma 11. *All points in $L(M(v))$ are maximal. Furthermore, if r and s are adjacent points in $L(M(v))$ with $y(r) > y(s)$, then $LOC(s)$ is either r or is a point t such that $y(r) < y(t) < y(s)$.*

Proof. We first show that since p is maximal, all points in $M(v)$ with greater y-coordinate are also maximal. The proof is by contradiction. If not then there exists a point, say p', at a node above v in MT such that $y(p') > y(z)$ and $x(p') > x(z), z \in L(M(v))$. But p' also dominates p.

To show the second part of the lemma, consider r and s which are adjacent points in $L(M(v))$. s is connected to r by either a stairstep or by a kink edge. In the first case, $LOC(s)$ is r. In the second case, s is located in the staircase structure in between r and s. $LOC(s)$ is to be determined at $NextLeft(s)$ in the subtree rooted at $left(v)$, and thus $LOC(s)$ has y-coordinate less than that of r. $\square$

The following lemma justifies recomputation of $LOC(p)$.

Lemma 12. *If (p, q) is a kink edge or $SLOC(p) \neq q$, then p is marked.*

Proof. Suppose (p, q) is a kink edge. Then this kink edge must be due to the deletion of a point p' with x-coordinate less than $x(p)$. p is in $CLOC(p'')$ where p'' is such that $x(p'') < x(p)$. If p is not $MCLOC(p'')$, then either there exists a point that dominates p and thus p is not maximal or $p \notin CLOC(p'')$. Thus p is $MCLOC(p'')$ and is marked.

Next, consider the case when $SLOC(p) \neq q$. Since p must be located onto the stairstep formed by q, $p \in CLOC(q)$ when q is inserted or $p \in CLOC(q')$ when q' is deleted and since p is maximal, p is marked. $\square$

To traverse $M(v)$ efficiently we keep with every point p the point next to p when $M(v)$ is ordered by decreasing value of y-coordinates. This point, say $Nextdown(p)$, is located onto the stairstep formed by p in $M(u)$, $u \in P(p)$ where u is the highest node in $P(p)$ such that a point is located onto the stairstep formed by p at the nodes of the tree. Note that a point joined by a kink edge to p is also said to be located onto the stairstep formed by p. $Nextdown(p)$ is determined by keeping the following variable, $INTERLOC$, with every point p.

Definition 5. *$INTERLOC(p)$ is the node, say u, in the tree, MT, where $q = Nextdown(p)$ is located onto the stairstep formed by p in $M(u)$.*

$INTERLOC(p)$ can be computed in $O(\log n)$ steps. The sequence of points $M(v)$, starting from $TOP(v)$, can be listed using $INTERLOC$ values in linear time. Moreover $INTERLOC(p)$ can be easily maintained during insertions and deletions and rotations. The maintenance is similar to that of $NextLeft$ values.

Lemma 13. *$INTERLOC(p), p \in S$ can be maintained in $O(\log n)$ steps during insertions, deletions, and rotations.*

Proof. Consider an insertion of a point q. At most one location is performed during an insertion. Suppose q is located onto a point p. Only $INTERLOC(p)$ needs to be updated. This can be done by comparing with the current value of $INTERLOC(p)$. Moreover, during relocations kink edges are changed. Each $INTERLOC$ value affected during a relocation, say $INTERLOC(p)$, is updated by maintaining p and the highest node, denoted by $H(p)$, at which a kink edge is joined to p as the relocation path is traversed from a leaf to the root. At each step along this path only one point, q', and $H(q')$ need be maintained. This follows from the way that the relocation procedure, $RELOCATE$, modifies kink edges using the point q'. The procedure changes q' along the relocation path such that $y(q')$ increases. Let r and s be two points such that $q' = s$ and subsequently $q' = r$. Since $y(s) < y(r)$ no kink edge will be joined onto s once q' becomes r. Thus only $INTERLOC(p)$, where p is q' currently, needs to be modified as $RELOCATE$ processes nodes on the relocation path.

During a deletion, two relocations are performed. $INTERLOC$ values affected are updated again in $O(\log n)$ steps. Furthermore, kink edges are introduced. Each $INTERLOC$ value affected by kink edges can be updated in constant time since this value can be maintained as we traverse up the path where a deletion or relocation takes place. The maintenance detail is similar to that described for insertions.

Finally, during rotations, a constant number of relocations are performed. $INTERLOC$ values can again be updated as in the case of insertions and deletions in $O(\log n)$ steps. □

We thus have the following lemma.

LEMMA 14. *Let LS be the subset of maximal points such that $SLOC(p), p \in LS$, needs to be updated. Given $INTERLOC(p), p \in S$, LS can be determined in time linear in the output size.*

Proof. Consider the linear search performed at a node v using $INTERLOC$ values to determine if $LOC(p)$ is to be updated. By Lemma 11 all the points scanned are in the output, and by the discussion above $L(M(v))$ can be determined in linear time. Moreover, as shown in Lemma 11, these points need not be scanned more than once since this scan allows us to determine if LOC updates of points in $L(M(v))$ is either already determined or is to be determined by a location. □

Note that while updating $LOC(p)$, $NextLeft$ values at leaf nodes also need to be updated. This is done by a procedure similar to that which updates $NextLeft$ values during locations while inserting or deleting and requires $O(\log n)$ steps. We can thus state the following result.

THEOREM 3. *The set of maxima of a set of points in the plane can be maintained in $O(n \log n + m \log n + r)$ operations when there are n insertions, m deletions, and r is the number of points reported. Furthermore, each update requires $O(\log n)$ steps in the worst case.*

Proof. We need show the correctness of only the reporting procedure. By Lemma 12, during a report a point is located only if it is marked. Furthermore, a point, p such that $LOC(p)$ is not correctly maintained is always marked (Lemma 10).

We next analyze the time complexity. The time complexity for insertion, delete, and report operations without the maintenance of $LOC(p)$ has been shown before in Theorem 2. We consider the effect of marking of nodes ignoring the time required to compute sets $CLOC$ and $OTHER$ since our procedure avoids these computations. The analysis uses the following potential function:

$$PF = \log n \times (\text{ No. of marked points}).$$

When we use this potential function, the amortized time for an insertion is $O(\log n)$ since at most one location is done and one point is marked. The amortized time for a deletion operations is also $O(\log n)$ since apart from a constant number of locations at most three marked nodes are affected. During the reporting of the maxima, marked points are located and the points are unmarked. Since the number of locations is equal to the number of marked points decreased by becoming unmarked, the amortized time is $O(r)$ where there are r points reported.

In the actual procedure described above the marking is not explicitly done, but a marked point p is detected by checking $SLOC(p)$ or the existence of a kink edge onto which p is located. By Lemma 12 this suffices to detect marked points. Moreover, by Lemma 14 these points can be detected in $O(r)$ time since Lemma 13 shows that $INTERLOC$ values can be maintained in $O(\log n)$ steps. The theorem follows. □

6. Maintaining maximas under insertions in d-dimension. In this section we show how to extend the scheme of maintenance of maximas to higher dimensions. We only consider insertions.

We use the fact that the problem can be solved in two dimensions in $O(\log n)$ time per update. To this we add the range restriction technique [WL]. The ranges are added to all dimensions other than the last two where the solution described above is used.

The details of the data structure are as follows: The primary structure is a balanced tree T_d representing the dth dimension. It stores at its nodes secondary structures which represent the maximal points of the subset of points at that node. At each secondary structure it suffices to maintain the set of maxima in $d-1$ dimensions of a set of points. This set is obtained from the projection of the subset of points in the subtree rooted at the node onto a d-dimensional hyper-plane $x_d = xmin$ where $xmin$ is the minimum d-coordinate of the subset of points. Thus each node u in the tree is also associated with a hyperplane $H(u)$ onto which maximal points in the subtree are projected. The secondary structure is denoted by $T_{d-1}(u)$. To compute the maxima at a node the maxima obtained from the left child are filtered from those at its right child. The filtering operation is well known [KLP].

We next define the filtering operation when a point is added into the data structure. Let P be the path from the leaf where the point is added to the root in T_d. Suppose the point added is a maximal point in a subset of points present at node u in T_d. Let w be the node such that v is a child, and let u be its other child. First consider the case when v is a right child. Then the point p eliminates maxima from the set of maxima at u. This filtering operation is performed recursively as follows: Let p' be the projection of p on the hyperplane $H(u)$ at u. We determine the points that are eliminated in the secondary structure at u by p'. This is done by inserting p' into the secondary structure $T_{d-1}(u)$ using a similar procedure. The recursion stops when the secondary structure contains two-dimensional points. At this stage the procedure defined above for maintaining the maxima in two dimensions is used to obtain the set of points eliminated by the introduction of p. The time required for this filtering operation is $O(\log^{d-1} n)$.

We next consider the case when v is a left child of w. In this case it has to be determined whether p is eliminated by the set of maxima at u the other child of w. To do this we recursively evaluate whether p', the projection of p onto $H(u)$, is dominated by the set of maxima in $T_{d-1}(u)$. This is done recursively as follows: Let $P(p')$ be the path from the leaf to the root. For all subtrees along the path containing points with greater $d-1st$ coordinate it is determined whether p' is dominated by the points in the subtree in the remaining $d-2$ dimensions also. The recursion stops at the second dimension when the point is checked to be dominated or not in $O(\log n)$ steps using the staircase structure.

The filtering operation gives a set of points that are to be deleted and inserted from the set of maxima at w. Consider the deletion of a point q from a secondary structure $T_{d-1}(w)$. Again let $P(q)$ be the path from the leaf containing q to the root. The point is recursively removed from each of the secondary data structures at the nodes on the path. The recursion stops when the dimension is 2. At this point the procedure defined before is used to delete the point from the set of two-dimensional maxima in $O(\log n)$ operations. The insertion of a point is done similarly.

The details of the balancing procedures and the changes to the secondary structures are similar to that described in Willard and Leuker [WL] and are omitted here.

We thus achieve changes in the maxima tree structure in $O(\log^{d-2} n)$ time per change after the insert and filter steps which require $O(\log^{d-1} n)$ operations.

Conclusions and acknowledgments. We have shown efficient schemes for dynamically maintaining the set of maxima of a set of points in two dimensions. We do so by maintaining approximate information. Other applications of this technique would be interesting. I would like to thank A. Dutta and S. Sen for helpful discussions.

REFERENCES

[GR] G. N. Frederickson and S. Rodger, *A new approach to the dynamic maintenance of maximal points in a plane*, Discrete Comput. Geom., 5 (1990), pp. 365–374.

[HS] J. Hershberger and S. Suri, *Offline maintenance of planar configurations*, in Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, SIAM, Philadelphia, 1991, pp. 32–41.

[J] R. Janardan, *On the dynamic maintenance of maximal points in the plane*, Inform. Process. Lett., 40 (1991), pp. 59–64.

[KLP] H. T. Kung, F. Luccio, and F. P. Preparata, *On finding the maxima of a set of vectors*, J. ACM, 22 (1975), pp. 469–476.

[OL] M. H. Overmars and J. L. Van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. System Sci., 23 (1981), pp. 166–204.

[FP] F. P. Preparata and M. I. Shamos, *Computational Geometry—An Introduction*, Springer-Verlag, New York, 1985.

[T] R. E. Tarjan, *Updating a balanced search tree in $O(1)$ rotations*, Inform. Process. Lett., 16 (1983), pp. 253–257.

[WL] D. E. Willard and G. S. Lueker, *Adding range restriction capability to dynamic data structures*, J. ACM, 32 (1985), pp. 597–617.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 1878–1888

THE COMPLEXITY OF THE $A\ B\ C$ PROBLEM[*]

JIN-YI CAI[†], RICHARD J. LIPTON[‡], AND YECHEZKEL ZALCSTEIN[§]

Abstract. We present a deterministic polynomial-time algorithm for the $A\,B\,C$ problem, which is the membership problem for 2-generated commutative linear semigroups over an algebraic number field. We also obtain a polynomial-time algorithm for the (easier) membership problem for 2-generated abelian linear groups. Furthermore, we provide a polynomial-sized encoding for the set of all solutions.

Key words. polynomial-time algorithm, membership problem, semigroup, commutative, lattice

AMS subject classifications. 20M99, 68Q25, 68W40

PII. S0097539794276853

1. Introduction.

Most algorithmic questions about infinite groups and semigroups given by presentations (generators and relations) are known to be undecidable [11], [38], [1], [40]. However, the most useful and interesting representation of groups is by matrices over a field. Most groups occurring in physics and many finite simple groups arise as groups of matrices. Nonetheless, it should be noted that many interesting groups do not have faithful matrix representations. An interesting class of groups that cannot be represented by matrices is the uncountable class of infinite Burnside groups introduced by Grigorchuk [20] and whose computational properties were studied in [16]. Rabin has stated without proof in [41] that if the groups are represented by matrices over a field then the word problem is decidable. This result was improved to log space complexity by Lipton and Zalcstein [33] for fields of characteristic zero and by Simon [42] for fields of positive characteristic.

The word problem is a "checking" problem. The corresponding "search" problem is the membership problem: Precisely, given a finite number of elements $a_1, \ldots, a_k$ (given as matrices or permutations) of a group G and a target element g, does g belong to the group generated by $a_1, \ldots, a_k$?

In this generality the problem is undecidable [37], even for groups of 4-by-4 matrices. Thus work on this problem has focused on *finite* groups. Babai and Szemeredi [5] have shown that, for finite groups of matrices over a finite field the problem is in NP. This problem has also motivated the introduction of the complexity class AM [2]. Since, as has been observed in [5], the membership problem for finite matrix groups over finite fields is closely related in complexity to that of the discrete logarithm problem, further progress seems unlikely and most research has concentrated

[*]Received by the editors November 8, 1994; accepted for publication (in revised form) July 13, 1998; published electronically April 4, 2000. A preliminary version of this paper, "Complexity of the membership problem for 2-generated communatative semigroups of rational matrices," was presented at the 1994 Annual Symposium on Foundations of Computer Science (FOCS), pp. 135–142.

http://www.siam.org/journals/sicomp/29-6/27685.html

[†]Department of Computer Science, SUNY at Buffalo, Buffalo, NY 14260 (cai@cs.buffalo.edu). Research supported in part by NSF grants CCR 9057486 and CCR 9319093 and an Alfred P. Sloan Fellowship.

[‡]Department of Computer Science, Princeton University, Princeton, NJ 08544 (rjl@cs.princeton.edu). Research supported in part by NSF grant CCR-9304718

[§]Division of Computer and Computation Research, National Science Foundation, 4201 Wilson Boulevard, Arlington, VA 22230 (zzalcste@nsf.gov). The statements contributed by this author to the paper represent his personal opinions and are not necessarily those of the National Science Foundation.

on groups represented by *permutations* rather than matrices. Over the years, this research has culminated in efficient sequential and parallel algorithms [3, 4, 15, 22]. Progress on the matrix representation case is very recent. Luks [35] has presented a polynomial-time membership test for solvable matrix groups over "good" finite fields (where the discrete logarithm problem can be bypassed). Beals and Babai [6] have given a polynomial-time *Las Vegas* membership test for arbitrary finite matrix groups over an algebraic number field. For semigroups the problem is even harder. It is known to be undecidable even if the target element g is the zero matrix and the matrices are 3 by 3 [39] (for a related unsolvable problem for semigroups of triangular 3-by-3 matrices see [23]). For finite semigroups the problem is PSPACE-hard [25], even if the semigroups are inverse (this follows from [9, Theorem 5.4]) and it is NP-hard for finite *commutative* semigroups [8].

In this paper, we extend previous algorithmic results in two directions. First, we allow infinite systems, where even decidability is not clear, and second, we consider as semigroups as well. On the other hand, since the general problem is undecidable, some restrictions are needed. In 1980, Kannan and Lipton [28] solved the following *orbit problem*, which is the membership problem for a *cyclic* semigroup, by giving a polynomial-time algorithm:

Given two commuting matrices A and B over the rational numbers,
does there exist a nonnegative integer i, such that $A^i = B$?

The following *generalized orbit problem*, or *the A B C problem*, has remained open since 1980:

Given commuting matrices A, B, and C over the rational numbers,
does there exist nonnegative integers i and j, such that $A^iB^j = C$?

In this paper, we resolve the A B C problem by giving a polynomial-time algorithm which not only decides the solvability of a given instance A, B and C, but also finds all solutions by showing that the set of solutions is a (possibly empty) "affine lattice" and producing a polynomial-sized basis for that lattice.

We also obtain the corresponding result for the group case. We would like to point out that the semigroup problem is harder, even when the generating matrices are invertible, since we do not allow the use of the inverse operation. The results are new, even in the group case. Prior to this work, the best result for the problem in the group case was that it is decidable, a result by Kopytov [24]. The proof in [24], which constructs "yes" and "no" lists, does not give any complexity bounds. Nothing was previously known for the semigroup case.

The techniques presented here can be generalized and modified to solve the membership problem for the case of $k > 2$ generators, where k is fixed [7]. However, we should point out that, for semigroups, the number of generators must be fixed; otherwise, as noted above, the problem is already NP-hard in the finite case.

We will explain briefly why the problem is challenging. Let us assume first (this is seemingly the easiest case) that the matrices A, B, and C are all diagonalizable. Since they commute they must be simultaneously diagonalizable. In this case the problem is reduced to n instances of the following problem (n is the size of the matrices): Given algebraic numbers α, β, and γ, are there nonnegative integers i, j such that $\alpha^i\beta^j = \gamma$? Or more generally, for a fixed k, given algebraic numbers $\alpha_1, \ldots, \alpha_k$, and η, are there nonnegative integers $i_1, \ldots, i_k$ such that

$$\alpha_1^{i_1} \ldots \alpha_k^{i_k} = \eta \tag{1}$$

holds? In case some of the α's are not units, one can use Kummer's unique factorization of ideals and apply an argument based on the exponential growth of the norms $[\mathbf{N}(\alpha_i)]^i$. However, the hard case is when all the α's are units. The theorem of Blanksby and Montgomery [10] used in [28] is inadequate for the case $k \geq 2$. Another natural attempt is to use Dirichlet's unit theorem to decompose the α's in terms of fundamental units. However, the complexity of computing such decompositions has not, to our knowledge, been analyzed and it may not be computationally tractable. Furthermore, Dirichlet's theorem handles only the group case. Finally, in reducing the general case to the diagonalizable case, Dirichlet's theorem is not enough. We need a description of all solutions of (1). Such a description, for the special case that $\eta = 1$, has been obtained only recently by Masser [36]. A more recent result, by Ge [19] gives a polynomial-time algorithm for computing a basis for all solutions of (1), for $\eta = 1$.

The proof builds upon Ge's result and the observation that if a matrix A is given in Jordan normal form (JNF), then there are closed form formulas for the powers of A. A recent result [12], developed simultaneously with this paper, gives a polynomial-time algorithm for computing JNF. This observation, together with Theorem 3.2 which handles the scalar case, provides a quick proof of the Kannan–Lipton theorem. It should be pointed out that [12] uses the L^3 algorithm [31], which had not yet been discovered at the time that [28] was written. Generalizing to the case of two generators, if both generators are diagonalizable, the problem is reduced to the scalar case. If at least one of the generator matrices is not diagonalizable, it is tempting to believe that, since the matrices commute, they can be brought simultaneously to JNF. If this were true, an extension of the argument in the diagonalizable case could possibly be constructed. However, this is not true. To overcome this difficulty, we introduce a technique which we call *successive restriction.*

Our results are as follows.

THEOREM 1.1. *There is a polynomial-time algorithm for the following problem: Given $k+1$ algebraic numbers $\alpha_1, \ldots, \alpha_k$ and η, are there nonnegative integers $i_1, \ldots, i_k$ such that* (1) *holds? If so, find all solutions (in a polynomial-sized encoding).*

THEOREM 1.2. *There is a polynomial-time algorithm to decide the solvability of the A B C problem; namely, given commuting matrices A, B, and C over an algebraic number field, do there exist nonnegative integers i and j such that $A^iB^j = C$? Further, if there is a solution, the algorithm finds all solutions (in a polynomial-sized encoding).*

THEOREM 1.3. *There is a polynomial-time algorithm to solve the A B C problem in the group case, i.e., where the exponents are allowed to be arbitrary integers.*

It is possible to generalize Theorem 1.3 as follows. A group G satisfies the *permutation* (or *rewriting*) property P_3 iff for any three elements a_1, a_2, a_3 in G, the product $a_1a_2a_3$ equals a product which is a nontrivial permutation of a_1, a_2, a_3. Any abelian group satisfies the property P_3. It is known [14] (see also [17]) that a group satisfies P_3 iff it has at most one nontrivial commutator. Thus in a group satisfying P_3, any product of powers of the generators equals the product $a_1^{i_1} \ldots a_k^{i_k}$, possibly post-multiplied by the unique commutator. Thus membership testing is reduced to the abelian case. Unfortunately, the P_3 semigroups have not been characterized completely (for some partial results see [17]) and the complexity of the membership problem for P_3 semigroups is not known.

The plan for this paper is as follows. In section 2 we will deal with some preliminary issues. Proofs in section 2 are omitted. In section 3 we prove Theorem 3.2, from which Theorem 1.1 follows. In section 4, we first give an illustration of the power of our new techniques in a simple setting by giving a transparent proof of the Kannan–Lipton theorem. Then in section 4 we use the method of successive restriction to complete the solution of the A B C problem.

2. Preliminaries. First, without loss of generality, we may assume that the matrices are over the rationals. This reduction uses the classical matrix representation of a finite algebraic extension over the ground field [21]. Let A, B, and $C \in \mathbf{Q}^{n\times n}$ be $n \times n$ rational matrices. The input size to the problem is n plus the sum of binary lengths of all entries. By an extension of a technique from [28], we may assume that C is given as a polynomial in A and B with rational coefficients. This is certainly a necessary condition, and whether C is such a polynomial can be decided in polynomial time, as it is a question of linear dependence over the rationals [26].

Thus, we are given $A, B \in \mathbf{Q}^{n\times n}$ such that A, B commute, and a polynomial p in A and B. The question is to decide if p can be expressed as a product of nonnegative powers of A and B. In [28], the computation of eigenvalues could be circumvented, using the crucial fact that the ring of polynomials in one variable over a field is principal. Since the ring of polynomials in two variables is not principal, this strategy no longer works. We will compute in various number fields, mostly symbolically; i.e., we will have an irreducible polynomial f (proven so by the L^3-algorithm), and each element in the field will be represented by a polynomial of degree $< \deg(f)$ [31, 29]. This can be done over any number fields as well, not just $\mathbf{Q}$. However, an important issue is that we cannot allow the degree of the extension over $\mathbf{Q}$ to be too large; in general, to stay within polynomial time, we cannot operate in the splitting field of an irreducible polynomial of degree say, n, since the Galois group of this extension field might be too large. It is not known how to compute in polynomial time in an extension field where the Galois group has size $n!$ and in fact it is believed to be impossible [32].

We will need to compute the JNF for either of the matrices A or B. What we will show in the paper is a technique of successive restriction which is of independent interest and is almost as good for our purposes as simultaneous JNF. But before we get to that, let's first ask if we can compute a basis change for the JNF of a matrix A in polynomial time. The difficulty is that if we use the standard textbook algorithm, we will be dealing with the splitting field of χ_A over $\mathbf{Q}$. Thus the standard algorithm will not do. However, there is a polynomial-time algorithm [12] that finds an invertible matrix T, such that $T^{-1}AT = J_A$, the JNF for A. The matrix T has entries from the splitting field of χ_A, but the trick is that in [12] we can find a T where every entry t in fact belongs to a much smaller extension field, which varies with t and can be computed in polynomial time within the smaller field. The matrix J_A is *not* computed from the *product* $T^{-1}AT$; in fact, computing T^{-1} from T, say using Gaussian elimination, involves the splitting field again. However, a different technique is used in [12] to compute J_A, and to compute T^{-1} from T, without ever going to the splitting field, all accomplished in polynomial time.

Next there is the issue of how to apply this basis transformation to another matrix M. The fact is that this cannot be done in polynomial time. The same difficulty with the splitting field gets in the way. *But it can be done for the matrices B and C*, using the fact that they commute with A. It is only in this limited sense that we have obtained this basis transformation computationally. Fortunately, this is sufficient for the solution of the A B C problem.

There is another technical issue concerning conjugates of the same irreducible polynomial. When it is necessary to distinguish one root from another, we will use a "good enough" rational approximation, which gives a polynomial number of bits and uniquely identifies the root on the complex plane. We refer the reader to [34] for the details.

In the following we will speak freely of computing JNF, invariant subspaces, etc., all in polynomial time, without further comment. We will reduce the situation to the case where both A and B have exactly one eigenvalue each, λ and μ, respectively. We view A and B as linear transformations over the field $\mathbf{C}$ of complex numbers. Let $p(.)$ be a polynomial with complex coefficients. Consider a subspace V defined as $\ker(p(A))$ or $\mathrm{Im}(p(A))$. It is easy to see that V is an invariant subspace for both A and B

Now consider the subspace $V_\lambda = \ker(\lambda I_n - A)^{n-1}$, where λ is any eigenvalue of A, i.e., $\lambda \in \mathit{Spec}(A)$. Since A and B commute, V_λ is an invariant subspace for both A and B.

After some preparation, we can arrive at the following situation: we have a decomposition of $\mathbf{C}^n$ as a direct sum of subspaces $V_{\lambda,\mu}$, invariant for both A and B, where λ and μ range over $\mathit{Spec}(A)$ and $\mathit{Spec}(B)$, i.e., all eigenvalues of A and B, such that there is just one eigenvalue λ (and μ, respectively) for A (and B, respectively) on $V_{\lambda,\mu}$:

$$\mathbf{C}^n = \bigoplus_{\lambda \in \mathit{Spec}(A), \mu \in \mathit{Spec}(B)} V_{\lambda,\mu}.$$

Lemma 2.1.
Suppose A, B commute and C is a polynomial in A and B; then for any i, j,

$$A^i B^j = C \Longleftrightarrow (A\,|_{V_{\lambda,\mu}})^i (B\,|_{V_{\lambda,\mu}})^j = C\,|_{V_{\lambda,\mu}},$$

$\forall \lambda \in \mathit{Spec}(A), \mu \in \mathit{Spec}(B)$.

For what follows, we will fix any pair of eigenvalues $\lambda \in \mathit{Spec}(A)$ and $\mu \in \mathit{Spec}(B)$ and consider $A\,|_{V_{\lambda,\mu}}$ and $B\,|_{V_{\lambda,\mu}}$. We will call them A and B, respectively, when no confusion arises.

3. The affine lattice of solutions.

We will need the following theorem of Masser [36].

Theorem 3.1. *Fix any $k \geq 1$. Let $\alpha_1, \ldots, \alpha_k$ be nonzero algebraic numbers over $\mathbf{Q}$. Let $D = [\mathbf{Q}(\alpha_1, \ldots, \alpha_k) : \mathbf{Q}]$ be the degree of the algebraic extension, and let h be the maximum height* [1] *of $\alpha_1, \ldots, \alpha_k$ over $\mathbf{Q}$. Then, the relation group*

$$L = \{\ (e_1, \ldots, e_k) \in \mathbf{Z}^k \mid \alpha_1^{e_1} \cdots \alpha_k^{e_k} = 1\ \}$$

is a lattice with a small basis. More specifically, L has a generating set $v_1, \ldots, v_\ell \in \mathbf{Z}^k$, $1 \leq \ell \leq k$, such that the maximum entry in these vectors $\max_{1 \leq i \leq \ell\ 1 \leq j \leq k} |v_{i,j}|$ is at most

$$(ckh)^{k-1} D^{k-1} \frac{(\log(D+2))^{3k-3}}{(\log\log(D+2))^{3k-4}},$$

where c is some absolute constant.

In order to appreciate this remarkable theorem of Masser, let's look at its implication on the computational complexity of finding such a basis. Note that in the

[1] The height function of an algebraic number is essentially the sum of the degree and the binary length of all coefficients in the defining equation over $\mathbf{Q}$.

inequality, the left hand side refers to the *quantity* $v_{i,j}$, while the right hand side essentially refers to the binary length of the input data. (Recall that k is fixed.) Thus, an exhaustive search can be done in polynomial time to find a small basis.

We call a shift of a lattice $L + v$, for some $v \in \mathbf{Z}^k$, an *affine lattice*. We say that an affine lattice has *small description* if it is $L + v$ for some small v and L has a small basis; here small means all entries are polynomially bounded in *quantity* (not merely in binary length).

THEOREM 3.2. *Given nonzero algebraic numbers $\alpha, \beta_1, \ldots, \beta_k$, the set*

$$\{ \ (j_1, \ldots, j_k) \mid \alpha\beta_1^{j_1} \cdots \beta_k^{j_k} = 1 \ \}$$

is either empty or is an affine lattice with rank at most k and a small description. Moreover, it is decidable in polynomial time whether it is empty, and if not, to compute a small description in terms of an off-set vector v and a small basis. Finally, if the rank of the affine lattice is k, then all of $\alpha, \beta_1, \ldots, \beta_k$ are roots of unity.

Proof. Consider the lattice

$$L = \{ \ (i, j_1, \ldots, j_k) \mid \alpha^i \beta^{j_1} \cdots \beta^{j_k} = 1 \ \}.$$

By Ge's theorem [19], we get a small basis $v_1, \ldots, v_\ell$, $\ell \leq k + 1$. Now we wish to intersect this lattice with the affine lattice $\{(1, j_1, \ldots, j_k) \mid j_1, \ldots, j_k \in \mathbf{Z}\}$.

Write

$$\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_\ell \end{pmatrix} = \begin{pmatrix} v_{10} & v_{11} & \ldots & v_{1k} \\ v_{20} & v_{21} & \ldots & v_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ v_{\ell 0} & v_{\ell 1} & \ldots & v_{\ell k} \end{pmatrix},$$

where $v_{ij} \in \mathbf{Z}$ and $|v_{ij}| \leq n^{O(1)}$.

We can first perform a basis reduction to transform the basis to a so-called canonical basis in the sense of Hermite. This can be done in polynomial time [27]

$$\begin{pmatrix} v_{10} & v_{11} & \ldots & v_{1k} \\ 0 & v_{21} & \ldots & v_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \cdot \end{pmatrix},$$

where $0 \leq v_{10}$, $0 \leq v_{11} \leq v_{21}$, etc. The row vectors still form a basis for L. We will still call them $v_1, \ldots, v_\ell$.

Now it is clear that this lattice intersects with $\{(1, j_1, \ldots, j_k) \mid j_1, \ldots, j_k \in \mathbf{Z}\}$ iff $v_{10} = 1$, and if so, then the intersection is $L' + v_1$, where L' is the lattice spanned by $v_2, \ldots, v_\ell$.

If the rank of the affine lattice is k, then $\ell = k + 1$. The basis matrix is a square matrix and all diagonal entries are nonzero. In particular, $v_{kk} \neq 0$, and $\beta_k^{v_{kk}} = 1$. Thus, β_k is a root of unity. Continuing, we have $v_{k-1,k-1} \neq 0$, and $\beta_{k-1}^{v_{k-1,k-1}} = \beta_k^{-v_{k-1,k}}$, which is a root of unity. Thus, β_{k-1} is also a root of unity. The proof is completed by an easy induction. $\square$

If any of the algebraic numbers $\alpha, \beta_1, \ldots, \beta_k$ are 0, we will adopt the convention that $0^0 = 1$ and $0^i = 0$ for $i > 0$. Under this convention, the statement in Theorem 3.2 can be easily adapted to allow 0's. Now to prove Theorem 1.1, note that the set of k-tuples of nonnegative integers $(i_1, \ldots, i_k)$ such that (1) holds is obtained by intersecting the affine lattice L of Theorem 3.2 with the set of k-tuples of nonnegative integers. By Lenstra [30], membership in the intersection can be decided in polynomial time.

4. The proof.

We will first illustrate the new technique, in a simple setting, by demonstrating how swiftly the Kannan–Lipton Theorem [28] can be proved using this technique.

4.1. The Kannan–Lipton theorem.

THEOREM 4.1. *There is a polynomial-time algorithm for the following problem:*
Given two rational matrices A, B, is there a nonnegative integer i
such that $A^i = B$?

Proof. Put A in JNF. Restrict to a subspace V_λ, where the only eigenvalue for A on V_λ is λ.

- Case 1. $\lambda = 0$. Then $A|_{V_\lambda}$ is nilpotent, $(A|_{V_\lambda})^n = 0$. Thus we need only check the cases $(A|_{V_\lambda})^i = (B|_{V_\lambda})$, for $0 \leq i \leq n$.
- Case 2. $\lambda \neq 0$. Consider the solutions to $\lambda^i = c$, which is obtained by looking at any diagonal entry in $(A|_{V_\lambda})^i = B|_{V_\lambda}$. If the solution set is empty, we are finished. Suppose the solution set is an affine lattice of rank 0, i.e., there is a unique solution i and since $i \leq n^{O(1)}$ we can directly check if $A^i = B$. The only nontrivial case is rank 1. Then by Theorem 3.2, λ is a root of unity. So must be c, hence $c \neq 0$. The order of λ must be polynomially bounded, since $\chi_A(\lambda) = 0$.
  1. $A|_{V_\lambda}$ is diagonal. Then it is completely determined by Theorem 3.2.
  2. $A|_{V_\lambda}$ is not diagonal. Then some Jordan block has size ≥ 2. We can symbolically compute the powers of $A|_{V_\lambda}$, up to $(A|_{V_\lambda})^i$, for $i \leq 2^{n^{O(1)}}$. This uses the fact that λ is a root of unity, and we know the closed form formula for powers of JNF. Thus, on the diagonal we get equation $\lambda^i = c$, and on the off-diagonal we get an equation $\binom{i}{1}\lambda^{i-1} = c'$. This gives the necessary condition that $i = \lambda c'/c$. If $\lambda c'/c$ is not a nonnegative integer, then there is no solution. Otherwise, we simply check directly if $\lambda^i = c$.

The Kannan–Lipton Theorem is proved. ∎

4.2. The method of successive restriction. Now the setting is $V_{\lambda,\mu}$. We will write A for $A|_{V_{\lambda,\mu}}$ and do the same for B. There will be four cases, depending on whether $\lambda = 0$ and/or $\mu = 0$. The cases when at least one of the eigenvalues is 0 are in fact simpler, since then the matrix is nilpotent, so that we need only check powers up to n. (Recall that on $V_{\lambda,\mu}$, each matrix has exactly one eigenvalue.) We will omit the details on these three cases and assume $\lambda \neq 0$ and $\mu \neq 0$. If both A and B are diagonal matrices $\lambda I$and μI, then this case is completely determined by Theorem 3.2. Suppose A is not diagonal. Put A in its JNF. Since $A \neq \lambda I$, at least one block is of dimension > 1.

Let

$$V' = \ker(\lambda I - A)$$

and

$$V'' = \ker(\lambda I - A)^2.$$

Since A and B commute, both V' and V'' are invariant subspaces of B as well as A. In terms of the Jordan form of A, V' corresponds to the first rows and columns of each λ-block. And V'' corresponds to the first and the second rows and columns (whenever a second row and column exists) of each λ-block. Since at least one block is of dimension > 1, $V' \neq V''$.

Let $n_1 = \dim V'$ and $n_1 + n_2 = \dim V''$; then $n_1 \geq n_2 \geq 1$.

Let $v_{11}, \ldots, v_{1k_1}, v_{21}, \ldots, v_{2k_2}, \ldots, v_{\ell 1}, \ldots, v_{\ell k_\ell}$ be the basis vectors for which A has its Jordan form, where $v_{i1}, \ldots, v_{ik_i}$ is the basis vectors corresponding to the ith block. We also assume that the blocks are ordered so that $k_1 \geq \cdots \geq k_\ell$. Then $v_{11}, \ldots, v_{\ell 1}$ spans the eigenspace $V' = \ker(\lambda I - A)$, and consequently $\ell = n_1$. By assumption $k_1 \geq 2$, and let $\ell' = \max\{i \mid k_i \geq 2\}$; then $v_{11}, \ldots, v_{\ell,1}, v_{12}, \ldots, v_{\ell' 2}$ spans V'' and $\ell' = n_2$.

Consider $A\mid_{V''}$ and $B\mid_{V''}$. If we order the basis vectors as $v_{11}, \ldots, v_{\ell,1}, v_{12}, \ldots,$ $v_{\ell' 2}$, then A has the form

$$A = \begin{pmatrix} \lambda I_{n_2} & 0 & I_{n_2} \\ 0 & \lambda I_{n_1 - n_2} & 0 \\ 0 & 0 & \lambda I_{n_2} \end{pmatrix}.$$

By decomposing B in blocks of the same dimensions, one can easily verify that

$$A \text{ and } B \text{ commute } \iff B \text{ is of the following} \\ \text{block upper triangular form:}$$

$$B = \begin{pmatrix} X & Y & Z \\ 0 & U & V \\ r0 & 0 & X \end{pmatrix},$$

where X, Y, Z, U, V are of the appropriate dimensions.

(If $n_1 = n_2$, then the middle blocks $\lambda I_{n_1 - n_2}$ in A, and Y, U, and V in B are understood not to appear.)

4.2.1. Restriction to V'. Choose a basis transformation of V' such that $B\mid_{V'}$ is in its Jordan form. Note that on V', $A\mid_{V'}$ is the scalar matrix λI_{n_1} which is unchanged under all basis transformations. Thus, under this basis, $A\mid_{V'}$ is still the scalar matrix λI_{n_1}, and $B\mid_{V'}$ is a direct sum of Jordan blocks all of which have μ on its diagonal, since *Spec*$(B\mid_{V'}) = \{\mu\}$. By comparing eigenvalues, we get $\lambda^i \mu^j = c$. By Theorem 3.2 we get either an empty set of solutions or an affine lattice of small description. If it is empty, we are finished; or if the rank is 0, we can directly check it. If the rank is 1, then we can reduce it to Kannan–Lipton's theorem. In fact, let $i = i_0 + as$ and $j = j_0 + bs$ for small i_0, j_0, a, b; then we have to solve for s in $(A^a B^b)^s = (CA^{-i_0} B^{-j_0})$.

Now suppose the rank is 2. Then both λ and μ are roots of unity. We note that in this case we can compute high powers of A and B, up to $2^{n^{O(1)}}$, in polynomial time. This is accomplished separately for A and B by first putting the target matrix in JNF, then taking its power, and finally reverting back.

Suppose $B\mid_{V'}$ is not a scalar matrix μI. Then some μ-block of $B\mid_{V'}$ has dimension > 1. Then consider

$$U' = \ker(\mu I - B\mid_{V'})$$

and

$$U'' = \ker(\mu I - B\mid_{V'})^2.$$

As before, both U' and U'' are invariant subspaces of A as well as B. Since at least one block of $B\,|_{V'}$ is of dimension > 1, $U' \neq U''$.

Let $m_1 = \dim U'$ and $m_1 + m_2 = \dim U''$; then $m_1 \geq m_2 \geq 1$, and if we order the basis vectors appropriately, we have $A\,|_{U''} = \lambda I_{m_1+m_2}$ and

$$B\,|_{U''} = \begin{pmatrix} \mu I_{m_2} & 0 & I_{m_2} \\ 0 & \mu I_{m_1-m_2} & 0 \\ 0 & 0 & \mu I_{m_2} \end{pmatrix}.$$

(Again, if $m_1 = m_2$, then the middle blocks are understood not to appear.)

Implied by the forms of $A\,|_{U''}$ and $B\,|_{U''}$ is the fact that the first and the last m_2 basis vectors together generate an invariant subspace for both A and B. By focusing on this subspace, we get a necessary condition of $A^iB^j = C$ in the form of a pair of equations

$$\lambda^i\mu^j = c, \qquad j\lambda^i\mu^{j-1} = c'.$$

This gives us, upon substitution, $j = \mu c'/c$, which reduces the problem to the one-variable case.

4.2.2. Restriction to V''. Now we suppose on V', $B\,|_{V'} = \mu I_{n_1}$; thus on V'',

$$B\,|_{V''} = \begin{pmatrix} \mu I_{n_2} & 0 & Z \\ 0 & \mu I_{n_1-n_2} & V \\ 0 & 0 & \mu I_{n_2} \end{pmatrix}.$$

(Once again, the middle blocks disappear if $n_1 = n_2$.)

Suppose $n_1 > n_2$. $V^* = \mathrm{span}\{v_{n_2+1\,1}, \ldots, v_{n_1 1}\}$, the subspace corresponding to the middle blocks, is an invariant subspace for both A and B. Consequently we may consider the quotient space V''/V^* and the induced action of A and B on this quotient space, denoted by $\tilde{A}$ and $\tilde{B}$, respectively. (If $n_1 = n_2$, then $V^* = 0$ and $V''/V^* = V''$.) Under the induced basis from $\{v_{11}, \ldots, v_{n_2,1}, v_{1,2}, \ldots, v_{n_2,2}\}$ for V''/V^*,

$$\tilde{A} = \begin{pmatrix} \lambda I_{n_2} & I_{n_2} \\ 0 & \lambda I_{n_2} \end{pmatrix}$$

and

$$\tilde{B} = \begin{pmatrix} \mu I_{n_2} & Z \\ 0 & \mu I_{n_2} \end{pmatrix}.$$

If $Z = 0$ then we will get a necessary condition

$$\lambda^i\mu^j = c, \qquad i\lambda^{i-1}\mu^j = c'.$$

This gives us, upon substitution, $i = \lambda c'/c$, which reduces the problem to the one-variable case.

If $Z \neq 0$ then we will get a necessary condition

$$\lambda^i\mu^j = c, \qquad j\lambda^i\mu^{j-1}z + i\lambda^{i-1}\mu^j = c',$$

for some known algebraic numbers c, c' and $z \neq 0$. Upon substitution, we get

$$\frac{i}{\lambda} + z\frac{j}{\mu} = \frac{c'}{c}.$$

Solving i in terms of j,

$$i = \frac{\lambda c'}{c} - \frac{z\lambda}{\mu} j.$$

If $\frac{z\lambda}{\mu}$ is not rational, then we will have at most one pair of integral solution (i, j), and we can easily find it and test it. If $\frac{z\lambda}{\mu}$ is rational, then unless $\lambda c'/c$ is rational, there is no solution, and if $\lambda c'/c$ is rational, then there is an integral relationship between i and j,

$$ui + vj = w.$$

We can solve this equation in general form, $i = i_0 + v_1 t$ and $j = j_0 - u_1 t$, where $u_1 = u/\gcd(u, v)$ and $v_1 = v/\gcd(u, v)$. Substituting back in $A^i B^j = C$, we get $(A^{i_0} B^{j_0})(A^{v_1} B^{-u_1})^t = C$, which reduces it to the case with one variable t. This proves Theorem 1.2. Theorem 1.3 is proved by removing the restriction to nonnegative exponents at the appropriate places in the proof. Alternatively, as suggested by the referee, apply Theorem 1.2 four times with the respective generator pairs A, B; A, B^{-1}; A^{-1}, B; and A^{-1}, B^{-1}.

Acknowledgments. This paper would not have been possible without the contribution of M. L. Robinson, who has pointed us to transcendental number theory and in particular to Masser's theorem. We thank him for this and for many helpful discussions. We would also like to acknowledge helpful discussions with Katalin Friedl, Ravi Kannan, Susan Landau, Laci Lovász, Andrew Odlyzko, Lajos Ronyai, and William Velez. Zeke Zalcstein would also like to thank Kamal Abdali for TEXnical assistance.

REFERENCES

[1] S. I. Adyan, *Algorithmic unsolvability of problems of recognition of certain properties of groups*, Dokalady Akad. Nauk SSSR, 103 (1955), pp. 533–535 (in Russian).

[2] L. Babai, *Trading group theory for randomness*, in Proceedings of the 17th Symposium on the Theory of Computing, Providence, RI, 1985, pp. 421–429.

[3] L. Babai, E. Luks, and A. Seress, *Permutation groups in NC*, in Proceedings of the 19th Symposium on the Theory of Computing, New York, 1987, pp. 272–282.

[4] L. Babai, G. Cooperman, L. Finkelstein, E. Luks, and A. Seress, *Fast Monte Carlo algorithms for permutation groups*, in Proceedings of the 23rd Symposium on the Theory of Computing, New Orleans, 1991, pp. 90–100.

[5] L. Babai and E. Szemeredi, *On the complexity of matrix group problems* I, in Proceedings of the 25th Symposium on the Foundation of Computer Science, Singer Island, FL, 1984, pp. 229–240.

[6] R. Beals and L. Babai, *Las Vegas algorithms for matrix groups*, in Proceedings of the 33rd Symposium on the Foundation of Computer Science, Palo Alto, CA, 1993, pp. 427–436.

[7] L. Babai, R. Beals, J. Cai, G. Ivanos, and E. Luks, *Multiplicative equations over commuting matrices*, in Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Atlanta, GA, 1996, pp. 498–507.

[8] M. Beaudry, *Membership testing in commutative transformation semigroups*, Inform. and Comput., 79 (1988), pp. 84–93.

[9] J.-C. Birget, S. Margolis, J. Meakin, and P. Weil, *PSPACE-completeness of certain algorithmic problems on the subgroups of free groups*, in Proceedings International Comput. Algorithms Lang. Programming, 1994, pp. 274–285.

[10] P. E. Blanksby and H. L. Montgomery, *Algebraic integers near the unit circle*, Acta Arith., 18 (1971), pp. 355–369.

[11] W. W. Boone, *The word problem*, Ann. of Math., 70 (1952), pp. 207–265.

[12] J. Cai, *Computing Jordan normal forms exactly for commuting matrices in polynomial time*, Internat. J. Foundations Comput. Sci., 5 (1994), pp. 293–302.

[13] J. Cai, R. Lipton, and Y. Zalcstein, *Complexity of the membership problem for* 2*-generated commutative semigroups of rational matrices*, in Proceedings Foundation of Computer Science, Santa Fe, NM, 1994, pp. 135–142.

[14] M. Curzio, P. Longobardi, and M. Mai, *Su di un problema combinatorio in teoria dei gruppi*, Atti Lincei VIII 74, (1983), pp. 136–142.

[15] M. Furst, J. Hopcroft, and E. Luks, *Polynomial time algorithms for permutation groups*, in Proceedings Foundation of Computer Science, 1980, pp. 60–78.

[16] M. Garzon and Y. Zalcstein, *Complexity of Grigorchuk groups with application to cryptography*, Theoret. Comput. Sci., 88 (1991), pp. 83–98.

[17] M. Garzon and Y. Zalcstein, *On permutation properties in groups and semigroups*, Semigroup Forum, 35 (1987), pp. 337–351.

[18] G. Ge, *Testing equalities of multiplicative representations in polynomial time*, in Proceedings Foundation of Computer Science, 1993, pp. 422–426.

[19] G. Ge, *Algorithms Related to the Multiplicative Representations of Algebraic Numbers*, Ph.D. dissertation, Department of Mathematics, University of California at Berkeley, Berkeley, CA, 1993.

[20] R. I. Grigorchuk, *Degrees of growth of finitely generated groups and the theory of invariant means*, Math. USSR Izvestiya, 25 (1985), pp. 259–300.

[21] I. Kaplansky, *Fields and Rings*, 2nd ed., University of Chicago Press, Chicago, IL, 1972.

[22] D. E. Knuth, *Notes on Efficient Representation of Permutation Groups*, manuscript, 1981.

[23] M. Krom, *An unsolvable problem with products of matrices*, Math. Systems Theory, 14 (1981), pp. 335–337.

[24] L. Kopytov, *Solvability of the occurrence problem in finitely generated soluble groups of matrices over the field of algebraic numbers*, Algebra and Logic, 7 (1968), pp. 388–393.

[25] D. Kozen, *Lower bounds for natural proof systems*, in Proceedings Foundation of Computer Science, Providence, RI, 1977, pp. 254–266.

[26] R. Kannan, *The Size of Numbers in the Analysis of Certain Algorithms*, Ph.D. dissertation, Operations Research Dept., Cornell University, Ithaca, NY, 1980.

[27] R. Kannan and A. Bachem, *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8 (1979), pp. 499–507.

[28] R. Kannan and R. Lipton, *The orbit problem is decidable*, in Proceedings of the 12th Symposium on the Theory of Computing, Los Angeles, CA, 1980, pp. 252–261. See also *Polynomial-time algorithms for the orbit problem*, JACM, 33 (1986), pp. 808–821.

[29] R. Kannan, A. K. Lenstra, and L. Lovász, *Polynomial factorization and non-randomness of bits of algebraic numbers and certain transcendental numbers*, Math. Comp., 50 (1988), pp. 235–250.

[30] H. W. Lenstra, *Integer programming with a fixed number of variables*, Math. Oper. Res., 8 (1983), pp. 538–548.

[31] A. K. Lenstra, H. W. Lenstra, and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 515–534.

[32] S. Landau, *personal communication*, University of Massachusetts, Amherst, MA, 1994.

[33] R. Lipton and Y. Zalcstein, *Word problems solvable in logspace*, JACM, 24 (1977), pp. 522–526.

[34] L. Lovasz, *An Algorithmic Theory of Numbers, Graphs and Convexity*, SIAM, Philadelphia, 1986.

[35] E. Luks, *Computing in solvable matrix groups*, in Proceedings Foundation of Computer Science, Pittsburgh, PA, 1992, pp. 111–120.

[36] D. W. Masser, *Linear relations on algebraic groups*, in New Advances in Transcendence Theory, A. Baker, ed., Cambridge University Press, Cambridge, UK, 1988, pp. 248–262.

[37] K. A. Mihailova, *The occurrence problem for a direct product of groups*, Dokl. Akad. Nauk, 119 (1958), pp. 1103–1105.

[38] P. S. Novikov, *On the algorithmic unsolvability of the problem of equality of words in group theory*, Trudy Mat. Inst. Akad. Nauk SSSR, 44 (1955), pp. 1–144 (in Russian).

[39] M. S. Paterson, *Undecidability in* 3 *by* 3 *matrices*, J. Math. Phys., 49 (1970), pp. 105–107.

[40] M. Rabin, *Recursive unsolvability of group theoretic problems*, Ann. Math., 67 (1958), pp. 172–194.

[41] M. Rabin, *Computable algebra, general theory and theory of computable fields*, Trans. Amer. Math. Soc., 95 (1960), pp. 341–360.

[42] H. U. Simon, *Word problems for groups and context-free languages*, in Foundations of Computation Theory, Lecture Notes in Comput. Sci., Springer-Verlag, New York, 1979, pp. 417–422.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1889–1906

THE LOAD AND AVAILABILITY OF BYZANTINE QUORUM SYSTEMS*

DAHLIA MALKHI[†], MICHAEL K. REITER[‡], AND AVISHAI WOOL[‡]

Abstract. Replicated services accessed via *quorums* enable each access to be performed at only a subset (quorum) of the servers and achieve consistency across accesses by requiring any two quorums to intersect. Recently, b-masking quorum systems, whose intersections contain at least $2b+1$ servers, have been proposed to construct replicated services tolerant of b-arbitrary (Byzantine) server failures. In this paper we consider a hybrid fault model allowing benign failures in addition to the Byzantine ones. We present four novel constructions for b-masking quorum systems in this model, each of which has optimal *load* (the probability of access of the busiest server) or optimal availability (probability of some quorum surviving failures). To show optimality we also prove lower bounds on the load and availability of any b-masking quorum system in this model.

Key words. quorum systems, Byzantine failures, replication, load, availability, distributed computing

AMS subject classifications. 62N05, 68M10, 68P15, 68Q22, 68R05, 90A28

PII. S0097539797325235

1. Introduction.

Quorum systems are well-known tools for increasing the efficiency of replicated services, as well as their availability when servers may fail benignly. A quorum system is a set of subsets (quorums) of servers, every pair of which intersect. Quorum systems enable each client operation to be performed only at a quorum of the servers, while the intersection property makes it possible to preserve consistency among operations at the service.

Quorum systems work well for environments where servers may fail benignly. However, when servers may suffer arbitrary (Byzantine) failures, the intersection property does not suffice for maintaining consistency; two quorums may intersect in a subset containing *faulty* servers only, which may deviate arbitrarily and undetectably from their assigned protocol. Malkhi and Reiter thus introduced *masking quorums systems* [25], in which each pair of quorums intersects in sufficiently many servers to mask out the behavior of faulty servers. More precisely, a *b-masking quorum system* is one in which any two quorums intersect in $2b+1$ servers, which suffices to ensure consistency in the system if at most b servers suffer Byzantine failures.

In this paper we develop four new constructions for b-masking quorum systems. For the first time in this context, we distinguish between masking Byzantine faults and surviving a possibly larger number of benign faults. Our systems remain available in the face of any f crashes, where f may be significantly larger than b (such a system is called f-resilient). In addition, our constructions demonstrate optimality (ignoring constants) in two widely accepted measures of quorum systems, namely *load* and *crash probability*. The load ($\mathcal{L}$), a measure of best-case performance of the quorum system, is the probability with which the busiest server is accessed under the best

*Received by the editors July 29, 1997; accepted for publication (in revised form) March 23, 1999; published electronically April 11, 2000. A preliminary version of this paper appeared in Proceedings of the 16th ACM Symposium on Principles of Distributed Computing, 1997.
http://www.siam.org/journals/sicomp/29-6/32523.html

[†]AT&T Labs–Research, Florham Park, NJ, and The Institute of Computer Science, The Hebrew University of Jerusalem, Israel (dalia@cs.huji.ac.il).

[‡]Bell Labs, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ (reiter@research.bell-labs.com, yash@acm.org).

possible strategy for accessing quorums. The crash probability (F_p) is the probability, assuming that each server crashes with independent probability p, that all quorums in the system will contain at least one crashed server (and thus will be unavailable). The crash probability is an even more refined measure of availability than f, as a good system will tolerate many failure configurations with more than f crashes. Three of our systems are the first systems to demonstrate optimal load for b-masking quorum systems, and two of our systems each demonstrate optimal crash probability for its resilience f. In proving optimality of our constructions, we prove new lower bounds for the load and crash probability of masking quorum systems.

The techniques for achieving our constructions are of interest in themselves. Two of the constructions are achieved using a *boosting* technique, which can transform any regular (i.e., benign fault-tolerant) quorum system into a masking quorum system of an appropriately larger system. Thus, it makes all known quorum constructions available for Byzantine environments (of appropriate sizes). In the analysis of one of our best systems we employ strong results from percolation theory.

The rest of this paper is structured as follows. We review related work and preliminary definitions in sections 2 and 3, respectively. In section 4 we prove bounds on the load and crash probability for b-masking quorum systems and introduce quorum composition. In sections 5–7 we describe our new constructions. We discuss our results in section 8.

2. Related work. Our work borrows from extensive prior work in benignly fault-tolerant quorum systems (e.g., [12, 39, 24, 11, 15, 4, 9, 1, 7, 31, 36]). The notion of availability we use here (crash probability) is well known in reliability theory [5] and has been applied extensively in the analysis of quorum systems (cf. [4, 34, 35] and the references therein). The load of a quorum system was first defined and analyzed in [31], which proved a lower bound of $\Omega(\frac{1}{\sqrt{n}})$ on the load of any quorum system (and, a fortiori, any masking quorum system) over n servers. In proving load-optimality of our constructions, we generalize this lower bound to $\Omega(\sqrt{\frac{b}{n}})$ for b-masking quorum systems.

Grids, which form the basis for our multigrid (denoted M-Grid) construction, were proposed in [24, 7, 21, 25]. The technique of quorum composition, which we use in our recursive threshold (RT) and boosted finite projective planes (boostFPP) constructions, has been studied in [29, 33, 32] under various names such as "coterie join" and "recursive majority." Our multipath (M-Path) construction generalizes the system of [41], coupled with the analysis of the Paths construction of [31], and the recent system of [6].

Several constructions of masking quorum systems were given in [25] for a variety of failure models. For the model we consider here—i.e., any b servers may experience Byzantine failures—that work gave two constructions. We compare those constructions to ours in section 8.

Hybrid failure models have been considered in other works (e.g., [10, 22, 23, 38]).

3. Preliminaries. In this section we introduce notation and definitions used in the remainder of the paper. Much of the notation introduced in this section is summarized in Table 1 for quick reference.

We assume a *universe* U of servers, $|U| = n$, over which our quorum systems will be constructed. Servers that obey their specifications are *correct*. A *faulty* server, however, may deviate from its specification arbitrarily. We assume that up to b servers may fail arbitrarily and that $4b < n$, since this is necessary for a b-masking quorum

TABLE 1
The notation used in this paper.

b	Maximum number of Byzantine server failures.
$c(\mathcal{Q})$	Size of the smallest quorum in $\mathcal{Q}$.
f	Resilience (Definition 3.4).
$F_p(\mathcal{Q})$	Crash probability (Definition 3.10).
$\mathcal{IS}(\mathcal{Q})$	Size of smallest intersection between any two quorums in $\mathcal{Q}$.
$\mathcal{L}(\mathcal{Q})$	Load of $\mathcal{Q}$ (Definition 3.8).
$\mathcal{MT}(\mathcal{Q})$	Size of a smallest transversal of $\mathcal{Q}$ (Definition 3.3).
n	Number of servers (i.e., $\|U\| = n$).
p	Independent probability that each server crashes.
$\mathcal{Q}$	A quorum system (Definition 3.1).
U	Universe of servers.

system to exist [25]. Beginning in section 3.2.2, we will also distinguish benign (crash) failures as a particular failure of interest, and in general there may be more than b such failures.

3.1. Quorum systems.

DEFINITION 3.1. *A* quorum system $\mathcal{Q} \subseteq 2^U$ *is a collection of subsets of* U*, each pair of which intersect. Each* $Q \in \mathcal{Q}$ *is called a* quorum.

We use the following notation. The cardinality of the smallest quorum in $\mathcal{Q}$ is denoted by $c(\mathcal{Q}) = \min\{|Q| : Q \in \mathcal{Q}\}$. The size of the smallest intersection between any two quorums is denoted by $\mathcal{IS}(\mathcal{Q}) = \min\{|Q \cap R| : Q, R \in \mathcal{Q}\}$. The degree of an element $i \in U$ in a quorum system $\mathcal{Q}$ is the number of quorums that contain i: $\deg(i) = |\{Q \in \mathcal{Q} : i \in Q\}|$.

DEFINITION 3.2. *A quorum system* $\mathcal{Q}$ *is* (s, d)-fair *if* $|Q| = s$ *for all* $Q \in \mathcal{Q}$ *and* $\deg(i) = d$ *for all* $i \in U$*.* $\mathcal{Q}$ *is called* fair *if it is* (s, d)*-fair for some* s *and* d*.*

DEFINITION 3.3. *A set* T *is a* transversal *of a quorum system* $\mathcal{Q}$ *if* $T \cap Q \neq \varnothing$ *for every* $Q \in \mathcal{Q}$*. The cardinality of the smallest transversal is denoted by* $\mathcal{MT}(\mathcal{Q}) = \min\{|T| : T \text{ is a transversal of } \mathcal{Q}\}$*.*

Regular quorum systems, with $\mathcal{IS}(\mathcal{Q}) = 1$, are insufficient to guarantee consistency in case of Byzantine failures. Malkhi and Reiter [25] defined several varieties of quorum systems for Byzantine environments, which are suitable for different types of services. In this paper we focus on masking quorum systems.

DEFINITION 3.4 (see [25]). *The* resilience f *of a quorum system* $\mathcal{Q}$ *is the largest* k *such that for every set* $K \subseteq U$, $|K| = k$*, there exists* $Q \in \mathcal{Q}$ *such that* $K \cap Q = \varnothing$*.*

Remark. The resilience of any quorum system $\mathcal{Q}$ is $f = \mathcal{MT}(\mathcal{Q}) - 1$.

DEFINITION 3.5 (see [25]). *A quorum system* $\mathcal{Q}$ *is a* b-masking quorum system *if it is resilient to* $f \geq b$ *failures, and obeys the following* consistency *requirement:*

$$\forall Q_1, Q_2 \in \mathcal{Q} : |Q_1 \cap Q_2| \geq 2b + 1. \tag{3.1}$$

Remark. Informally, if we view the service as a shared variable which is updated and read by the clients, then the resilience requirement of Definition 3.4 ensures that no set of $b \leq f$ faulty servers will be able to block update operations (e.g., by causing every update transaction to abort). The consistency requirement of Definition 3.5 ensures that read operations can mask out any faulty behavior of up to b servers. Examples of protocols implementing various data abstractions using b-masking quorum systems can be found in [25, 26, 27].

LEMMA 3.6. *Let* $\mathcal{Q}$ *be a quorum system. Then* $\mathcal{Q}$ *is* b*-masking if both the following conditions hold:*

(1) $\mathcal{MT}(\mathcal{Q}) \geq b+1$;
(2) $\mathcal{IS}(\mathcal{Q}) \geq 2b+1$.

Proof. Assume that $\mathcal{MT}(\mathcal{Q}) \geq b+1$. To see that $\mathcal{Q}$ is resilient to b failures, note that if there exists some K such that $K \cap Q \neq \varnothing$ for all $Q \in \mathcal{Q}$, then K is a transversal. By the minimality we have $|K| \geq b+1$, and we are done. Condition 2 immediately implies (3.1). $\square$

COROLLARY 3.7. *Let $\mathcal{Q}$ be a quorum system, and let $b = \min\{\mathcal{MT}(\mathcal{Q})-1, \frac{\mathcal{IS}(\mathcal{Q})-1}{2}\}$. Then $\mathcal{Q}$ is b-masking.*

3.2. Measures.

The goal of using quorum systems is to increase the availability of replicated services and decrease their access costs. A natural question is how well any particular quorum system achieves these goals, and moreover, how well it compares with other quorum systems. Several measures will be of interest to us.

3.2.1. Load.

A measure of the inherent performance of a quorum system is its *load*. Naor and Wool define the load of a quorum system as the frequency of accessing the busiest server using the best possible strategy [31]. More precisely, given a quorum system $\mathcal{Q}$, an *access strategy* w is a probability distribution on the elements of $\mathcal{Q}$; i.e., $\sum_{Q \in \mathcal{Q}} w(Q) = 1$. The value $w(Q) \geq 0$ is the frequency of choosing quorum Q when the service is accessed. The load is then defined as follows.

DEFINITION 3.8. *Let a strategy w be given for a quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ over a universe U. For an element $u \in U$, the load induced by w on u is $l_w(u) = \sum_{Q_i \ni u} w(Q_i)$. The load induced by a strategy w on a quorum system $\mathcal{Q}$ is $\mathcal{L}_w(\mathcal{Q}) = \max_{u \in U}\{l_w(u)\}$. The* system load *on a quorum system $\mathcal{Q}$ is $\mathcal{L}(\mathcal{Q}) = \min_w\{\mathcal{L}_w(\mathcal{Q})\}$, where the minimum is taken over all strategies.*

We reiterate that the load is a best-case definition. The load of the quorum system will be achieved only if an optimal access strategy is used and only in the case that no failures occur. A strength of this definition is that the load is a property of a quorum system and not of the protocol using it. Examples of load calculations can be found in [40]. As an aside, we note that not every quorum system can have a strategy that induces the same load on each server. In [16] it is shown that for some quorum systems it is impossible to balance the load perfectly.

Recall that $c(\mathcal{Q})$ denotes the cardinality of the smallest quorum in $\mathcal{Q}$. The next result will be useful to us in what follows (recall Definition 3.2).

PROPOSITION 3.9 (see [31]). *Let $\mathcal{Q}$ be a fair quorum system. Then $\mathcal{L}(\mathcal{Q}) = c(\mathcal{Q})/n$.*

3.2.2. Availability.

By definition a b-masking quorum system can mask up to b arbitrary (Byzantine) failures. However, such a system may be resilient to more *benign* failures. By benign failures we mean any failures that render a server unresponsive, which we refer to as *crashes* to distinguish them from Byzantine failures.

The resilience f of a quorum system provides one measure of how many crash failures a quorum system is *guaranteed* to survive, and indeed this measure has been used in the past to differentiate among quorum systems [3]. However, it is possible that an f-resilient quorum system, though vulnerable to a few failure configurations of $f+1$ failures, can survive many configurations of more than f failures. One way to measure this property of a quorum system is to assume that each server crashes independently with probability p and then to determine the probability F_p that some quorum survives with no faulty members. This is known as *crash probability* and is formally defined as follows.

Definition 3.10. *Assume that each server in the system crashes independently with probability p. For every quorum $Q \in \mathcal{Q}$ let $\mathcal{E}_Q$ be the event that Q is* hit, *i.e., at least one element $i \in Q$ has crashed. Let $crash(\mathcal{Q})$ be the event that all the quorums $Q \in \mathcal{Q}$ were hit, i.e., $crash(\mathcal{Q}) = \bigwedge_{Q \in \mathcal{Q}} \mathcal{E}_Q$. Then the system crash probability is $F_p(\mathcal{Q}) = \mathbb{P}(crash(\mathcal{Q}))$.*

We would like F_p to be as small as possible. A desirable asymptotic behavior of F_p is that $F_p \to 0$ when $n \to \infty$ for *all* $p < 1/2$, and such an F_p is called Condorcet (after the Condorcet jury theorem [8]).

4. Building blocks. In this section, we prove several theorems which will be our basic tools in what follows. First we prove lower bounds on the load and availability of b-masking quorum systems, against which we measure all our new constructions. Then we prove the properties of a quorum composition technique, which we later use extensively.

4.1. The load and availability of masking quorum systems. We begin by establishing a lower bound on the load of b-masking quorum systems, thus tightening the lower bound on general quorum systems [31] as presented in [25].

Theorem 4.1. *Let $\mathcal{Q}$ be a b-masking quorum system. Then $\mathcal{L}(\mathcal{Q}) \geq \max\{\frac{2b+1}{c(\mathcal{Q})}, \frac{c(\mathcal{Q})}{n}\}$.*

Proof. Let w be any strategy for the quorum system $\mathcal{Q}$, and fix $Q_1 \in \mathcal{Q}$ such that $|Q_1| = c(\mathcal{Q})$. Summing the loads induced by w on all the elements of Q_1, and using the fact that any two quorums have at least $2b+1$ elements in common, we obtain

$$\sum_{u \in Q_1} l_w(u) = \sum_{u \in Q_1} \sum_{Q_i \ni u} w(Q_i) = \sum_{Q_i} \sum_{u \in (Q_1 \cap Q_i)} w(Q_i)$$
$$\geq \sum_{Q_i} (2b+1) w(Q_i) = 2b+1.$$

Therefore, there exists some element in Q_1 that suffers a load of at least $\frac{2b+1}{|Q_1|}$.

Similarly, summing the total load induced by w on all of the elements of the universe, and using the minimality of $c(\mathcal{Q})$, we get

$$\sum_{u \in U} l_w(u) = \sum_{u \in U} \sum_{Q_i \ni u} w(Q_i) = \sum_{Q_i} |Q_i| w(Q_i)$$
$$\geq \sum_{Q_i} c(\mathcal{Q}) w(Q_i) = c(\mathcal{Q}).$$

Therefore, there exists some element in U that suffers a load of at least $\frac{c(\mathcal{Q})}{n}$. ◻

Corollary 4.2. *Let $\mathcal{Q}$ be a b-masking quorum system. Then $\mathcal{L}(\mathcal{Q}) \geq \sqrt{\frac{2b+1}{n}}$, and equality holds if $c(\mathcal{Q}) = \sqrt{(2b+1)n}$.*[1]

Remark. Corollary 4.2 shows that the threshold construction of [25] in fact has optimal load when $b = \Omega(n)$. E.g., when $b \approx n/4$ the obtained load is ≈ 0.75, but for such systems we can only hope for a constant load of $\approx 1/\sqrt{2} = 0.707$. However, the load of the threshold construction is always $\geq 1/2$, which is far from optimal for smaller values of b.

On the other hand, the grid-based construction of [25] does not have optimal load. It has quorums of size $O(b\sqrt{n})$ and load of roughly $2b/\sqrt{n}$. In what follows we

[1]To avoid repetitive notation, we omit floor and ceiling brackets from expressions for integral quantities.

show systems which significantly improve this: some of our new constructions have quorums of size $O(\sqrt{bn})$ and optimal load.

Our next propositions show lower bounds on the crash probability F_p in terms of $\mathcal{MT}(\mathcal{Q})$ and b.

PROPOSITION 4.3. *Let $\mathcal{Q}$ be a quorum system. Then $F_p(\mathcal{Q}) \geq p^{\mathcal{MT}(\mathcal{Q})} = p^{f+1}$ for any $p \in [0,1]$.*

Proof. Consider a minimal transversal T with $|T| = \mathcal{MT}(\mathcal{Q})$. If all the elements of T crash, then every quorum contains a crashed element, so $F_p(\mathcal{Q}) \geq p^{\mathcal{MT}(\mathcal{Q})}$. ∎

PROPOSITION 4.4. *Let $\mathcal{Q}$ be a b-masking quorum system. Then $F_p(\mathcal{Q}) \geq p^{c(\mathcal{Q})-2b}$ for any $p \in [0,1]$.*

Proof. Let $Q \in \mathcal{Q}$ be a minimal quorum with $|Q| = c(\mathcal{Q})$, and consider $Z \subset Q$, $|Z| = 2b$. Since $\mathcal{Q}$ is b-masking, then $|R \cap Q| \geq 2b+1$ for any $R \in \mathcal{Q}$, and so $|(Q \setminus Z) \cap R| \geq 1$ and $Q \setminus Z$ is a transversal. Therefore $\mathcal{MT}(\mathcal{Q}) \leq c(\mathcal{Q}) - 2b$, which we plug into Proposition 4.3. ∎

The next proposition is less general than Proposition 4.4, but it is applicable for most of our constructions and it gives a much tighter bound.

PROPOSITION 4.5. *Let $\mathcal{Q}$ be a b-masking quorum system such that $\mathcal{MT}(\mathcal{Q}) \leq (\mathcal{IS}(\mathcal{Q})+1)/2$. Then $F_p(\mathcal{Q}) \geq p^{b+1}$ for any $p \in [0,1]$.*

Proof. If $\mathcal{MT}(\mathcal{Q}) \leq (\mathcal{IS}(\mathcal{Q})+1)/2$, then from Corollary 3.7 we have that $b+1 = \mathcal{MT}(\mathcal{Q})$, which again we plug into Proposition 4.3. ∎

4.2. Quorum system composition. Quorum system composition is a well-known technique for building new systems out of existing components. We compose a quorum system $\mathcal{S}$ over another system $\mathcal{R}$ by replacing each element of $\mathcal{S}$ with a distinct copy of $\mathcal{R}$. In other words, when element i is used in a quorum $S \in \mathcal{S}$ we replace it with a complete quorum from the ith copy of $\mathcal{R}$. Using the terminology of reliability theory, the system $\mathcal{S} \circ \mathcal{R}$ has a *modular decomposition* where each module is a copy of $\mathcal{R}$. Formally, we have the following.

DEFINITION 4.6. *Let $\mathcal{S}$ and $\mathcal{R}$ be two quorum systems, over universes of sizes n_S and n_R, respectively. Let $\mathcal{R}_1, \dots, \mathcal{R}_{n_S}$ be n_S copies of $\mathcal{R}$ over disjoint universes. Then the* composition *of $\mathcal{S}$ over $\mathcal{R}$ is*

$$\mathcal{S} \circ \mathcal{R} = \left\{ \bigcup R_i : S \in \mathcal{S}, R_i \in \mathcal{R}_i \text{ for all } i \in S \right\}.$$

The next theorem summarizes the properties of quorum composition.

THEOREM 4.7. *Let $\mathcal{S}$ and $\mathcal{R}$ be two quorum systems, and let $\mathcal{Q} = \mathcal{S} \circ \mathcal{R}$. Then*

- *The universe size is $n_Q = n_S n_R$.*
- *The minimal quorum size is $c(\mathcal{Q}) = c(\mathcal{S})c(\mathcal{R})$.*
- *The minimal intersection size is $\mathcal{IS}(\mathcal{Q}) = \mathcal{IS}(\mathcal{S})\mathcal{IS}(\mathcal{R})$.*
- *The minimal transversal size is $\mathcal{MT}(\mathcal{Q}) = \mathcal{MT}(\mathcal{S})\mathcal{MT}(\mathcal{R})$.*
- *Denote the crash probability functions of $\mathcal{S}$ and $\mathcal{R}$ by $s(p) = F_p(\mathcal{S})$ and $r(p) = F_p(\mathcal{R})$. Then $F_p(\mathcal{Q}) = s(r(p))$.*
- *The load is $\mathcal{L}(\mathcal{Q}) = \mathcal{L}(\mathcal{S})\mathcal{L}(\mathcal{R})$.*

Proof. The behavior of the combinatorial parameters n_Q, $c(\mathcal{Q})$, $\mathcal{IS}(\mathcal{Q})$, and $\mathcal{MT}(\mathcal{Q})$ is obvious. The behavior of $F_p(\mathcal{Q})$ is standard in reliability theory (cf. [5]). As for the load, consider the following strategy: pick a quorum $S \in \mathcal{S}$ using the optimal strategy for $\mathcal{S}$. Then for each element $i \in S$, pick a quorum $R_i \in \mathcal{R}_i$ using the optimal strategy for (the ith copy of) $\mathcal{R}$. Clearly this strategy induces a load of $\mathcal{L}(\mathcal{S})\mathcal{L}(\mathcal{R})$, and hence $\mathcal{L}(\mathcal{Q}) \leq \mathcal{L}(\mathcal{S})\mathcal{L}(\mathcal{R})$.

We now show the inequality in the opposite direction. Enumerate the elements of $\mathcal{Q}$ by denoting the jth element in $\mathcal{R}_i$ by u_{ij}, let $Q(S) = \{\bigcup R_i : R_i \in \mathcal{R}_i \text{ for all } i \in S\}$

be the set of all quorums that are based on some $S \in \mathcal{S}$, and let $w^{\mathcal{Q}}$ be an access strategy on $\mathcal{Q}$. Then $w^{\mathcal{Q}}$ induces a strategy $w^{\mathcal{S}}$ on $\mathcal{S}$ defined by

$$w^{\mathcal{S}}(S) = \sum_{Q \in Q(S)} w^{\mathcal{Q}}(Q). \tag{4.1}$$

The load on an element $i \in \mathcal{S}$ (i.e., the frequency of accessing the quorum system $\mathcal{R}_i$) is then $l_{w^{\mathcal{S}}}(i) = \sum_{S \ni i, S \in \mathcal{S}} w^{\mathcal{S}}(S)$. Similarly, $w^{\mathcal{Q}}$ induces a strategy on each copy $\mathcal{R}_i$ defined by

$$w^{\mathcal{R}_i}(R) = \left(\sum_{Q \supseteq R} w^{\mathcal{Q}}(Q) \right) \Big/ l_{w^{\mathcal{S}}}(i). \tag{4.2}$$

This $w^{\mathcal{R}_i}$ is well defined when $l_{w^{\mathcal{S}}}(i) > 0$. It is easy to verify that $w^{\mathcal{S}}$ and $w^{\mathcal{R}_i}$ are indeed strategies, i.e., that the probabilities add up to 1.

CLAIM 4.8. *Let $l_{w^{\mathcal{Q}}}(u_{ij})$ be the load induced by $w^{\mathcal{Q}}$ on an element $u_{Ij} \in \mathcal{R}_i$, and let $l_{w^{\mathcal{R}_i}}(u_{ij})$ be the load induced on it by $w^{\mathcal{R}_i}$. Then $l_{w^{\mathcal{Q}}}(u_{ij}) = l_{w^{\mathcal{S}}}(i) \cdot l_{w^{\mathcal{R}_i}}(u_{ij})$.*

Proof of Claim. Using (4.1) and (4.2) we have that

$$\begin{aligned} l_{w^{\mathcal{S}}}(i) \cdot l_{w^{\mathcal{R}_i}}(u_{ij}) = l_{w^{\mathcal{S}}}(i) \sum_{R \ni u_{ij}} w^{\mathcal{R}_i}(R) \;&=\; l_{w^{\mathcal{S}}}(i) \sum_{R \ni u_{ij}} \left(\sum_{Q \supseteq R} w^{\mathcal{Q}}(Q) \right) \Big/ l_{w^{\mathcal{S}}}(i) \\ &= \sum_{R \ni u_{ij}} \sum_{Q \supseteq R} w^{\mathcal{Q}}(Q) \;=\; \sum_{Q \ni u_{ij}} w^{\mathcal{Q}}(Q) \;=\; l_{w^{\mathcal{Q}}}(u_{ij}). \qquad \square \end{aligned}$$

To complete the proof of Theorem 4.7, assume that $w^{\mathcal{Q}}$ is an optimal strategy for $\mathcal{Q}$. Consider the copy $\mathcal{R}_i$ for which $l_{w^{\mathcal{S}}}(i)$ is maximal, i.e., $\mathcal{L}_{w^{\mathcal{S}}}(\mathcal{S}) = l_{w^{\mathcal{S}}}(i)$, and let u_{ij} be the maximally loaded element in this $\mathcal{R}_i$. Clearly $l_{w^{\mathcal{S}}}(i) > 0$ so $w^{\mathcal{R}_i}$ is well defined for this i. Note that we do not require u_{ij} to be the maximally loaded element in all of $\mathcal{Q}$. Using the claim and the minimality of $\mathcal{L}(\mathcal{S})$ and $\mathcal{L}(\mathcal{R})$ we obtain that

$$\begin{aligned} \mathcal{L}(\mathcal{Q}) = \mathcal{L}_{w^{\mathcal{Q}}}(\mathcal{Q}) \;&\geq\; l_{w^{\mathcal{Q}}}(u_{ij}) \;=\; l_{w^{\mathcal{S}}}(i) \cdot l_{w^{\mathcal{R}_i}}(u_{ij}) \\ &= \mathcal{L}_{w^{\mathcal{S}}}(\mathcal{S}) \cdot \mathcal{L}_{w^{\mathcal{R}_i}}(\mathcal{R}) \;\geq\; \mathcal{L}(\mathcal{S})\mathcal{L}(\mathcal{R}). \end{aligned}$$

By combining this inequality with the upper bound we had before we conclude that $\mathcal{L}(\mathcal{Q}) = \mathcal{L}(\mathcal{S})\mathcal{L}(\mathcal{R})$. $\square$

The multiplicative behavior of the combinatorial parameters in composing quorum systems provides a powerful tool for "boosting" existing constructions into larger systems with possibly improved characteristics. Below, we use quorum composition in two cases and demonstrate that this technique yields improved constructions over their basic building blocks, for appropriately larger system sizes. In particular, in section 6 we show a composition that allows us to transform any regular quorum construction into a (larger) b-masking quorum system.

5. Simple systems.

In this section we show two types of constructions, M-Grid and RT. These systems significantly improve upon the original constructions of [25]; however, both are still suboptimal in some parameter: M-Grid has optimal load but can mask only up to $b = O(\sqrt{n})$ failures and has poor crash probability; and RT can mask up to $b = O(n)$ failures and has near-optimal crash probability but has suboptimal load.

In sections 6 and 7 we present systems which are superior to the M-Grid and RT. Nonetheless, we feel that the simplicity of the M-Grid and RT systems, and the fact that they are suitable for very small universe sizes, are what makes them appealing.

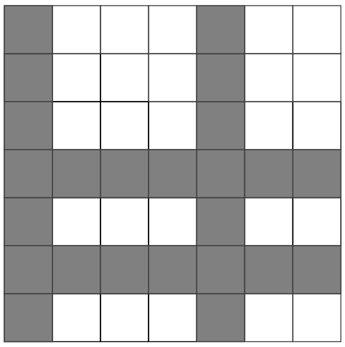


FIG. 1. *The multigrid construction, $n = 7 \times 7, b = 3$, with one quorum shaded.*

5.1. The M-Grid system. We begin with the M-Grid system, which achieves an optimal load among b-masking quorum systems, where $b \leq (\sqrt{n} - 1)/2$. The idea of the construction is as follows. Arrange the elements in a $\sqrt{n} \times \sqrt{n}$ grid. A quorum in an M-Grid consists of any choice of $\sqrt{b+1}$ rows and $\sqrt{b+1}$ columns, as shown in Figure 1. Formally, denote the rows and columns of the grid by R_i and C_i, respectively, where $1 \leq i \leq \sqrt{n}$. Then, the quorum system is

$$\text{M-Grid}(b) = \left\{ \bigcup_{j \in J} C_j \cup \bigcup_{i \in I} R_i : J, I \subseteq \{1 \dots \sqrt{n}\}, |J| = |I| = \sqrt{b+1} \right\}.$$

PROPOSITION 5.1. *The multigrid* M-Grid(b) *is a b-masking quorum system for* $b \leq (\sqrt{n} - 1)/2$.

Proof. Consider two quorums $R, S \in$ M-Grid(b). If they have either a row or a column in common, then $|R \cap S| \geq \sqrt{n} \geq 2b + 1$ and we are done. Otherwise the intersection of S's columns with R's rows is disjoint from the intersection of R's columns with S's rows, so $|R \cap S| \geq 2\sqrt{b+1}\sqrt{b+1} > 2b + 1$. Therefore consistency holds.

Resilience holds since $f = \mathcal{MT}(\text{M-Grid}(b)) - 1 = \sqrt{n} - \sqrt{b+1} \geq b$. Therefore $\mathcal{MT}(\text{M-Grid}(b)) \geq b + 1$, and Lemma 3.6 finishes the proof. □

PROPOSITION 5.2. $\mathcal{L}(\text{M-Grid}(b)) \approx 2\sqrt{\frac{b+1}{n}}$.

Proof. Since M-Grid(b) is fair we can use Proposition 3.9 to get $\mathcal{L}(\text{M-Grid}(b)) = c(\text{M-Grid}(b))/n$. □

Remark. The load of M-Grid(b) is within a factor of $\sqrt{2}$ from the optimal load which can be achieved for $b \approx \sqrt{n}/2$.

A disadvantage of the M-Grid system is its poor asymptotic crash probability. If crashes occur with some constant probability p, then any configuration of crashes with at least one crash per row disables the system. Therefore, as shown by [20, 40],

$$F_p(\text{M-Grid}) \geq (1 - (1-p)^{\sqrt{n}})^{\sqrt{n}} \underset{n \to \infty}{\longrightarrow} 1.$$

5.2. RT systems. An RT system RT(k, ℓ) of depth h is built by taking a simple building block, which is an ℓ-of-k threshold system (with $k > \ell > k/2$), and recursively composing it over itself to depth h. In what follows, we often omit the depth parameter h when it has no effect on the discussion. The RT systems generalize the recursive

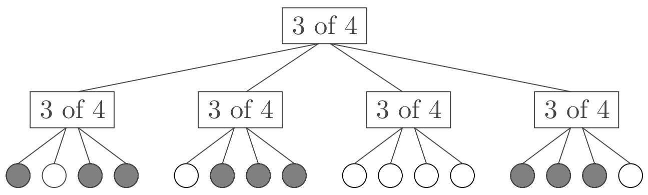


FIG. 2. *An* RT(4, 3) *system of depth $h = 2$, with one quorum shaded.*

majority constructions of [29], the HQC system of [19] is an RT(3, 2) system, and in fact the threshold system of [25] can be viewed as a trivial RT$(4b+1, 3b+1)$ system with depth $h = 1$. As an example throughout this section we will use the RT(4, 3) system, depicted in Figure 2.

PROPOSITION 5.3. *An* RT(k, ℓ) *system of depth h is a fair quorum system, with $n = k^h$ elements, quorums of size* $c(\mathrm{RT}(k,\ell)) = \ell^h$, *intersection size of* $\mathcal{IS}(\mathrm{RT}(k,\ell)) = (2\ell - k)^h$, *and minimal transversals of size* $\mathcal{MT}(\mathrm{RT}(k,\ell)) = (k - \ell + 1)^h$.

Proof. The basic ℓ-of-k system is symmetric (and therefore fair), with $c(\ell\text{-of-}k) = \ell$, $\mathcal{MT}(\ell\text{-of-}k) = k - \ell + 1$, and $\mathcal{IS}(\ell\text{-of-}k) = 2\ell - k$. The combinatorial parameters are computed by activating Theorem 4.7 h times and the composition preserves the fairness. □

Plugging this into Corollary 3.7 we obtain the following.

COROLLARY 5.4. *An* RT(k, ℓ) *system over a universe of size n is a b-masking quorum system for*

$$b = \min\{(n^{\log_k(2\ell-k)} - 1)/2, n^{\log_k(k-\ell+1)} - 1\}. \qquad \square$$

In the 3-of-4 example we have $\mathcal{IS}(\text{3-of-4}) = \mathcal{MT}(\text{3-of-4}) = 2$ and $c(\text{3-of-4}) = 3$. Therefore for the whole system (to depth $\log_4 n$) we get $c(\mathrm{RT}(4,3)) = n^{\log_4 3} = n^{0.79}$, with $\mathcal{IS}(\mathrm{RT}(4,3)) = \mathcal{MT}(\mathrm{RT}(4,3)) = \sqrt{n}$ and thus $b = (\sqrt{n} - 1)/2$. Note that the basic 3-of-4 system is not even 1-masking since intersections of size 2 are too small; however, already from $h = 2$ (i.e., $n = 16$) we obtain a masking system.

PROPOSITION 5.5. *The load* $\mathcal{L}(\mathrm{RT}(k,\ell)) = n^{-(1-\log_k \ell)}$.

Proof. Since RT(k, ℓ) is fair we can use Proposition 3.9 to get $\mathcal{L}(\mathrm{RT}(k,\ell)) = c(\mathrm{RT}(k,\ell))/n$. □

Remark. In general the load is suboptimal for this construction. For instance, in the RT(4, 3) system we obtain $\mathcal{L}(\mathrm{RT}(4,3)) = n^{-0.21}$. However for $b = (\sqrt{n} - 1)/2$ we could hope for a load of $\sqrt{(2b+1)/n} = n^{-0.25}$.

PROPOSITION 5.6. *There exists a unique critical probability* $0 < p_c < 1/2$ *for which*

$$\lim_{h\to\infty} F_p(\mathrm{RT}(k,\ell) \text{ of depth } h) = \begin{cases} 0, & p < p_c, \\ 1, & p > p_c. \end{cases}$$

Proof. Let $g(p)$ be the crash probability function of the ℓ-of-k system and let $F(h) = F_p(\mathrm{RT}(k,\ell) \text{ of depth } h)$ denote the crash probability for the RT(k, ℓ) system of depth h. Then $F(h)$ obeys the recurrence

$$F(h) = \begin{cases} g(F(h-1)), & h \geq 1, \\ p, & h = 0. \end{cases} \tag{5.1}$$

Now $g(p)$ is a reliability function, and therefore it is "S-shaped" (see [5]). This implies that there exists a unique critical probability $0 < p_c < 1$ for which $g(p_c) = p_c$, such that $g(p) < p$ when $p < p_c$ and $g(p) > p$ when $p > p_c$ (and [34] shows that for quorum systems such as RT in fact $p_c < 1/2$). Therefore if $p < p_c$, then repeated applications of recurrence (5.1) would decrease $F(h)$ arbitrarily close to 0, and when $p > p_c$ the limit is 1. ◻

PROPOSITION 5.7. *If $p < 1/\binom{k}{\ell-1}$ and $\ell < k$, then $F_p(\mathrm{RT}(k,\ell)) < \exp(-\Omega(n^{\log_k(k-\ell+1)}))$, which is optimal for systems with resilience $f = n^{\log_k(k-\ell+1)}$.*

Proof. Let $g(p)$ and $F(h)$ be as in the proof of Proposition 5.6. Any configuration of at least $k-\ell+1$ crashes disables the ℓ-of-k system, so

$$g(p) = \sum_{j=k-\ell+1}^{k} \binom{k}{j} p^j (1-p)^{k-j}.$$

By Lemma A.2 (see Appendix A) we have that

$$g(p) \le \binom{k}{\ell-1} p^{k-\ell+1}.$$

Plugging this into (5.1) gives that

$$\begin{aligned} F(h) &\le \binom{k}{\ell-1}^{1+(k-\ell+1)+\cdots+(k-\ell+1)^{h-1}} p^{(k-\ell+1)^h} \\ &< \left[\binom{k}{\ell-1} p\right]^{(k-\ell+1)^h}. \end{aligned}$$

If $p < 1/\binom{k}{\ell-1}$, then the last expression decays to zero with h, so $F_p(\mathrm{RT}(k,\ell)) < \exp(-\Omega(n^{\log_k(k-\ell+1)}))$.

The lower bound of Proposition 4.3 shows that

$$F_p(\mathrm{RT}(k,\ell)) \ge p^{n^{\log_k(k-\ell+1)}},$$

so our analysis is tight. ◻

For the $\mathrm{RT}(4,3)$ system a direct calculation shows that $g(p) = 6p^2 - 8p^3 + 3p^4$ and $p_c = 0.2324$. Therefore Proposition 5.6 guarantees that when the element crash probability is in the range $p < 0.2324$, then $F_p \to 0$ when $n \to \infty$. Furthermore, when $p < 1/6$ then Proposition 5.7 shows that the decay is rapid, with $F_p(\mathrm{RT}(4,3)) < (6p)^{\sqrt{n}}$, which is optimal.

6. boostFPP.

In this section we introduce a family of b-masking quorum systems, the *boosted finite projective planes*, which we denote by boostFPP. A boostFPP system is a composition of a finite projective plane (FPP) over a threshold system (Thresh).

The first component of a boostFPP system is an FPP of order q (a good reference on FPPs is [14]). It is known that FPPs exist for $q = p^r$ when p is prime. Such an FPP has $n_F = q^2 + q + 1$ elements and quorums of size $c(\mathrm{FPP}) = q + 1$. This is a regular quorum system, i.e., it has intersections of size $\mathcal{IS}(\mathrm{FPP}) = 1$. The minimal transversals of an FPP are of size $\mathcal{MT}(\mathrm{FPP}) = q + 1$ (in fact the only transversals of this size are the quorums themselves). The load of FPP was analyzed in [31] and shown to be $\mathcal{L}(\mathrm{FPP}) = \frac{q+1}{n_F} \approx 1/\sqrt{n_F}$, which is optimal for regular quorum systems.

The second component of a boostFPP is a Thresh system, with $n_T = 4b+1$ elements and a threshold of $3b+1$. This is a b-masking quorum system in itself, with $\mathcal{IS}(\text{Thresh}) = 2b+1$, $\mathcal{MT}(\text{Thresh}) = b+1$, and a load of $\mathcal{L}(\text{Thresh}) \approx 3/4$.

PROPOSITION 6.1. *Let* $\text{boostFPP}(q,b) = \text{FPP}(q) \circ \text{Thresh}(3b+1 \text{ of } 4b+1)$. *Then the composed system has* $n = (4b+1)(q^2+q+1)$ *elements, with quorums of size* $c(\text{boostFPP}(q,b)) = (3b+1)(q+1)$*, intersections of size* $\mathcal{IS}(\text{boostFPP}(q,b)) = 2b+1$*, and minimal transversals of size* $\mathcal{MT}(\text{boostFPP}(q,b)) = (b+1)(q+1)$*. Therefore* $\text{boostFPP}(q,b)$ *is a* b*-masking quorum system.*

Proof. We obtain the combinatorial parameters by plugging the values of the component systems into Theorem 4.7. By Corollary 3.7 we have that the system can mask $\min\{(b+1)(q+1)-1, b\} = b$ failures. $\square$

PROPOSITION 6.2. $\mathcal{L}(\text{boostFPP}(q,b)) \approx \frac{3}{4q}$*, which is optimal for* b*-masking quorum systems with* $n \approx 4bq^2$ *elements.*

Proof. $\text{boostFPP}(q,b)$ is a fair quorum system since both its components are fair, so by Proposition 3.9 we have

$$\mathcal{L}(\text{boostFPP}(q,b)) = \frac{c(\text{boostFPP}(q,b))}{n} = \frac{(3b+1)(q+1)}{(4b+1)(q^2+q+1)} \approx \frac{3}{4q}.$$

On the other hand, for b-masking systems with $n \approx 4bq^2$ elements the lower bound of Theorem 4.1 gives

$$\mathcal{L}(\text{boostFPP}(q,b)) \geq \sqrt{\frac{2b}{n}} \approx \frac{1}{\sqrt{2}q}. \qquad \square$$

Note that the optimality of the load holds for *any* choice of q and b. Therefore when the number of servers (or elements) increases, the $\text{boostFPP}(q,b)$ system can scale up using different policies while maintaining load optimality. There are two extremal policies:

(1) Fix q and increase b; then the system can mask more failures when new servers are added; however, the load on the servers does not decrease.

(2) Fix b and increase q; then the load decreases when new servers are added, but the number of failures that the system can mask remains unchanged.

It is important to note that systems of arbitrarily high resilience can be constructed using the first policy since b can be chosen independently of q. In particular, we can choose $b = q^a$ for any $a > 0$. Then the resulting system has $n \approx 4bq^2 = 4b^{\frac{a+2}{a}}$, and so $b \approx \left(\frac{n}{4}\right)^{\frac{a}{a+2}}$, thus asymptotically approaching the resilience upper bound of $\frac{n}{4}$.

Finally we analyze the crash probability of boostFPP. The following proposition shows that boostFPP has good availability as long as $p < 1/4$.

PROPOSITION 6.3. *If* $p < 1/4$*, then* $F_p(\text{boostFPP}(q,b)) \leq \exp(-\Omega(b - \log q))$.

Proof. We start by estimating $F_p(\text{Thresh})$. Let *#crashed* denote the number of crashed elements in a universe of size $4b+1$. Let $\gamma = \frac{b+1}{4b+1} - p$; thus $0 < \gamma < 1$ when $p < 1/4$. Then using the Chernoff bound we obtain

$$\begin{aligned} F_p(\text{Thresh}) &= \mathbb{P}(\#\textit{crashed} \geq b+1) \\ &= \mathbb{P}(\#\textit{crashed} \geq (p+\gamma)(4b+1)) \\ &\leq e^{-2(4b+1)\gamma^2} \approx e^{-b(1-4p)^2/2}. \end{aligned} \tag{6.1}$$

Next we estimate $F_p(\text{FPP})$. Let $Q_0 \in \text{FPP}$ be some quorum. Then

$$F_p(\text{FPP}) = 1 - \mathbb{P}(\text{there exists } Q \in \text{FPP} : Q \text{ is alive})$$
$$\leq 1 - \mathbb{P}(Q_0 \text{ is alive}) = 1 - (1-p)^{q+1} \leq (q+1)p. \tag{6.2}$$

Using Theorem 4.7 we plug (6.1) into (6.2) to obtain

$$F_p(\text{boostFPP}(q,b)) \leq (q+1)e^{-b(1-4p)^2/2} = e^{-\Omega(b-\log q)}. \qquad \square$$

Remarks.

- In general the crash probability is not optimal; since $\mathcal{MT}(\text{boostFPP}(q,b)) \approx bq$ then the lower bound of Proposition 4.3 shows we could hope for a crash probability of $\exp(-\Omega(bq))$. Nevertheless if q is constant, then F_p is asymptotically optimal, and if $b \gg q$, then the gap between the upper and lower bounds is small.
- The final estimate we get for $F_p(\text{boostFPP})$ seems poor, as the bound is *higher* than the crash probability of the Thresh components. However, this is not an artifact of overestimates in our analysis. Rather, it is a result of the property that the crash probability of FPP is higher than p, and in fact $F_p(\text{FPP}) \to 1$ as shown by [37, 40]. In this light it is not surprising that boostFPP does not have an optimal crash probability.
- The requirement $p < 1/4$ is essential for this system; if $p > 1/4$, then in fact $F_p(\text{boostFPP}) \to 1$ as $n \to \infty$.

7. The M-Path system.

Here we introduce the construction we call the multipath system, denoted by M-Path. The elements of this system are the vertices of a triangulated square $\sqrt{n} \times \sqrt{n}$ grid; formally, the vertices are the points $\{(i,j) \in \mathbb{R}^2 : 1 \leq i,j \leq \sqrt{n};\ i,j \in \mathbb{Z}\}$. The triangulated grid has an edge between (i_1, j_1) and (i_2, j_2) if one of the following three conditions holds: (i) $i_1 = i_2$ and $j_2 = j_1 + 1$; (ii) $j_1 = j_2$ and $i_2 = i_1 + 1$; (iii) $i_2 = i_1 - 1$ and $j_2 = j_1 + 1$. A quorum in the M-Path system consists of $\sqrt{2b+1}$ disjoint paths from the left side to the right side of the grid (LR paths) and $\sqrt{2b+1}$ disjoint top-bottom (TB) paths (see Figure 3).

The M-Path system has several characteristics similar to the basic M-Grid system of section 5, namely an ability to mask $b = O(\sqrt{n})$ failures and optimal load. Its major advantage is that it also has an optimal crash probability F_p. Moreover, it is the only construction we have for which $F_p \to 0$ as $n \to \infty$ when the individual crash probability p is arbitrarily close to $1/2$. We are able to prove this behavior of F_p using results from percolation theory [18, 13].

Remark. The system we present here is based on a triangular lattice, with elements corresponding to vertices, as in [41, 6]. We have also constructed a second system which is based on the square lattice with elements corresponding to the edges, as in [31]. The properties of this second system are almost identical to those of M-Path, so we omit it.

PROPOSITION 7.1. *M-Path(b) has minimal quorums of size $c(\text{M-Path}) \leq 2\sqrt{n(2b+1)}$, minimal intersections of size $\mathcal{IS}(\text{M-Path}) \geq 2b+1$, and minimal transversals of size $\mathcal{MT}(\text{M-Path}) = \sqrt{n} - \sqrt{2b+1} + 1$. Therefore M-Path is a b-masking quorum system for $b \leq \sqrt{n} - \sqrt{2}n^{1/4}$.*

Proof. Let $Q_1, Q_2 \in \text{M-Path}(b)$. Then the $\sqrt{2b+1}$ LR paths of Q_1 intersect the $\sqrt{2b+1}$ TB paths of Q_2 in $\geq 2b+1$ elements, since the LR and TB paths are disjoint. As in the M-Grid system we have that $\mathcal{MT}(\text{M-Path}(b)) = \sqrt{n} - \sqrt{2b+1} + 1$, so when $b \leq \sqrt{n} - \sqrt{2}n^{1/4}$ it follows that $\mathcal{MT}(\text{M-Path}(b)) \geq b+1$ and we are done. $\square$

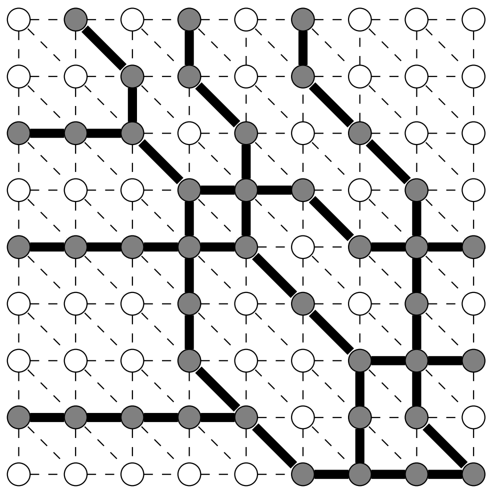


FIG. 3. *A multipath construction on a 9×9 grid, $b = 4$, with one quorum shaded.*

PROPOSITION 7.2. *$\mathcal{L}(\text{M-Path}(b)) \le 2\sqrt{\frac{2b+1}{n}}$, which is optimal.*

Proof. The strategy only uses straight line LR and TB paths. It picks $\sqrt{2b+1}$ of the $\sqrt{n}$ rows uniformly at random and likewise for the columns. Clearly the load equals the probability of accessing some element in position i, j, which is

$$\begin{aligned}\mathcal{L}(\text{M-Path}) &\le \mathbb{P}(\text{row } i \text{ chosen}) + \mathbb{P}(\text{column } j \text{ chosen})\\ &\le 2\binom{\sqrt{n}-1}{\sqrt{2b+1}-1}\Big/\binom{\sqrt{n}}{\sqrt{2b+1}}\\ &= 2\sqrt{2b+1}/\sqrt{n}.\end{aligned}$$

By Corollary 4.2 this is optimal. □

PROPOSITION 7.3. *$F_p(\text{M-Path}(b)) \le \exp(-\Omega(\sqrt{n} - \sqrt{b}))$ for any $p < 1/2$, which is optimal for systems with resilience $f = O(\sqrt{n} - \sqrt{b})$.*

Proof. We use the notation $\mathbb{P}_p(\mathcal{E})$ to denote the probability of event $\mathcal{E}$ defined on the grid when the individual crash probability is p. A path is called "open" if all its elements are alive.

Let LR be the event "there exists an open LR path in the grid," and let LR_k be the event "there exist k open LR paths." A failure configuration in M-Path(b) is one in which either $\sqrt{2b+1}$ open LR paths or $\sqrt{2b+1}$ open TB paths do not exist. By symmetry we have that

$$F_p(\text{M-Path}(b)) \le 2\mathbb{P}_p(\overline{LR_{\sqrt{2b+1}}}) = 2(1 - \mathbb{P}_p(LR_{\sqrt{2b+1}})). \tag{7.1}$$

Fix some p' such that $p < p' < 1/2$. Then by Theorem B.3 (see Appendix B) we have that

$$1 - \mathbb{P}_p(LR_{\sqrt{2b+1}}) \le \left(\frac{1-p}{p'-p}\right)^{\sqrt{2b+1}-1} [1 - \mathbb{P}_{p'}(LR)]. \tag{7.2}$$

TABLE 2
Constructions in this paper (n = number of servers).

System	$b <$	f	$\mathcal{L}$	F_p
Threshold [25]	$n/4$	$O(n-b)$	$1/2 + O(b/n)$	$\exp(-\Omega(f))$ *
Grid [25]	$\sqrt{n}/3$	$O(\sqrt{n}-b)$	$O(b/\sqrt{n})$	$\underset{n\to\infty}{\longrightarrow} 1$
M-Grid	$\sqrt{n}/2$	$O(\sqrt{n}-\sqrt{b})$	$O(\sqrt{b/n})$ †	$\underset{n\to\infty}{\longrightarrow} 1$
RT(k, ℓ)‡	$O(\min\{n^{\alpha_1}, n^{\alpha_2}\})$ ‡	$O(b)$	$n^{-(1-\log_k \ell)}$	$\exp(-\Omega(f))$ *
boostFPP	$n/4$	$O(\sqrt{bn})$	$O(\sqrt{b/n})$ †	$\exp(-\Omega(b - \log(n/b)))$
M-Path	$(1-o(1))\sqrt{n}$	$O(\sqrt{n}-\sqrt{b})$	$O(\sqrt{b/n})$ †	$\exp(-\Omega(f))$ *

† Optimal for b-masking systems
* Optimal for f-resilient systems
‡ $\alpha_1 = \log_k(2\ell - k)$ and $\alpha_2 = \log_k(k - \ell + 1)$

Plugging the bound on $\mathbb{P}_{p'}(LR)$ from Theorem B.1 into (7.2) and (7.1) yields

$$\begin{aligned} F_p(\text{M-Path}(b)) &\le 2\left(\frac{1-p}{p'-p}\right)^{\sqrt{2b+1}-1} e^{-\psi(p')\sqrt{n}} \\ &= 2e^{-\psi(p')\sqrt{n}+(\sqrt{2b+1}-1)\ln\left(\frac{1-p}{p'-p}\right)} \end{aligned}$$

for some function $\psi(p') > 0$. Now $\sqrt{2b+1} = O(n^{1/4})$, so for large enough n we can certainly write

$$F_p(\text{M-Path}(b)) \le \exp(-\Omega(\sqrt{n} - \sqrt{b})).$$

This is optimal by Proposition 4.3. □

8. Discussion. We have presented four novel constructions of b-masking quorum systems. For the first time in this context, we considered the resilience of such systems to crash failures in addition to their tolerance of (possibly fewer) Byzantine failures. Each of our constructions is optimal in either its load or its crash probability (for sufficiently small p). Moreover, one of our constructions, namely M-Paths, is optimal in both measures. One of our constructions is achieved using a novel boosting technique that makes all known benign fault-tolerant quorum constructions available for Byzantine environments (of appropriate sizes). In proving optimality of our constructions, we also contribute lower bounds on the load and crash probability of any b-masking quorum system.

The properties of our various constructions are summarized in Table 2, alongside the properties of two other b-masking constructions proposed in [25], namely Threshold and Grid.

Determining the best quorum construction depends on the goals and constraints of any particular settings, as no system is advantageous in all measures. For example, suppose we fix n to be 1024, the desired load $\mathcal{L}$ to be approximately 1/4, and assume that the individual failure probability of components is 1/8. In these settings, an M-Grid system can tolerate $b = 15$ Byzantine failures and up to $f = 28$ benign failures, but it has a failure probability $F_p \ge 0.638$. In the same settings, a boostFPP system (with $n = 1001$, $q = 3$) can tolerate $b = 19$, up to $f = 79$ benign failures, with somewhat better failure probability: it has $F_p \le 0.372$. The M-Path construction, with four LR and four TB paths per quorum, has $b = 7$ here and can tolerate up to $f = 29$ benign failures, but it has a good crash probability: $F_p \le 0.001$ (using the estimate following Theorem B.1, together with Theorem B.3 with $p' = 1/7$). In this setting, the RT$(4,3)$ construction, with depth $h = 5$, is the best, with $b = 15$, $f = 31$, and an excellent failure probability of only $F_p \le 0.0001$.

More generally, if masking large numbers of Byzantine server failures is important, then of the systems listed in Table 2, only Threshold and boostFPP can provide the highest possible masking ability, i.e., up to $b < n/4$. However, Threshold can mask $n/4$ Byzantine failures for any system size, whereas boostFPP approaches such degree of Byzantine resilience only for very large n. If, on the other hand, load is more crucial, then Threshold suffers in load whereas boostFPP offers reduced load, as do the other three systems in this paper, albeit with lower masking ability. If masking fewer Byzantine server failures is allowable, then other quorum constructions can be used, in particular RT and M-Path. These two constructions have similar masking ability, resilience, and load, but M-Path has asymptotically superior crash probability when p is close to $1/2$.

Finally, we note that it is impossible to achieve optimal resilience and load simultaneously: since necessarily $f \le c(\mathcal{Q})$, Theorem 4.1 implies that $f \le n\mathcal{L}(\mathcal{Q})$, i.e., when load is low then so is resilience, and when resilience is high then so is load. In order to break this trade-off, in [28] we propose relaxing the intersection property of masking quorum systems, so that "quorums" chosen according to a specific strategy intersect each other in enough correct servers to maintain correctness of the system with a high probability.

Appendix A. Combinatorial lemmas.

Lemma A.1. *Let $0 \le i, d \le k$ be integers. Then* $\frac{\binom{k}{d+i}}{\binom{k}{d}} \le \binom{k-d}{i}$.

Proof.

$$\frac{\binom{k}{d+i}}{\binom{k}{d}} = \frac{k!d!(k-d)!}{(d+i)!(k-d-i)!k!} = \frac{(k-d)!}{(k-d-i)!}\frac{d!}{(d+i)!} \le \frac{(k-d)!}{(k-d-i)!i!} = \binom{k-d}{i}. \qquad \square$$

Lemma A.2. *Let $0 \le d \le k$ be integers and let $p \in [0,1]$. Then*

$$\sum_{j=d}^{k} \binom{k}{j} p^j (1-p)^{k-j} \le \binom{k}{d} p^d.$$

Proof.

$$\sum_{j=d}^{k} \binom{k}{j} p^j (1-p)^{k-j} = \binom{k}{d} p^d \sum_{j=d}^{k} \frac{\binom{k}{j}}{\binom{k}{d}} p^{j-d}(1-p)^{k-j},$$

so it suffices to show that the last sum is ≤ 1. But using Lemma A.1 we get

$$\sum_{j=d}^{k} \frac{\binom{k}{j}}{\binom{k}{d}} p^{j-d}(1-p)^{k-j} = \sum_{i=0}^{k-d} \frac{\binom{k}{d+i}}{\binom{k}{d}} p^i (1-p)^{k-d-i}$$
$$\le \sum_{i=0}^{k-d} \binom{k-d}{i} p^i (1-p)^{k-d-i} = [p + (1-p)]^{k-d} = 1. \qquad \square$$

Appendix B. Theorems of percolation theory. In this section we list the definitions and results that are used in our analysis of the M-Path system, following [18, 13].

The percolation model we are interested in is as follows. Let $\mathbb{Z}$ be the graph of the (infinite) triangle lattice in the plane. Assume that a vertex is *closed* with probability p and *open* with probability $1 - p$, independently of other vertices. This model is known as site percolation on the triangle lattice. Another natural model, which plays a minor role in our work, is the bond percolation model. In it the *edges* are closed with probability p.

A key idea in percolation theory is that there exists a *critical probability*, p_c, such that graphs with $p < p_c$ exhibit qualitatively different properties than graphs with $p > p_c$. For example, $\mathbb{Z}$ with $p < p_c$ has a single connected (open) component of infinite size. When $p > p_c$ there is no such component. For site percolation on the triangle $p_c = 1/2$ [17].

The following theorem shows that when the probability p for a closed vertex is below the critical probability, the probability of having long open paths tends to 1 exponentially fast. Recall that LR is the event "there exists an open LR path in the $\sqrt{n} \times \sqrt{n}$ grid." Then [30] (see also [13, p. 287]) implies the following.

THEOREM B.1. *If $p < 1/2$, then $\mathbb{P}_p(LR) \geq 1 - e^{-\psi(p)\sqrt{n}}$, for some $\psi(p) > 0$ independent of n.*

Remark. The dependence of ψ on p is such that $\psi(p) \to 0$ when $p \to 1/2$. However, for p's not too close to $1/2$ we can obtain concrete estimates using elementary techniques. For instance, a counting argument similar to that of Bazzi [6] shows that

$$\mathbb{P}_p(LR) \geq 1 - \frac{\sqrt{n}(3p)^{\sqrt{n}}}{1 - 3p},$$

when $p < 1/3$.

DEFINITION B.2. *Let $\mathcal{E}$ be an event defined in the percolation model. Then the* interior *of $\mathcal{E}$ with depth r, denoted $I_r(\mathcal{E})$, is the set of all configurations in $\mathcal{E}$ which are still in $\mathcal{E}$ even if we perturb the states of up to r vertices.*

We may think of $I_r(\mathcal{E})$ as the event that $\mathcal{E}$ occurs and is "stable" with respect to changes in the states of r or fewer vertices. The definition is useful to us in the following situation. If LR is the event "there exists an open left-right path in a rectangle D," then it follows that $I_r(LR)$ is the event "there are at least $r+1$ disjoint open left-right paths in D."

THEOREM B.3 (see [2]). *Let $\mathcal{E}$ be an increasing event and let r be a positive integer. Then*

$$1 - \mathbb{P}_p(I_r(\mathcal{E})) \leq \left(\frac{1-p}{p'-p}\right)^r [1 - \mathbb{P}_{p'}(\mathcal{E})]$$

whenever $0 \leq p < p' \leq 1$.

The theorem amounts to the assertion that if $\mathcal{E}$ is likely to occur when the crash probability is p', then $I_r(\mathcal{E})$ is likely to occur when the crash probability p is smaller than p'.

Acknowledgments. We thank Oded Goldreich and the anonymous referees of the 16th ACM Symposium on Principles of Distributed Computing for many helpful comments on an earlier version of this paper.

REFERENCES

[1] D. Agrawal and A. El-Abbadi, *An efficient and fault-tolerant solution for distributed mutual exclusion*, ACM Trans. Comput. Systems, 9 (1991), pp. 1–20.

[2] M. Aizenman, J. T. Chayes, L. Chayes, J. Fröhlich, and L. Russo, *On a sharp transition from area law to perimeter law in a system of random surfaces*, Comm. Math. Phys., 92 (1983), pp. 19–69.

[3] D. Barbara and H. Garcia-Molina, *The vulnerability of vote assignments*, ACM Trans. Comput. Systems, 4 (1986), pp. 187–213.

[4] D. Barbara and H. Garcia-Molina, *The reliability of vote mechanisms*, IEEE Trans. Comput., C-36 (1987), pp. 1197–1208.

[5] R. E. Barlow and F. Proschan, *Statistical Theory of Reliability and Life Testing*, Holt, Rinehart and Winston, New York, 1975.

[6] R. A. Bazzi, *Planar quorums*, in Proceedings of the 10th International Workshop on Dist. Algorithms, Bologna, Italy, 1996, Lecture Notes in Comput. Sci. 1151, Springer-Verlag, New York, 1996, pp. 251–268.

[7] S. Y. Cheung, M. H. Ammar, and M. Ahamad, *The grid protocol: A high performance scheme for maintaining replicated data*, IEEE Trans. Knowledge Data Engrg., 4 (1992), pp. 582–592.

[8] N. Condorcet, *Essai sur l'application de l'analyse à la probabilité des decisions rendues à la pluralite des voix*, Paris, 1785.

[9] A. El-Abbadi and S. Toueg, *Maintaining availability in partitioned replicated databases*, ACM Trans. Database Systems, 14 (1989), pp. 264–290.

[10] J. A. Garay and K. J. Perry, *A continuum of failure models for distributed computing*, in Proceedings of the 6th International Workshop on Dist. Algorithms, Lecture Notes in Comput. Sci. 647, Springer-Verlag, Berlin, New York, 1992.

[11] H. Garcia-Molina and D. Barbara, *How to assign votes in a distributed system*, J. ACM, 32 (1985), pp. 841–860.

[12] D. K. Gifford, *Weighted voting for replicated data*, in Proceedings of the 7th Annual ACM Symposium Oper. Sys. Principles, 1979, pp. 150–159.

[13] G. R. Grimmett, *Percolation*, Springer-Verlag, Berlin, New York, 1989.

[14] M. Hall, *Combinatorial Theory*, 2nd ed., John Wiley, 1986.

[15] M. Herlihy, *A quorum-consensus replication method for abstract data types*, ACM Trans. Comput. Systems, 4 (1986), pp. 32–53.

[16] R. Holzman, Y. Marcus, and D. Peleg, *Load balancing in quorum systems*, SIAM J. Discrete Math., 10 (1997), pp. 223–245.

[17] H. Kesten, *The critical probability of bond percolation on the square lattice equals $\frac{1}{2}$*, Comm. Math. Phys., 71 (1980), pp. 41–59.

[18] H. Kesten, *Percolation Theory for Mathematicians*, Birkhäuser, Boston, 1982.

[19] A. Kumar, *Hierarchical quorum consensus: A new algorithm for managing replicated data*, IEEE Trans. Comput., 40 (1991), pp. 996–1004.

[20] A. Kumar and S. Y. Cheung, *A high availability $\sqrt{n}$ hierarchical grid algorithm for replicated data*, Inform. Process. Lett., 40 (1991), pp. 311–316.

[21] A. Kumar, M. Rabinovich, and R. K. Sinha, *A performance study of general grid structures for replicated data*, in Proceedings of the 13th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 178–185.

[22] P. Lincoln and J. Rushby, *A formally verified algorithm for interactive consistency under a hybrid fault model*, in Proceedings of the 23rd IEEE Symposium on Fault-Tolerant Computing, Toulouse, France, 1993, pp. 402–411.

[23] P. Lincoln and J. Rushby, *Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model*, in Proceedings of the 9th IEEE Conference on Computer Assurance, Gaithersburg, MD, 1994, pp. 107–120.

[24] M. Maekawa, *A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems*, ACM Trans. Comput. Systems, 3 (1985), pp. 145–159.

[25] D. Malkhi and M. K. Reiter, *Byzantine quorum systems*, Distrib. Comput., 11 (1998), pp. 203–213.

[26] D. Malkhi and M. K. Reiter, *Secure and scalable replication in Phalanx*, in Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, IN, 1998, pp. 51–58.

[27] D. Malkhi and M. K. Reiter, *Survivable consensus objects*, in Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, IN, 1998, pp. 271–279.

[28] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright, *Probabilistic Byzantine Quo-*

rum Systems, Tech. Report 98.7, AT&T Research, 1998; also available online from http://www.research.att.com/library/trs/TRs/98/98.7/98.7.1.body.ps.gz.

[29] Y. Marcus and D. Peleg, *Construction Methods for Quorum Systems*, Tech. Report CS92–33, The Weizmann Institute of Science, Rehovot, Israel, 1992.

[30] M. V. Menshikov, *Coincidence of critical points in percolation problems*, Soviet Math. Dokl., 33 (1986), pp. 856–859.

[31] M. Naor and A. Wool, *The load, capacity, and availability of quorum systems*, SIAM J. Comput., 27 (1998), pp. 423–447.

[32] M. L. Neilsen, *Quorum Structures in Distributed Systems*, Ph.D. thesis, Department of Computing and Information Sciences, Kansas State University, 1992.

[33] M. L. Neilsen and M. Mizuno, *Coterie join algorithm*, IEEE Trans. Parallel Distrib. Systems, 3 (1992), pp. 582–590.

[34] D. Peleg and A. Wool, *The availability of quorum systems*, Inform. and Comput., 123 (1995), pp. 210–223.

[35] D. Peleg and A. Wool, *The availability of crumbling wall quorum systems*, Discrete Appl. Math., 74 (1997), pp. 69–83.

[36] D. Peleg and A. Wool, *Crumbling walls: A class of practical and efficient quorum systems*, Distrib. Comput., 10 (1997), pp. 87–98.

[37] S. Rangarajan, S. Setia, and S. K. Tripathi, *A fault-tolerant algorithm for replicated data management*, in Proceedings of the 8th IEEE International Conference on Data Engineering, Tempe, AZ, 1992, pp. 230–237.

[38] M. K. Reiter and K. P. Birman, *How to securely replicate services*, ACM Trans. Prog. Lang. Systems, 16 (1994), pp. 986–1009.

[39] R. H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Trans. Database Systems, 4 (1979), pp. 180–209.

[40] A. Wool, *Quorum Systems for Distributed Control Protocols*, Ph.D. thesis, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1996.

[41] C. Wu and G. G. Belford, *The triangular lattice protocol: A highly fault tolerant protocol for replicated data*, in Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems, 1992, pp. 66–73.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1907–1938

AN ONLINE ALGORITHM FOR IMPROVING PERFORMANCE IN NAVIGATION[*]

AVRIM BLUM[†] AND PRASAD CHALASANI[‡]

Abstract. We consider the following scenario. A point robot is placed at some start location s in a 2-dimensional scene containing oriented rectangular obstacles. The robot must repeatedly travel back and forth between s and a second location t in the scene. The robot knows the coordinates of s and t but initially knows nothing about the positions or sizes of the obstacles. It can only determine the obstacles' locations by bumping into them. We would like an intelligent strategy for the robot so that its trips between s and t both are relatively fast initially and improve as more trips are taken and more information is gathered.

In this paper we describe an algorithm for this problem with the following guarantee: in the first $k \leq n$ trips, the average distance per trip is at most $O(\sqrt{n/k})$ times the length of the shortest s-t path in the scene, where n is the Euclidean distance between s and t. We also show a matching lower bound for deterministic strategies. These results generalize known bounds on the one-trip problem. Our algorithm is based on a novel method for making an optimal trade-off between search effort and the goodness of the path found. We improve this algorithm to a "smooth" variant having the property that for *every* $i \leq n$, the robot's ith trip length is $O(\sqrt{n/i})$ times the shortest s-t path length.

A key idea of this paper is a method for analyzing obstacle scenes using a tree structure that can be defined based on the positions of the obstacles.

Key words. online algorithms, robot path planning, exploration vs. exploitation, learning, navigation, unfamiliar terrain, competitive analysis, lower bounds

AMS subject classification. 68Q25

PII. S0097539795290593

1. Introduction.

This paper addresses an abstraction of the following type of scenario. Imagine you have just moved to a new city. You are at your home and must travel to your office, but you do not have a map (let's assume you know the coordinates of your office; you just do not know the street layout). Several papers in recent literature have discussed strategies that can be used to plan one's route in this type of situation so that the distance traveled is not too much longer than the shortest path. But now, suppose you have reached your office, spent the day there, and it is time to go home. You could retrace your path, but you now have some information about the city (what you saw on your way to work in the morning) and would like to do better. The next morning you have even more information and so on. What is a strategy that allows your path taken each time to be good and to improve with experience? Perhaps you might even design your paths explicitly so as to gain more information for future trips.

Specifically, we consider the scenario (examined in [17, 7, 11, 10]) where there is a start point s and target t in a 2-dimensional plane filled with nonoverlapping, axis-parallel rectangular obstacles, having corners at integral coordinates. A point robot begins at s, and knows its current position and that of the target, but it does *not*

[*]Received by the editors August 25, 1995; accepted for publication (in revised form) June 22, 1999; published electronically April 11, 2000. This material is based upon work supported under NSF National Young Investigator grant CCR-9357793 and a Sloan Foundation Research Fellowship.
http://www.siam.org/journals/sicomp/29-6/29059.html

[†]School of Computer Science, Carnegie–Mellon University, Pittsburgh, PA 15213 (avrim@theory.cs.cmu.edu).

[‡]HBK Investments, 12 E. 49th Street, Floor 21, New York, NY 10017 (pchalasani@yahoo.com).

know the positions and extents of the obstacles; it only discovers their existence as it bumps into them. In the problem considered in previous papers, the robot's goal is to travel from s to t as quickly as possible. We call this the *one-trip* problem. For this problem, if n is the Euclidean s-t distance, [7] presents an algorithm that guarantees an $O(\sqrt{n})$ ratio of the distance traveled to the shortest path length, which is known to be optimal for deterministic algorithms [17]. Here, we consider the situation where the robot may be asked to make *multiple* trips between s and t. We would like an intelligent strategy for the robot so that its trips between s and t both are as fast as can be hoped for initially and improve as more trips are made and more information is gathered. For instance, after making one trip that achieves the above $O(\sqrt{n})$ ratio, the robot has some partial information about the scene. Can it exploit this information to improve its ratio on the second trip? Can it continue to exploit new information gained on future trips? It is important to note that partial information may not help if it is somehow not sufficiently relevant. Thus the challenge is to perform as well as possible on each trip given the information gained so far and at the same time acquire information that will be useful for improving on later trips. This makes the multitrip problem more difficult than the single trip problem.

The multitrip problem has aspects of both a machine learning and an online algorithms problem. As in machine learning settings, we would like our algorithm to improve its performance with experience. As in standard online algorithms settings (e.g., [16]), decisions the robot makes now may affect the costs it experiences in the future. However, our scenario also exhibits key differences. In particular, unlike typical online algorithms problems, here the algorithm may have *partial information* about the future—namely, the positions of obstacles that lie ahead that it has already encountered. There is also a value associated with information gathering in our setting in the sense that such information may (or may not) prove to be useful on the future trips made. One contribution of our work is a method for analyzing problems of this sort and quantifying the information that is most relevant in this setting.

We study the case of oriented rectangular obstacles for two main reasons. First, scenes containing such obstacles are complex enough to embody many of the strategic issues that arise in path planning. For example, there is the question of when one should "give up" on a difficult region in the scene and move to a new region that might be more promising, and there is also the question of which information is worth gathering. Second, if one allows arbitrarily shaped obstacles, it is known [7] that one cannot perform much better in the worst case than a simpleminded depth-first-search strategy. Thus, such scenes do not allow one to demonstrate theoretically the value of a useful approach by the performance guarantees achieved.

An extended abstract of this paper appears as [6].

1.1. Results and goodness measures. Given the basic scenario described above, the first question to be addressed is the measure of success to use. Clearly we do not want to give high marks to a solution in which the robot makes an artificially long first trip and then subsequently "improves" on future trips. Instead we would like an algorithm that performs as well as possible at all times. For this reason, we will analyze our algorithms using a type of "competitive analysis." The idea of competitive analysis is to compare the performance of one's algorithm to the best one could hope to do if there were no missing information (in our case, if a map of the scene were known). For instance, for the one-trip problem, Papadimitriou and Yannakakis [17] showed that for any deterministic algorithm and integer $n > 0$, there exist scenes having Euclidean s-t distance n (the width of the thinnest obstacle is taken as 1 unit), forcing the algorithm to travel $\Omega(\sqrt{n})$ times the length of the shortest path. Subsequently, Blum, Raghavan, and Schieber [7] showed an algorithm (BRS algorithm) having a performance guarantee that matches this lower bound.

For the multitrip problem we consider two similar measures of performance. In the *cumulative* measure, we compare the total distance traveled on the first k trips to the length of the optimal path. Our first main result is a deterministic strategy having the property that given $k \leq n$, it guarantees that the total distance traveled in the first k trips is at most $O(L\sqrt{nk})$, where L is the length of the shortest s-t path. (If $k \geq n$ the distance becomes $O(Lk)$.) We also show that up to constant factors this is the best guarantee achievable by a deterministic strategy. In particular, for any deterministic strategy and any n and $k \leq n$, there exist scenes which force the strategy to travel distance $\Omega(L\sqrt{nk})$ on the first k trips. One problem with the cumulative measure is that it does not force the algorithm to perform as well as possible on *each* trip. For this reason, we also consider a *per-trip* measure in which we separately bound the cost of each trip. Our second main result is an improvement on the cumulative algorithm having the property that for *all* $i \leq n$, the ith trip of the robot has length at most $O(L\sqrt{n/i})$. This is optimal in the sense that (up to constant factors) it meets the cumulative lower bound simultaneously for all i.

1.2. Main ideas and the basic strategy. The core of our results (and the bulk of the paper) is a method for achieving a smooth *search-quality trade-off*: smoothly trading off in a single trip the exploration cost with the goodness of the path found. Specifically, we design an algorithm that, given any $k \leq n$, searches a distance $O(L\sqrt{nk})$ and finds an s-t path of length at most $O(L\sqrt{n/k})$. In other words, at a cost of only t times the cost of the BRS algorithm ($t = \sqrt{k}$ in our case) we find a path that is a factor of t better than the BRS guarantee. In addition, our method for achieving this trade-off has the property that it can be performed in a "piecemeal" fashion (somewhat like the piecemeal learning of [5]). In particular, the searching can be performed a little bit at a time on each trip. This latter property is what allows us to turn our cumulative algorithm into one that is more like a learning algorithm, with optimal per-trip performance. As more trips are made, better searches are performed, and cheaper paths are found for the future trips. An example of an exploratory trip achieving the desired trade-off is given in Figure 1.1.

Our main idea for achieving this search-quality trade-off is a method for analyzing an obstacle scene and determining which pieces of information are the most important. In particular, we show that a *tree* structure can be defined in the scene, where the nodes are portions of certain obstacles and the edges are short paths from a node to its children. This tree can be tailored to the search cost and path quality desired. Our search algorithm is essentially an online strategy to traverse a sequence of trees optimally, and the path found is a concatenation of specific root-to-leaf paths from each tree. Besides its use for achieving a search-quality trade-off on a single trip, the tree structure enables us to spread the search over several trips: since there is a "short" path from the root to each node, we can suspend our tree-traversal on one trip and resume the exploration on a later trip by moving quickly to the point where we stopped. The tree structure is defined formally in section 5.

1.3. Related work. Versions of the multitrip problem have been addressed in the framework of reinforcement learning. Thrun [18] describes heuristics for path improvement in scenes containing (possibly concave) obstacles and presents empirical results. Koenig and Simmons [12] consider a similar problem on graphs. In other

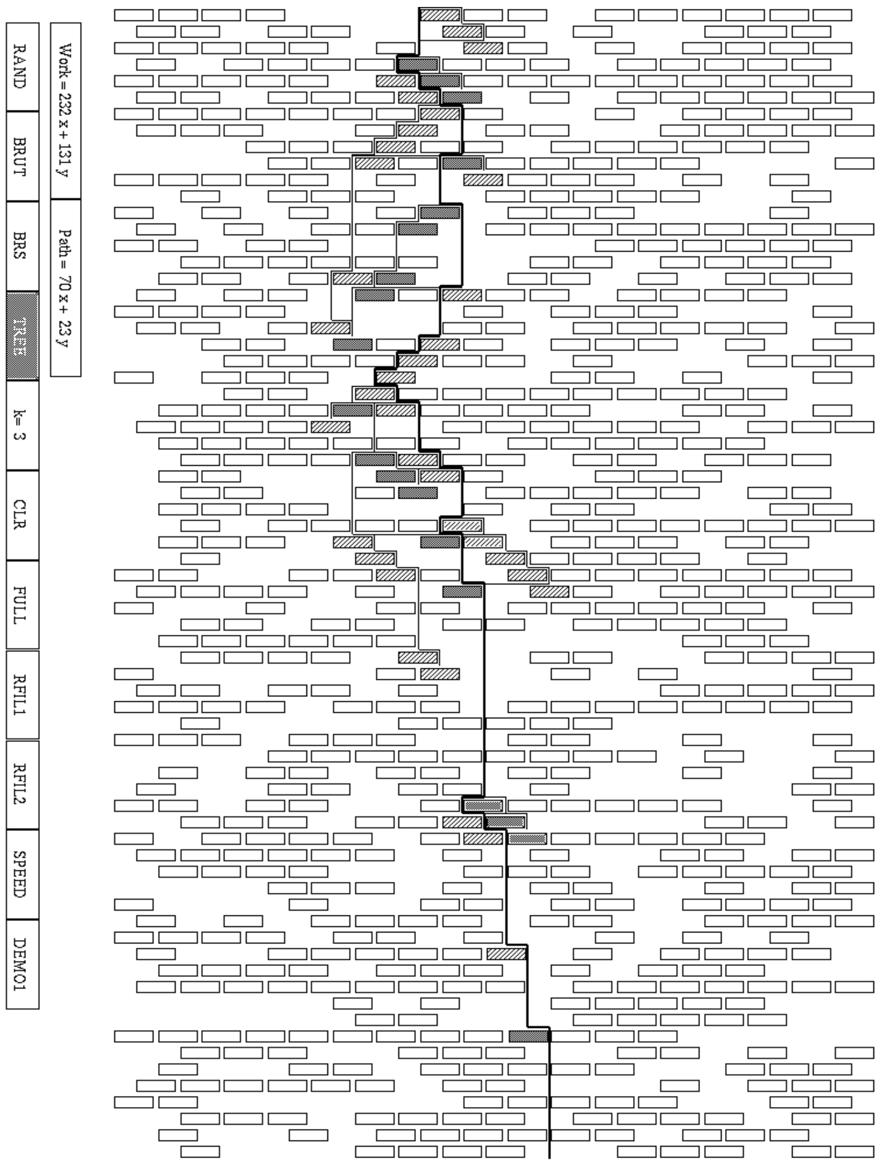


FIG. 1.1. *An example of an initial search trip, $k = 3$. The thick line shows the s-t path found (s is at center top, t is at the bottom), and the thin and thick lines are the search path. The robot occasionally will back up, which accounts for the dead ends. Obstacles hit are shaded. In each "fence group" (see section* 5.1*) fences* 1 *and* 3 *are lightly shaded and fence* 2 *is darkly shaded. This figure is a screen dump of a demonstration program that allows a user to create a "simple scene" (see section* 4*) and then run various algorithms on it.*

machine learning literature, Chen [9] considers how the computation *time* for path-planning in a *known* scene can be improved by making use of (portions of) solutions to previous path planning problems in the same scene.

Betke, Rivest, and Singh [5] consider a related problem of completely exploring an environment, but with the restriction that the robot must return to the start to refuel every d steps for some distance d. They call this *piecemeal learning* and provide algorithms for the case of a bounded region with axis-parallel rectangular obstacles.

Lumelsky [13, 14] and Lumelsky and Stepanov [15] describe some very simple algorithms that can be used to solve the one-trip problem for *arbitrary* (nonconvex) obstacles having the property that the distance traveled is at most the Euclidean s-t distance plus 1.5 times the sum of the perimeters of all the obstacles.

2. The model.

Let $\mathcal{S}(n)$ denote the class of scenes in which the Euclidean distance between s and t is n. We define s to be at the origin $(0, 0)$. As mentioned above, we assume that the width and height of each obstacle is at least 1 (this in essence defines the units of n) and for simplicity assume that the x-coordinates of the corners of obstacles are integral. Thus no more than n obstacles can be placed side by side between s and t. We assume that when obstacles touch, the point robot can move between them.

To simplify the exposition, for most of this paper we will take t to be the infinite vertical line (a "wall") $x = n$ and require the robot only to get to any point on this line; this is the wall problem of [7]. Our algorithms are easily extended to the case where t is a point, using the room problem algorithms of [7] or [3], and we describe this modification in section 8.

We model the robot as having only *tactile* sensors; that is, it discovers an obstacle only when it bumps into it. It will be convenient to assume, however, that when the robot hits an obstacle, it is told which corner of the obstacle is nearest to it and how far that corner is from its current position. As in [7], our algorithms can be modified to work without this assumption with only a constant factor penalty. We describe these modifications toward the end of the paper.

Consider a robot strategy R for making k trips between s and t. Let $R_i(S)$ be the distance traveled by the robot in the ith trip, in scene S. Let $L(S)$ be the length of the shortest obstacle-free path in the scene between s and t. We define the *cumulative* k-trip competitive ratio as

$$\rho(R, n, k) = \max_{S \in \mathcal{S}(n)} \frac{R^{(k)}(S)}{kL(S)},$$

where $R^{(k)}(S) = \sum_{i=1}^{k} R_i(S)$ is the *total* distance traveled by the robot in k trips. That is, $\rho(R, n, k)$ is the ratio between the robot's average distance traveled in k trips and L. We define the per-trip competitive ratio for the ith trip as

$$\rho_i(R, n) = \max_{S \in \mathcal{S}(n)} \frac{R_i(S)}{L(S)}.$$

Given this notation, our main results can be described as follows. First, we show for any k, n, and deterministic algorithm R, that $\rho(R, n, k) = \Omega(\sqrt{n/k})$. Second, we describe a deterministic algorithm that given $k \leq n$ achieves $\rho(R, n, k) = O(\sqrt{n/k})$. Finally, we show an improvement to that algorithm that achieves $\rho_i(R, n) = O(\sqrt{n/i})$ for *all* $i \leq n$. Notice that the latter algorithm is optimal in that it matches the lower bound simultaneously for all k. I.e., $\frac{1}{k}\sum_{i=1}^{k}\sqrt{n/i} = O(\sqrt{n/k})$. The simplest of these results is the lower bound, which we describe first.

Conventions. We will use the words up, down, left, and right to mean the directions $+y, -y, -x$, and $+x$, respectively. When we say point A is above, below, behind, or ahead of a point B we will mean that A is in the $+y, -y, -x$, or $+x$

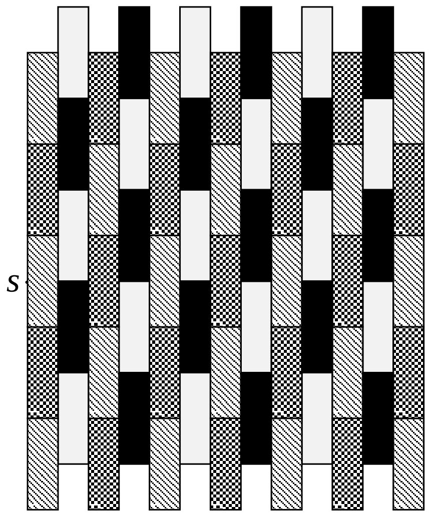


FIG. 3.1. *A 4-coloring of the brick pattern for the lower bound.*

direction, respectively, from B. Finally, vertical (respectively, horizontal) motion is parallel to the y (respectively, x) axis. At any point in time, the current coordinates of the robot (which are known to the robot) are denoted by (x, y).

3. A lower bound for k-trips.

THEOREM 3.1 (k-trip cumulative lower bound). *For $k \leq n$, the ratio $\rho(R, n, k)$ is at least $\Omega(\sqrt{n/k})$ for any deterministic algorithm R.*

Proof. Since R is deterministic, an adversary can simulate it and place obstacles in $\mathcal{S}$ as follows. Recall that s is the point $(0, 0)$.

The adversary first places obstacles of fixed height $2h \geq \sqrt{n}$ and width 1, in a full "brick pattern" on the entire plane, as shown in Figure 3.1, with s at the center of the left side of an obstacle. (Recall that the point robot can "squeeze" between bricks.) The adversary simulates R on this scene, notes which obstacles it has touched at the end of k trips, then removes all other obstacles from the scene. This is the final scene that the adversary creates for the algorithm. Let us say it contains M obstacles. The brick pattern ensures that R must have hit at least one brick at every integer x-coordinate, so $M \geq n$. Further, this arrangement forces the robot to hit a brick at every integer x-coordinate on *every* trip. Whenever it hits a brick, it must move vertically up or down a distance h, so its total k-trip distance $R^{(k)}$ is at least nkh.

We now show that there is a path from s to the wall of length at most $O(\sqrt{R^{(k)}h})$. Imagine the full brick pattern to be built out of four kinds of bricks (red, blue, yellow, and green, say) arranged in a periodic pattern as shown in the figure. This arrangement has the following property: for each color, to go from a point on an obstacle of that color to a point on any other obstacle of the same color, the robot must move a distance at least h. Out of the M obstacles hit by the robot, at least $M/4$ must have the same color, say blue. So regardless of how the robot moved, since it has visited $M/4$ blue obstacles, we have $R^{(k)} \geq Mh/4$, which implies $M \leq 4R^{(k)}/h$.

We claim there is a nonnegative integer $j \leq \sqrt{M}$ such that at most $\sqrt{M}$ obstacles have centers at the y-coordinate jh. This is because a given obstacle intersects at most one y-coordinate of the form jh, and there are M obstacles. Thus, there is a path to t that goes vertically to the y-coordinate jh, then horizontally along this y-coordinate, going around at most $\sqrt{M}$ obstacles. The total length of this path is at most $2h\sqrt{M} + 2h\sqrt{M} + n$, which is at most $6h\sqrt{M}$ since $n \leq M$ and $\sqrt{n} \leq 2h$. Since $M \leq 4R^{(k)}/h$, this path is in fact of length at most $6\sqrt{4hR^{(k)}}$. Thus the k-trip ratio is at least $R^{(k)}/(6k\sqrt{4hR^{(k)}})$. Recalling that $R^{(k)} \geq nkh$, this is at least $\frac{1}{12}\sqrt{n/k} = \Omega(\sqrt{n/k})$. □

It is not hard to see that this lower bound also holds for the case where t is a point rather than a wall.

4. The k-trip cumulative algorithm: Preliminaries.

We now give some preliminary observations needed for our algorithm.

We begin by assuming for simplicity that the algorithm knows the length L of the shortest obstacle-free path from s to t. In section 5.1 we show that this assumption can be removed by using a standard "guessing and doubling" trick. One simple observation is that the shortest obstacle-free s-t path must lie entirely within a *window* of height $2L$ centered at s, since any s-t path that leaves the window must be longer than L. *In the remainder of the paper we will refer to the rectangular region of height* $2L$ *centered vertically at* s, *and extending horizontally between* s *and* t, *as "the window."* This observation immediately leads to an easy algorithm to achieve a cumulative k-trip ratio of $O(1)$ for $k \geq n$:

First trip: Using a depth-first-search, explore the entire window. This can be done by walking a total distance of $O(Ln)$. Compute the shortest obstacle-free s-t path (of length L).

Remaining trips: Use the shortest path.

Clearly the average trip length is $O(L)$, so the cumulative n-trip ratio is $O(1)$.

Thus, the cases $k = 1$ (the BRS algorithm) and $k \geq n$ can be done with known methods. In fact, at the high level, our optimal cumulative strategy for $1 \leq k \leq n$ trips is similar to the n-trip algorithm just described:

First trip: Somehow perform an "exploratory" walk of length $O(L\sqrt{nk})$, in such a way that an s-t path P of length $O(L\sqrt{n/k})$ is discovered.

Remaining $k - 1$ trips: Use the path P.

The average trip length of this algorithm is $O(\frac{1}{k}(L\sqrt{nk}+(k-1)L\sqrt{n/k})) = O(L\sqrt{n/k})$, so the cumulative k-trip ratio is $O(\sqrt{n/k})$. Thus, as mentioned in the introduction, the key question is how to find a path that is a factor $\Omega(\sqrt{k})$ better than the BRS guarantee while traveling a distance that is only $O(\sqrt{k})$ longer.

In order to make the main ideas clear, we first describe our algorithm for a class of scenes we call *simple scenes* that capture most of the difficulties in designing online navigation algorithms (for both the one-trip and k-trip problems). In section 6 we show how to extend this algorithm to handle the general case. A scene is *simple* if (a) all obstacles have the same height $2h$ and width 1 and (b) the obstacle corners have coordinates of the form (i, jh) for integer i and j. For instance, the obstacles in the lower bound of section 3 form a simple scene. Observe that in a simple scene one can move unimpeded vertically along any integer x-coordinate without encountering any obstacles.

Notice that if $h \leq L/\sqrt{nk}$, then a brute-force strategy that moves forward when possible and otherwise arbitrarily goes around any obstacle encountered will hit at most n obstacles and therefore travel a distance at most $O(nL/\sqrt{nk}) = O(L\sqrt{n/k})$, which is our desired bound. Thus, we may assume in what follows that $h > L/\sqrt{nk}$.

Conventions. For convenience, in a simple scene we define the *position* of an obstacle A to be the coordinates of the midpoint of the left edge of the obstacle. A horizontal path whose y coordinate is a multiple of h is said to *hit* an obstacle if it hits the obstacle's center (as opposed to grazing the top or bottom edge), i.e., if it reaches the obstacle's position. We reiterate that we will use the phrase "the window" to refer to the rectangular region of height $2L$ centered vertically at s and extending horizontally between s and t. As mentioned above, we will assume in what follows that $h > L/\sqrt{nk}$.

5. The algorithm for simple scenes.

5.1. Fences and the one-trip BRS algorithm. One key notion used in both the one-trip BRS algorithm and our k-trip algorithm is that of a *fence*.[1] An *up-fence* F is a sequence of M obstacles at points $(X(1), Y(1)), (X(2), Y(2)), \ldots, (X(M), Y(M))$ such that $Y(1) \leq -L$, $Y(M) \geq L$, and for $m = 1, 2, \ldots, M-1$,

$$X(m) \leq X(m+1), \tag{5.1}$$

$$Y(m+1) = Y(m) + h. \tag{5.2}$$

See Figure 5.1. A *down-fence* has the same definition except $Y(1) \geq L$, $Y(M) \leq -L$, and (5.2) is replaced by $Y(m+1) = Y(m) - h$. The mth obstacle (counting from the left) of fence F_i is denoted by $F_i(m)$ and its coordinates are $(X_i(m), Y_i(m))$. For each m, the rectangular region of height h whose opposite corners are $(X(m), Y(m))$ and $(X(m+1), Y(m+1))$ is called a *band.* A fence can thus be viewed as a contiguous sequence of bands extending across the window.

A point P is said to be *left* (respectively, *right*) of a fence F if an imaginary horizontal line from P to the left (respectively, right) does not intersect any obstacle of F. A path is said to *cross the fence* if it connects some point left of the fence to some point right of the fence *and* stays inside the window. Any path that crosses a fence has vertical length at least h since it must completely cross some band (see Figure 5.1).

It is easy to see how a robot can find a fence with vertical motion at most $2L$. Specifically, starting from the bottom of the window, an up-fence can be found as follows:

```
    **Repeat until** at top of the window (i.e., $y = +L$): walk to the right until
    an obstacle is hit, then move up to the top of the obstacle.
```

The one-trip BRS algorithm restricted to simple scenes (and assuming L is known) reduces to the following:

```
    Initially, walk from $s$ down to the bottom of the window. Until the wall is
    reached, walk to the right, alternately building up- and down-fences across
    a window of height $2L$ centered at $s$.
```

The robot never walks backward in the BRS algorithm, so its total horizontal cost is n, and since $L \geq n$, this cost is only a small order term. Note that every obstacle hit by the robot is part of some fence. Thus every time the robot spends $2L$ (vertically) to build a fence, it is also forcing the optimal offline path to spend at least h to cross the fence. So if $h \geq L/\sqrt{n}$, the competitive ratio is $O(\sqrt{n})$. The case $h < L/\sqrt{n}$ is even easier to handle: the robot hits at most n obstacles (since they have width 1 and the robot never walks backward), so its total vertical cost is at most $nh < L\sqrt{n}$.

We say that two fences are *disjoint* if their bands do not intersect each other (see Figure 5.2). Because the bands of disjoint fences do not overlap, any path that crosses t disjoint fences must pay (vertically) at least th. For the k-trip problem an

[1]This is called a *sweep* in [7].

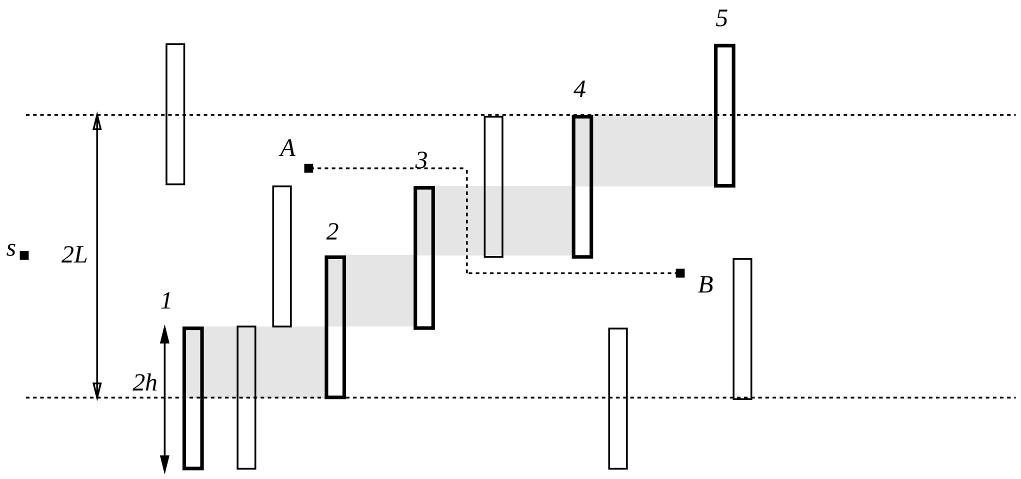


FIG. 5.1. *A fence in a simple scene. The obstacles* $\langle 1,2,3,4,5\rangle$ *with thick boundaries form a fence. The dashed line connecting points* A *and* B *crosses the fence. The shaded regions are the bands of the fence. Note: since* $L \geq n$*, the window in this figure should really be taller than its width. However for the sake of clarity, in this and all figures in this paper the vertical dimension has been compressed considerably.*

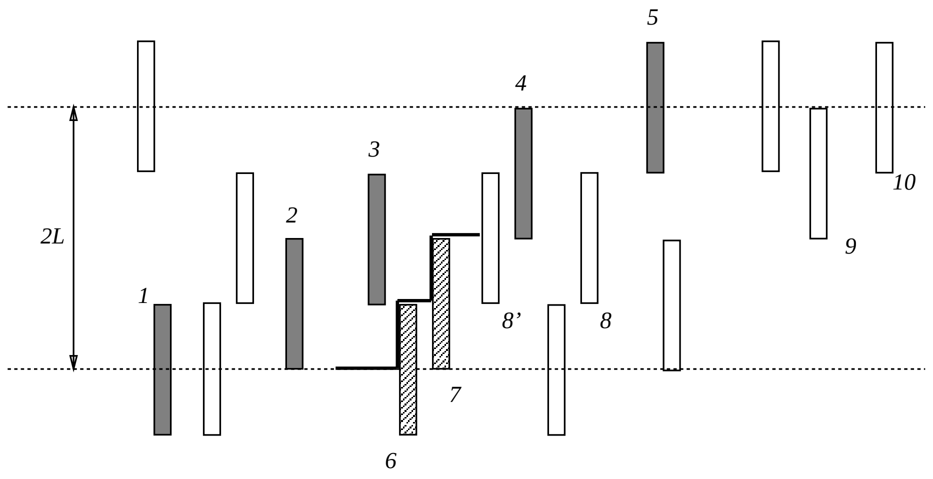


FIG. 5.2. *The fences* $F_1 = \langle 1,2,3,4,5\rangle$ *and* $F_2 = \langle 6,7,8,9,10\rangle$ *are disjoint.* F_1 *and* $F_3 = \langle 6,7,8',9,10\rangle$ *are not disjoint since the band of* F_3 *between* $8'$ *and* 9 *overlaps the band of* F_1 *between* 3 *and* 4. *In fact, one can cross both* F_1 *and* F_3 *at a total cost of only* h *by traveling between* $8'$ *and* 4. *A greedy strategy for constructing a fence such as* F_2 *disjoint from the previously found fence* F_1 *might be to go up and over obstacle* $8'$ *until obstacle* 4 *is hit, and then down around* 4 *to reach obstacle* 8. *However, this type of strategy might be expensive, as shown in Figure* 5.3.

intuitively reasonable approach is to extend the BRS idea as follows: On each trip, make new fences that are *disjoint* from previous fences. If on each trip one could find new disjoint fences "cheaply" ($O(L)$ cost) *and* one could cross old fences cheaply ($O(h)$ cost), then this would result in an optimal algorithm. However, we know of no way to find new disjoint fences this cheaply. The naive strategy of extending each new fence greedily and using previously found paths to bypass obstacles that enter

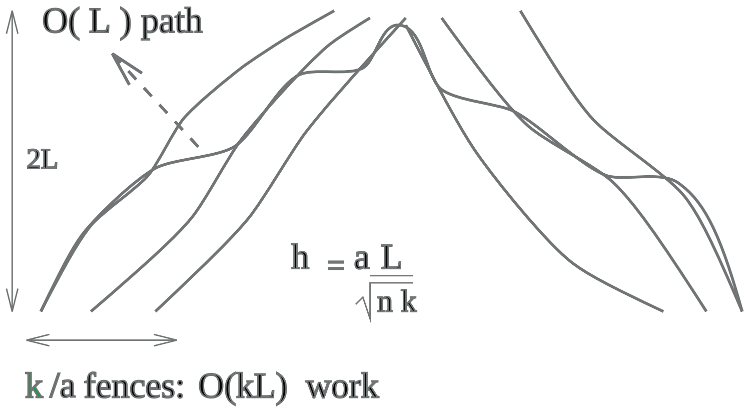


FIG. 5.3. *High-level view of the optimal k-trip strategy. First trip: create groups of fences with short group-crossing paths. Remaining trips: follow these short paths.*

existing fences can be too expensive in certain scenes. Examples of such scenes are given in [8].

Our approach, as hinted in section 4, is to give up on trying to create new disjoint fences on each trip and instead to try to find a group of disjoint fences all at once on one trip. Specifically, we do the following. Suppose that $h = aL/\sqrt{nk}$ for some $a \geq 1$ (recall that $2h$ is the obstacle height, and $a < 1$ is an easy case to handle). Then our strategy is the following.

First trip: Build a sequence of fence groups in which each group consists of $\lceil \frac{k}{a} \rceil$ disjoint fences (alternate between groups of up-fences and groups of down-fences) until the wall is reached. Ensure that
- (a) the cost of building each group is $O(kL)$,
- (b) an $O(L)$ length path crossing each group (i.e., going from the x-coordinate of the leftmost obstacle of the leftmost fence to the x-coordinate of the rightmost obstacle of the rightmost fence) is found, and
- (c) the right end of each group-crossing path is the left end of the next group-crossing path.

Remaining $k-1$ trips: Follow the group-crossing paths to the wall.

The first trip is shown schematically in Figure 5.3. To see why the above strategy achieves an $O(\sqrt{n/k})$ ratio, assuming we can somehow satisfy (a), (b), and (c), notice that the *average* online cost per trip to get past each fence group is $O(L)$ (amortizing the $O(kL)$ building cost). Crossing each group costs the optimal offline path at least $(k/a)h = L/\sqrt{n/k}$, so the average online trip length is within an $O(\sqrt{n/k})$ factor of optimal, as desired.

A search-quality trade-off. Since each fence group costs the offline optimum path at least $L/\sqrt{n/k}$ to cross, the robot will find at most $\sqrt{n/k}$ groups before reaching the wall. Thus the total length of its first trip is $O(kL\sqrt{n/k}) = O(L\sqrt{nk})$, and the total length of the group-crossing paths is $O(L\sqrt{n/k})$. Therefore, this achieves the search-quality trade-off mentioned earlier.

The doubling strategy when L is unknown. Note that if L is not known, we can just begin with a guess of $L = n$, and if the wall has not been reached after building $\sqrt{n/k}$ fence groups, we can double our guess and repeat the entire procedure. Thus there is only a constant factor penalty for not knowing L.

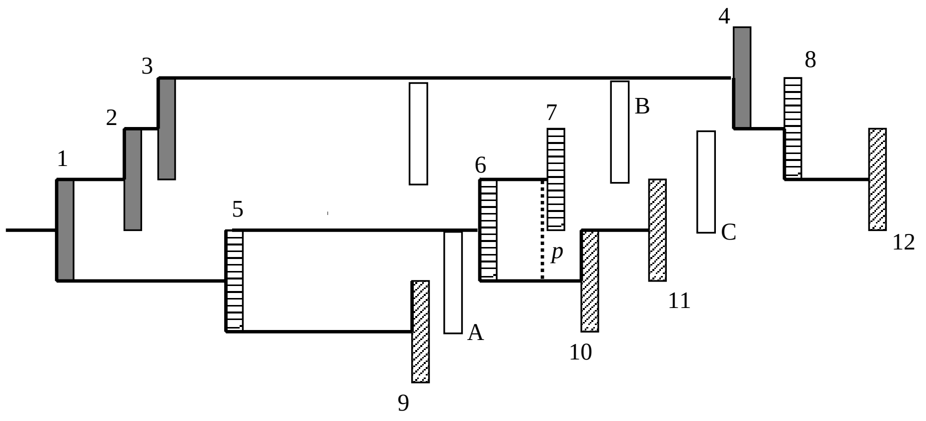


FIG. 5.4. *A scene showing a* 3×4 *fence-tree, where the root* $F_1(1)$ *is obstacle* 1. *The shaded obstacles are the nodes of the tree, and the dark lines are the edges. Differently-shaded obstacles constitute different fences; other obstacles are not part of any fence. For instance, obstacle* 4 *is node* $F_1(4)$, *obstacle* 7 *is node* $F_2(3)$, *and obstacle* 8 *is node* $F_2(4)$. *Note that the tree defines three disjoint fences with four obstacles each. If we had failed to follow the fence-tree rules and instead had made* $F_2(4)$ *be up-right from* $F_2(3)$ *(that is, obstacle B) then fences* 1 *and* 2 *would not be disjoint.*

The advantage of building an entire collection of fences on one trip is that this allows the robot to make more effective use of its movements. In [8] a detailed example is given where building fences one-by-one on consecutive trips can be too expensive. In fact a crucial property of the fence collections we will define is that the ith obstacle of a fence is always easily reachable either from the preceding obstacle on the same fence or the corresponding obstacle of the fence above.

5.2. Fence-trees. We would like to build a collection of $G = \lceil \frac{k}{a} \rceil$ up-fences across the window, and find an $O(L)$ length path crossing the collection, while paying a cost of only $O(kL)$. Our key idea is to define a *tree* structure whose nodes are obstacles in the scene and whose edges are "short" paths between the nodes. These nodes will constitute the desired collection of fences, and the path from the root node to the rightmost node is the desired $O(L)$ length path that crosses the collection. Furthermore, traversing all edges of this tree with a cost of $O(kL)$ is equivalent to building the desired collection of fences.

In order to define the tree structure we introduce some notation and terminology. The nodes of this tree will be denoted by $F_i(m)$, for $i = 1, 2, \ldots, G$ and $m = 1, 2, \ldots, M$ (M is roughly $2L/h$ and will be fully specified later). The reason for this notation is that $F_i(m)$ will turn out to be the mth obstacle of the ith fence. The coordinates of obstacle $F_i(m)$ are denoted by $(X_i(m), Y_i(m))$. We say an obstacle Q is *down-right* (*up-right*) from an obstacle P if Q is the first obstacle hit when moving to the right from the *bottom* (*top*) of P. The following rules then define the $G \times M$ fence-tree with root $F_1(1)$ (an example is given in Figure 5.4).

Fence-Tree Rules.

```
(1)  For $i = 2, 3, \ldots, G$, $F_i(1)$ is down-right from $F_{i-1}(1)$.
(2)  For $m = 2, 3, \ldots, M$, $F_1(m)$ is up-right from $F_1(m-1)$.
(3)  For $i = 2, 3, \ldots, G$:
         For $m = 2, 3, \ldots, M$:
             If $X_i(m-1) \geq X_{i-1}(m)$,
                 then $F_i(m)$ is up-right from $F_i(m-1)$
```

else $F_i(m)$ is down-right from $F_{i-1}(m)$.
(E.g., in Figure 5.4, obstacle 7 is up-right from obstacle 6, and obstacle 8 is down-right from obstacle 4.)

Thus each node $F_i(m)$ (except the root node $F_1(1)$) is defined to be either up-right from $F_i(m-1)$ or down-right from $F_{i-1}(m)$. We will call that "defining obstacle" of $F_i(m)$ its *parent*, and call the up-right or down-right path from the parent to $F_i(m)$ (consisting of a vertical portion of height h followed by a horizontal portion) an *edge* in the tree. That is, if rule (2) is used or if $F_i(m-1)$ is to the right of $F_{i-1}(m)$, then $F_i(m-1)$ is the parent of $F_i(m)$ and otherwise $F_{i-1}(m)$ is the parent of $F_i(m)$.

The fence-tree rules define a natural *binary rooted tree* structure. The tree structure is visually apparent in Figure 5.4. The $G \times M$ fence-tree in fact defines exactly the group of fences that we want to build.

THEOREM 5.1 (fence-tree). *Let P be an obstacle with y-coordinate $-L$ in a simple scene, and let $M \geq \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$. Then, the obstacles in a $G \times M$ fence-tree with root P form G disjoint up-fences with M obstacles each, with P as the first obstacle of the leftmost fence.*

Proof. It is easy to see that the obstacles $F_1(m)$ defined by rule (2) constitute a fence F_1. In addition, for all $i > 1$, the obstacles $F_i(m)$, $m = 1, 2, \ldots, M$, constitute a fence F_i since each is to the right of and exactly h higher than the previous obstacle. By rule (1), the initial obstacle of each fence is down-right from the initial obstacle of the fence F_{i-1} above it. Therefore, the fences are disjoint if and only if for each $m = 2, 3, \ldots, M$, $F_i(m)$ is to the right of $F_{i-1}(m)$, and this is guaranteed by rule (3).

It is easy to verify that the value $M = \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$ is sufficient to ensure that even the fence F_G that starts $Gh = h\lceil \frac{k}{a} \rceil$ below the bottom of the window extends at least $2L$ above the obstacle P, and thus all fences cross the window. ∎

Recall that we wanted to define the fence group so that there is a cheap ($O(L)$ length) path that crosses the group. One such path is the path from the root of the tree to the rightmost obstacle on the rightmost fence (strictly speaking, this path does not cross the rightmost fence, but it can be cheaply extended to one that does). In fact, the unique path in the tree from the root $F_1(1)$ to each node has length at most $O(L)$, as we show below.

LEMMA 5.2. *In a $G \times M$ fence-tree in a simple scene, where $G = \lceil \frac{k}{a} \rceil$ and $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$,*

(a) *the unique path in the tree from the root to each node has length $O(L)$, and*

(b) *the total length of all edges is $O(kL)$.*

Proof. Recall first that $k \leq n$ and $L \geq n$. Since the unique tree path from the root to any given node always proceeds to the right, the total horizontal cost of this path is at most n. On this path, each down-edge leads to a lower fence (and there are only $G = \lceil \frac{k}{a} \rceil$ fences), and each up-edge leads to an obstacle on the same fence (and there are only $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$ obstacles per fence). So, the total vertical cost of this path is at most $h(G + M)$. Thus the total length of any such path is at most $n + h(G+M) = n + h(2\lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil)$, which is $O(L)$ since $h = aL/\sqrt{nk} \leq aL/k$.

To bound the total length of all edges, note that each edge can be associated with a unique node, namely the one on its right (the child). Thus, the sum of the *vertical* portions of all edges is at most $h(GM-1) = O(kL)$. By fence rules (2) and (3), the length of the horizontal portion of the edge associated with $F_i(m)$ is no more than the horizontal distance between $F_i(m)$ and its predecessor $F_i(m-1)$ on the same fence. Thus the sum of the *horizontal* portions of the edges is a most Gn: n for each fence. This is also $O(kL)$. ∎

Thus if the robot traverses all edges of this tree it will have found not only a group of disjoint fences but also a cheap path that crosses all of them.

As noted in the proof above, there are GM obstacles in the fence-tree, and $GMh \leq kMh = O(kL)$. Thus we would like the robot to traverse the fence-tree with a cost proportional to h times the number of obstacles in the fences, or a cost proportional to the total length of the tree edges. We remark here that the fence-tree must be traversed online; a simple approach based on depth-first-traversal may not be efficient since the algorithm does not know where exactly the nodes are: the robot can locate $F_i(m)$ only after it has located both its potential parents $F_{i-1}(m)$ and $F_i(m-1)$ or at least after it has determined whether or not $X_i(m-1) \geq X_{i-1}(m)$. So, intuitively the difficulty is that before visiting a node such as $F_i(m)$ we need to visit both potential parent nodes, which may be in very different parts of the tree. (Actually, one could imagine an algorithm that attempted to visit nodes before finding both parents and only later verified whether or not those nodes were actually legally part of the tree; our algorithm does not do this.)

Conventions. In the subsequent sections, we will find it convenient to associate an edge in the tree with the obstacle at its right end, and we define the coordinates of an edge to be the coordinates of its associated obstacle. We say an edge belongs to a fence F_i if its associated obstacle belongs to F_i. When we say an object A, such as an edge or obstacle, is left (right) of another object B we will mean that the x-coordinate of A is strictly smaller (greater) than that of B. We will often identify a fence with its rightmost obstacle; thus when we say "fence F_i is to the left of obstacle P" we mean that the last obstacle of F_i is left of P. We use (X_i, Y_i) to denote the coordinates of the rightmost obstacle of F_i, and $|F_i|$ will denote the number of obstacles currently in F_i. To simplify the wording of our algorithms we will assume that $X_0 = \infty$, and $|F_0| = M + 1$.

5.3. Finding the fence-tree. Our algorithm builds a tree using a conservative strategy in the following sense. It adds a new edge to the current partial tree only when such an edge is certain to be part of the final tree being built. In addition, the algorithm visits a node $F_i(m)$, $i > 1, m > 1$, only after both its possible parents $F_{i-1}(m)$ and $F_i(m-1)$ have been visited. At any stage our algorithm will be located at the rightmost node found so far for some partial fence and then either adds a (down-right or up-right) edge from the current fence, or "jumps" to another fence.

It is reasonable to wonder whether an efficient fence-tree-traversal strategy exists that only walks along the tree edges when jumping from one fence to another. We do not know of any such strategy; our algorithm may often shortcut to another fence without necessarily walking along tree edges. Even with this freedom to walk outside the tree, it is important to note that a bad order of visiting the nodes of the fence-tree may make the jumps prohibitively expensive.

The key problems in designing a traversal strategy therefore are (a) deciding the order in which the nodes will be discovered, and (b) designing the jump procedures. Procedure FindFenceTree in Figure 5.5 finds the desired fence-tree, using a recursive procedure Raise described in Figure 5.6. In these procedures, it should be understood that if the "wall" $x = n$ is reached at any time, the robot halts and the procedures terminate. The procedure JumpDownLeft (i) and JumpDownRight (i) take the robot from the last obstacle of the current fence F_i to the last obstacle of the next lower fence F_{i+1}. These procedures are described in Figures 5.7 and 5.8. In all the procedures, the "retrace to $F_i(j)$" statements are executed by simply retracing the path used to reach the current position from $F_i(j)$.

```
1  procedure FindFenceTree
2    Move to the right until at an obstacle; this defines $F_1(1)$
3    for $i := 2$ to $G$ do
4        Add a down-right edge to define$F_i(1)$;
5    end
6    Retrace to $F_1(1)$;
7    Raise $(1, 0)$;
8  end
```

FIG. 5.5. *The main procedure for finding the fence-tree.*

```
1  procedure Raise $(i, q)$
2    while (Up($i$)) do
3            Retrace to $F_i$; $j := i$;
4            Add an up-right edge to $F_i$;
5            while ($j < G$ and $X_j > X_{j+1}$) do
6                    Retrace to $F_j$;
7                    if (Down($j$))
8                      Add a down-right edge;   $j := j + 1$;
9                    else
10                        JumpDownLeft $(j)$;
11                        Raise $(j + 1, j)$;
12                   fi
13           od
14   od
15   Let $F_d$ be the current fence;
16   if ($d < G$ and $X_{d+1} < X_q$)
17     JumpDownRight $(d)$;
18     Raise $(d + 1, q)$;
19   fi
20 end
```

FIG. 5.6. *Recursive procedure* Raise *used by* FindFenceTree.

We start with an intuitive description of the algorithm. The algorithm begins (FindFenceTree) by finding the first obstacle in each fence and placing itself at the first obstacle of the topmost one. It then calls the recursive procedure Raise $(1, 0)$. In general for $q < i$, the job of Raise (i, q) is to raise all fences i and lower that are currently behind F_q, as far as possible given the constraints imposed by the location of F_q. (For $i = 1, q = 0$, this means to raise all the fences until they each have M obstacles.)

The Raise procedure is a bit complicated, so is perhaps best described through the example of Figure 5.4. In this example, Raise $(1, 0)$ is first called at obstacle 1, and the algorithm knows only about obstacles 1, 5, and 9. Raise begins by adding new obstacles to its current fence (in line 4) so long as these are legal with respect to constraints imposed from above, until it has overshot the fence below it. In the example, these are obstacles 2, 3, and 4. Once it reaches obstacle 4, Raise realizes it

```
1 procedure JumpDownLeft ($j$)
2    Move left along tree edges until $x$-coordinate = $X_{j+1}$;
3    Move vertically down to last obstacle of $F_{j+1}$;
4 end
```

FIG. 5.7. *Procedure* JumpDownLeft *to jump from current fence F_j to the next lower fence F_{j+1} when $X_{j+1} < X_j$.*

```
1 procedure JumpDownRight ($d$)
2    Move vertically down until on a previously found tree edge;
3    Follow tree edges to the right until at last obstacle of $F_{d+1}$;
4 end
```

FIG. 5.8. *Procedure* JumpDownLeft *to jump from current fence F_d to the next lower fence F_{d+1} when $X_{d+1} \geq X_d$.*

is to the right of the fence F_2 below it (formally, the conditions of the inner while loop become satisfied) and will try to make sufficient progress on F_2 (and any others that are behind and below F_1, in this case F_3). Unfortunately, because the current obstacle (number 4) of fence F_1 is too high relative to obstacle 5 on fence F_2, the algorithm cannot simply add a down-right edge (formally, Down(j) is not satisfied) from obstacle 4 to discover the next obstacle of fence F_2. So, the algorithm runs the JumpDownLeft(1) procedure to reach the last obstacle (number 5) of fence F_2 and calls Raise $(2, 1)$ recursively to raise that fence. This call to Raise begins by finding obstacle 6. At this point the robot is to the right of the fence below it (the condition of the inner while loop is satisfied) and it is just high enough above the last obstacle of F_3 (i.e., Down(j) is satisfied) so it adds the down-right edge to obstacle 10 (line 8). At this point it goes back to obstacle 6 (since Up(i) is still satisfied) and makes the up-right edge to obstacle 7. Now, Up(i) is no longer satisfied because the current fence has bumped into the constraint imposed by the fence above it. So, Raise $(1, 0)$ drops down to line 17, where it calls JumpDownRight (2) to get back to obstacle 10 (using the path indicated by "p" in the figure) and then recursively calls itself to work on raising that fence. Finally, that recursive call ends with obstacle 11, we pop out of both levels of recursion, and at the very top level we retrace our path all the way to obstacle 4, finally adding down-right edges to obstacles 8 and 12 in the inner while loop.

We now give a formal analysis of the formal algorithm given in Figures 5.5–5.8. In order to establish the correctness and bound the cost of these procedures, we need to show that certain pre- and postconditions hold whenever they are invoked. For ease of reference we use mnemonic names for the various conditions:

- Up(i): $i \geq 1$ and an up-right edge is legal from fence F_i, i.e., either $X_i < X_{i-1}$ and $|F_i| < |F_{i-1}| - 1$, or $X_i \geq X_{i-1}$ and $|F_i| < |F_{i-1}|$.
- Down(i): $i < G$ and a down-right edge is legal from F_i, i.e., $X_{i+1} < X_i$ and $|F_{i+1}| = |F_i| - 1$.
- Ord(i, j): if $i \leq j$, then $X_i \leq \cdots \leq X_j$.
- At(i) means that the robot is at the last known obstacle of fence F_i.
- Eq(i, j) means "if $i \leq j$, then $|F_i| = \cdots = |F_j|$."
- Unch(i, j) stands for "if $i \leq j$, then the values of $|F_i|$ through $|F_j|$ have not

PreRaise(i, q):
1. $\mathrm{Eq}(q+1, i-1)$,
2. $\mathrm{At}(i)$,
3. $i > q$,
4. $X_i < X_q$,
5. $\mathrm{Ord}(q+1, G)$,
6. If $q+1 < i$, then NOT $\mathrm{Up}(q+1)$,
7. $\mathrm{Up}(i)$.

OuterWhile(k) (F_c is the current fence):
1. $\mathrm{Unch}(1, i-1)$,
2. $\mathrm{Eq}(i, c)$ and $c \geq i$,
3. $\mathrm{Ord}(i, G)$,
4. AtNewest holds after the first (if any) iteration of the loop.

PreJDL(j):
1. $\mathrm{At}(j)$,
2. $X_{j+1} < X_j$,
3. $\mathrm{Up}(j+1)$.

PostRaise(i, q) (F_c is the current fence):
1. $\mathrm{Unch}(1, i-1)$,
2. $\mathrm{Eq}(q+1, c)$,
3. $\mathrm{Ord}(q+1, G)$,
4. If $c < G$, $X_{c+1} \geq X_q$,
5. NOT $\mathrm{Up}(q+1)$,
6. AtNewest.

InnerWhile(k) (F_c is the current fence):
1. $\mathrm{Unch}(1, i-1)$,
2. $\mathrm{Eq}(i, j)$,
3. $\mathrm{Ord}(j+1, G)$,
4. AtNewest.
5. If $c > j$, then all PostRaise$(j+1, j)$ conditions hold.

PreJDR(d):
1. $\mathrm{At}(d)$,
2. $X_d \leq X_{d+1} < X_q$,
3. $\mathrm{Eq}(q+1, d)$,
4. NOT $\mathrm{Up}(q+1)$,
5. $\mathrm{Up}(d+1)$.

FIG. 5.9. *Various conditions required for the formal proof.*

changed since the start of the procedure or while loop under consideration."
- AtNewest means that the robot is at the newest obstacle found so far.
- AlmostOrd: No fence has more than one obstacle to the right of a lower fence. That is, for $i = 1, 2, \ldots, G$, if $|F_i| = m$, then $X_i(m-1) \leq X_j$ for all $j > i$.

We prepend a condition by NOT to signify that the logical negation of the condition holds. For easy reference, in Figure 5.9 we define several collections of conditions that will be useful in the correctness proof of the algorithm.

We use the next several lemmas to establish the correctness of the procedure FindFenceTree (Theorem 5.8).

LEMMA 5.3. *Whenever* Raise (i, q) *is called, the PreRaise*(i, q) *conditions hold. Moreover, the procedure terminates, and the PostRaise*(i, q) *conditions hold at that time.*

Proof. It is easy to verify that when FindFenceTree makes the first call Raise $(1, 0)$, the PreRaise$(1, 0)$ conditions hold. We claim that if the PreRaise(i, q) conditions hold when Raise (i, q) is called, then Raise (i, q) terminates and the PostRaise(i, q) conditions hold. We prove this by induction. The base case is $i = G$, i.e., an invocation of the form Raise (G, q): in this case there are no recursive calls to Raise, the inner while loop is not entered, and JumpDownLeft and JumpDownRight are not called. The only effect of Raise (G, q) is that up-right edges are added (line 4) to fence F_G until $\mathrm{Up}(G)$ is not true. It is easy to check that Raise (G, q) terminates with all the PostRaise(G, q) conditions holding.

Next, let us inductively assume that the claim holds for all calls to Raise $(j, .)$, for $j = i+1, \ldots, G$. Consider an invocation of Raise (i, q) at some point when the PreRaise(i, q) conditions hold. Before the outer while loop is entered, the current fence is F_c, where $c = i$. It is easily verified that the PreRaise(i, q) conditions imply that all the OuterWhile conditions hold at this point. By Lemma 5.4, when the outer while loop is exited, all the OuterWhile conditions continue to hold, *and* NOT Up(i) holds. At this point, if the "If" condition on line 16 fails, then we claim that all the PostRaise(i, q) conditions hold. Most of these conditions are easy to check, so we will only argue the less trivial ones. To argue that condition (2) Eq$(q + 1, c)$ holds, note that NOT Up(i), combined with Ord$(q + 1, G)$ and $i > q$ (from PreRaise(i, q)), imply Eq$(i - 1, i)$. This, combined with Eq$(q + 1, i - 1)$ (in PreRaise(i, q)) and Eq(i, c) (in OuterWhile) implies Eq$(q+1, c)$. Condition (5) NOT Up$(q+1)$ follows from Eq$(q+1, c)$ (which implies Eq$(q + 1, i)$) and NOT Up(i).

Thus if the condition on line 16 fails, then Raise (i, q) terminates with the PostRaise (i, q) conditions holding, and the lemma is proved. But if the condition on line 16 is true upon exit of the outer while loop, then we claim that the conditions PreJDR(d) hold. Again we will only show the arguments for the nontrivial ones among these: condition (2) $X_d \leq X_{d+1} < X_q$ follows from Ord(i, G) and $c \geq i$ (in OuterWhile), which imply $X_d \leq X_{d+1}$ (where $d = c$), and from the truth of the If condition. The reasoning to show that (5) Up$(d + 1)$ holds is as follows. Since AtNewest holds (from OuterWhile), this means that the robot has just found the new obstacle on F_c. Since obstacle number $|F_d|$ on F_{d+1} can only be found after the corresponding obstacle on F_d, this means that $|F_{d+1}| < |F_d|$. From the condition $X_d \leq X_{d+1}$ that we just argued, this implies that Up$(d + 1)$ must hold.

Now when JumpDownRight (d) is invoked on line 17, by Lemma 5.7, the robot ends up at (the most-recently discovered obstacle of) F_{d+1}. At this point we claim that the conditions PreRaise$(d + 1, q)$ hold. In particular, condition (3) $d + 1 > q$ holds since $d \geq i$ (from OuterWhile) and $i > q$ (from PreRaise(i, q)). (This actually implies that $d > q$, a fact we will use below.) Condition (4) Ord$(q + 1, G)$ is one of the PreRaise(i, q) conditions, which we assumed to hold. The fact that $d > q$ implies $q + 1 < d + 1$, and NOT Up$(q + 1)$ was already argued above, so (5) holds. The remaining PreRaise$(d + 1, q)$ conditions are easy to check.

By induction assumption, therefore, procedure Raise $(d + 1, q)$ terminates with the conditions PostRaise$(d + 1, q)$ holding. Of these conditions, all but condition (1) depend only on q and the number c of the current fence at the end of the procedure. So the conditions PostRaise(i, q) (2) through (6) hold. Finally, condition (1) Unch$(1, i-1)$ holds because it is an OuterWhile condition, and this completes the proof. ∎

LEMMA 5.4. *The conditions OuterWhile are invariants for the outer while loop.*

Proof. Suppose all the OuterWhile conditions hold just before entry of the outer while loop. If the outer while loop is entered, an obstacle is added to the current fence $F_c = F_i$ via an up-right edge at line 4, and at this point $j = i$. At this stage it is trivial to verify that the InnerWhile conditions hold. By Lemma 5.5, upon exit of the inner while loop, all the invariants InnerWhile continue to hold, *and* either $j = G$ or $j < G$ and Ord$(j, j + 1)$ holds. At this point we claim that the OuterWhile conditions hold: condition (1) Unch$(1, i-1)$ is an InnerWhile condition. We argue condition (2) Eq(i, c) as follows. If $c = j$, then Eq(i, j) (from InnerWhile) implies Eq(i, c). Otherwise, $c > j$, in which case from the InnerWhile conditions, all the PostRaise$(j + 1, j)$ conditions hold. In particular, Eq$(j + 1, c)$ holds and Up$(j + 1)$ is false. Since $c > j$ we must also have $j < G$ and Ord$(j, j + 1)$ (the failure of the conditions of the inner while loop), i.e., $X_j \leq X_{j+1}$. Since Up$(j + 1)$ is false, this must mean that $|F_j| = |F_{j+1}|$. Thus we have Eq(j, c). This, combined with Eq(i, j) from the InnerWhile conditions, implies Eq(i, c). Condition (3) Ord(i, G) is argued as follows. The InnerWhile conditions

Ord$(j+1, G)$ and Eq(i, j), and the condition Ord$(j, j+1)$ that holds because the inner while loop was just exited, imply Ord(i, G). Finally, condition (4) AtNewest follows from the fact that just before returning to the start of the outer while loop, either an up-right edge was added at line 4, or the inner while loop was executed, and AtNewest is one of its invariants. ☐

LEMMA 5.5. *The conditions InnerWhile are invariants for the inner while loop.*

Proof. Suppose all the conditions InnerWhile hold at the start of an iteration of the inner while loop. If the loop is entered, then clearly $X_j > X_{j+1}$. If at this point Down(j) holds, then a down-right edge is added, and j is incremented to $j+1$. At this point it is easy to check that all the InnerWhile conditions continue to hold. On the other hand, if Down(j) does not hold, then we claim that the conditions PreJDL(j) hold: (1) At(j) is clearly true. (2) $X_{j+1} < X_j$ holds as we observed above. (3) Up$(j+1)$ holds since $X_{j+1} < X_j$ and NOT Down(j) holds.

By Lemma 5.6, after JumpDownLeft is executed, the robot is at F_{j+1}. At this point we claim that all the conditions PreRaise$(j+1, j)$ hold. For instance, condition (5) Ord$(j+1, G)$ holds since by assumption it held when the inner while loop was entered (being one of the InnerWhile conditions), and before this invocation of Raise $(j+1, j)$, no new obstacles were discovered on any fence. The remaining Raise $(j+1, j)$ conditions are trivially checked.

By Lemma 5.3, after Raise is executed, all the conditions PostRaise$(j+1, j)$ will hold. At this point we claim that all the conditions InnerWhile continue to hold: (1) Unch$(1, i-1)$ is maintained since fences F_j and above are unaffected by Raise$(j+1, j)$ (this is the Unch$(1, j)$ condition in PostRaise$(j+1, j)$), and $j \geq i$. (2) Eq(i, j) is maintained for the same reason. Conditions (3) Ord$(j+1, G)$ and (4) AtNewest are also PostRaise$(j+1, j)$ conditions. Finally, we just argued above that the PostRaise$(j+1, j)$ conditions hold, and this is condition (5) of InnerWhile. ☐

We establish below that the procedures JumpDownLeft and JumpDownRight work correctly if they are invoked under appropriate conditions.

LEMMA 5.6. *Whenever* JumpDownLeft (j) *is invoked, the conditions PreJDL(j) hold, and the procedure terminates with the robot at the last known obstacle of F_{j+1}.*

Proof. That the conditions PreJDL(j) hold whenever JumpDownLeft (j) is invoked can easily be seen from the proofs of Lemmas 5.3 and 5.5. The PreJDL(j) condition $X_{j+1} < X_j$ implies that the motion in line 2 (following tree edges to the left) leads to a point where the x-coordinate is X_{j+1} (see Figure 5.10). Since edges followed to the left only lead to the same fence or to a higher one, this implies that the point at the end of the motion in line 2 is vertically above (and not below) the last known obstacle P of F_{j+1}. Thus moving vertically down in line 3 leads to obstacle P. ☐

LEMMA 5.7. *Whenever the procedure* JumpDownRight (d) *is invoked, the conditions PreJDR(d) hold, and the procedure terminates with the robot at the last known obstacle of F_{d+1}.*

Proof. That the conditions PreJDR(d) hold whenever JumpDownRight (d) is invoked can easily be seen from the proof of Lemma 5.3. Since $X_d \leq X_{d+1}$, moving vertically down from F_d will lead to a tree edge that is to the left of F_{d+1}. Therefore, following the tree edges to the right will lead to the most recently discovered obstacle of F_{d+1}. ☐

From the above lemmas it is easy to prove that the procedure FindFenceTree finds the desired fence-tree.

THEOREM 5.8. *When executing procedure* FindFenceTree*, the robot either finds a*

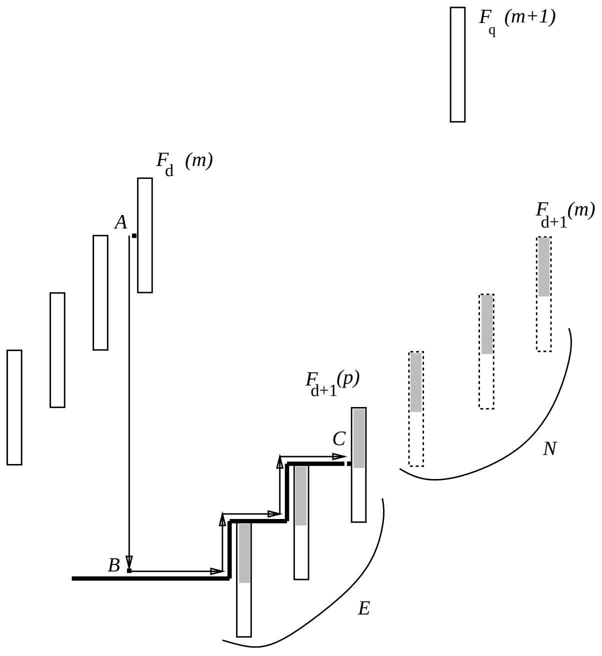


FIG. 5.10. *Showing a use of procedure* JumpDownRight *to jump from $F_d(m)$ to $F_{d+1}(p)$. Solid-boundary rectangles are obstacles found so far in the tree. The set N of dotted rectangles represents obstacles that will be found on fence F_{d+1} immediately following this procedure. Thick solid lines are tree edges. The thin solid line shows the path followed when executing the procedure. The procedure starts from A (obstacle $F_d(m)$), goes vertically down to a tree edge (point B), then follows the edges to the right to the final obstacle of F_{d+1} (point C). The set E is the set of edges followed in BC. The length of AB is no more than the lengths of the edges in E plus the heights of the obstacles in N.*

complete $G \times M$ fence-tree or reaches the wall (the line $x = n$) after having found a collection of i partial fences $F_1, F_2, \ldots, F_i$ that satisfy the fence-tree rules.

Proof. Recall that we have set $|F_0|$ to be $M + 1$ and X_0 to be infinite. After the first obstacle on fence $F_1, F_2, \ldots, F_G$ is found in line 4 of FindFenceTree, the robot returns to $F_1(1)$. At this point it is easy to check that the PreRaise$(1, 0)$ conditions are satisfied. When Raise $(1, 0)$ is invoked in line 7, by Lemma 5.3, the robot completes the procedure (if it hasn't reached the wall) with the PostRaise$(1, 0)$ conditions holding. In particular, condition (4) implies that the current fence upon completion of the procedure must be $F_c = F_G$ (since otherwise $X_{c+1} \geq X_0$, which is impossible since $X_0 = \infty$). Also, conditions (2) Eq$(1, c)$ and (5) NOT Up(1) imply that $|F_1| = |F_2| = \cdots = |F_G| = |F_0| - 1 = M$. ∎

Finally, we establish an invariant that will be useful later.

LEMMA 5.9. *The AlmostOrd invariant holds throughout any execution of* Raise (i, q).

Proof. The only two statements of the procedure which could possibly result in a violation of the AlmostOrd invariant are 4 (where an up-right edge is added) and 8 (where a down-right edge is added). But whenever line 4 is reached, Ord(i, G) holds: this follows from Ord$(q + 1, G)$, which is one of the OuterWhile conditions (which we prove below to be invariants for the outer while loop), and $i > q$, which is a PreRaise(i, q) condition. Thus even if the new obstacle added to F_i in line 4 is to the right of (the last obstacle of) a lower fence, this would be the only such obstacle of F_i. Similarly, whenever a down-right edge is added from fence F_j (thereby adding an obstacle to F_{j+1}), $\mathrm{Ord}(j+1, G)$ holds: this is one of the InnerWhile conditions, which we show below to be invariants for the inner while loop. Thus even if the new obstacle added to F_{j+1} in line 8 is to the right of some lower fence, it would be the only such obstacle of F_{j+1}. ∎

5.4. Cost analysis. Recall from subsection 5.2 that we would like our fence-tree-finding algorithm to travel a distance of no more than $O(kL)$. The next theorem establishes this.

THEOREM 5.10. *For* $G = \lceil \frac{k}{a} \rceil$ *and* $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$, *the algorithm* FindFenceTree *for finding a* $G \times M$ *fence-tree in a simple scene has total cost* $O(kL)$.

Proof. From Lemma 5.2, it suffices to show that the total distance traveled by the robot while executing FindFenceTree is bounded by some constant times the total length of all edges plus the heights of all obstacles in the fence-tree. There are four kinds of motions performed by the algorithm:

- adding an up-right edge (line 4 of Raise);
- adding a down-right edge (line 8 of Raise);
- retracing an old path (lines 3, 6 of Raise);
- jumping from a fence to the next lower one, using procedures JumpDownLeft and JumpDownRight.

Consider a specific iteration of the outer while loop of Raise. If the robot is not at F_i at the start of this iteration, it executes "Retrace to F_i" at line 3. This motion consists simply of retracing the paths it walked while executing the remaining lines of this loop, during the *previous* iteration of the loop. So the retracing motion at line 3 in a given iteration of the outer while loop can be charged off to the non-line-3 motions executed during the previous iteration of the outer loop. Similarly, the retracing motion at line 6 in a given iteration of the inner while loop can be charged off to the non-line-6 motions executed during the previous iteration of this loop. Thus it suffices to bound the cost of the remaining three kinds of motions. Clearly, the motion required to add up-right and down-right edges can be charged off to the edges created, so we only need to bound the cost of the procedures JumpDownLeft and JumpDownRight. Lemmas 5.11 and 5.12 below establish that the total cost of these procedures is $O(kL)$, which implies our theorem. ∎

LEMMA 5.11. *The total distance traveled during all invocations of* JumpDownLeft *is* $O(kL)$.

Proof. Consider a call to JumpDownLeft (j) at line 10 of Raise (i, q), and suppose the robot is at obstacle $F_j(m)$ when this procedure is invoked. The edge-following motion in line 2 of this procedure can be charged to the set E of edges followed. The right ends of all these edges are at obstacle number m of different fences. Note that just before JumpDownLeft (j) is invoked, $\mathrm{Up}(j+1)$ is true (this is a $\mathrm{PreJDL}(j)$ condition), and when Raise $(j+1, j)$ is completed after JumpDownLeft (j), $\mathrm{Up}(j+1)$ is *not* true (this is a $\mathrm{PostRaise}(j+1, j)$ condition). This means that the procedure Raise $(j+1, j)$ has discovered a collection N of *new* obstacles on fence F_{j+1}, so that $|F_{j+1}|$ would be at least $m-1$. The cost of the vertical motion in line 3 of JumpDownLeft (j) is clearly no more than the total length of the edges E plus the heights of the new obstacles N discovered by the subsequent call to Raise. We can thus charge the total cost of this specific invocation of JumpDownLeft (j) to the set E of edges and the set N of new obstacles. Now we need to argue that the sets E and N of future calls to JumpDownLeft will not overlap with those of the present call. Since the obstacles N

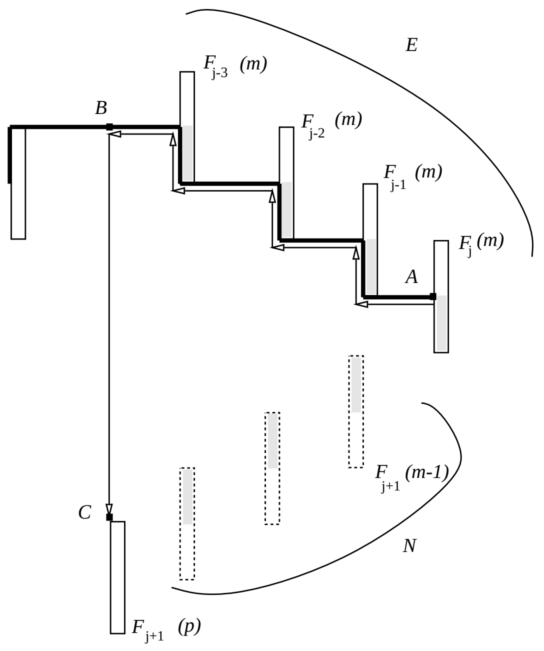


FIG. 5.11. *Showing a use of procedure* JumpDownLeft *to jump from $F_j(m)$ to $F_{j+1}(p)$. Solid-boundary rectangles are obstacles found so far in the tree. The set N of rectangles with dotted boundaries are obstacles that will be found on fence F_{j+1} immediately after this procedure completes. Thick solid lines are tree edges. The thin solid line is the path followed when executing the procedure. The procedure starts from A (obstacle $F_j(m)$), retraces tree edges to the left to point B, then goes vertically down to C, at the top of the final obstacle of F_{i+1}. The set E is the set of edges retraced in AB. The length of BC is no more than the total length of the edges in E plus the heights of the obstacles in N.*

are *new* obstacles discovered on F_{j+1} just after the present call to JumpDownLeft (j), the only future calls to JumpDownLeft whose N-sets can possibly overlap with the N set of the present one are calls to JumpDownLeft from the same fence F_j, i.e., calls to JumpDownLeft (j). However, as we observed above, the Raise $(j+1, j)$ executed just after the present JumpDownLeft (j) finds the obstacles on N before any future call to JumpDownLeft (j) is made. Moreover, it is easy to see that the execution of Raise $(j+1, j)$ does not involve any calls to JumpDownLeft (j). Therefore the N-sets of different calls to JumpDownLeft do not overlap.

Now we show that the E-sets of different calls to JumpDownLeft do not overlap. Again consider a specific invocation of JumpDownLeft (j) from obstacle $F_j(m)$. As we noted above, all the edges in E have at their ends the mth obstacle of different fences. Therefore the only future invocations of JumpDownLeft whose E sets can possibly overlap with the current E set are those from obstacle m of some fence F_u below F_j. We established before that an invocation of JumpDownLeft (u) is only made when the conditions PreJDL(u) hold, and in particular (a) $X_{u+1} < X_u$ and (b) Up$(u+1)$ must hold. However, just after the present execution of JumpDownLeft (j), Raise $(j+1, j)$ is executed, and at that point the PostRaise$(j+1, j)$ conditions hold. In particular, for any fences F_u below F_j such that $X_u < X_j$, Up(u) does not hold. Therefore when (if at all) a future call to JumpDownLeft (u) is made from obstacle m of a fence F_u below F_j, $X_{u+1} \geq X_j(m)$ must hold at that time. In such a future invocation, in line

2, edges are followed to the left until the x-coordinate equals X_{u+1}, so these edges would be to the *right* of $X_j(m)$ and so would not overlap with the edges E of the present set (all of which are to the *left* of X_j). Thus the "charge sets" E and N for different calls to JumpDownLeft (j) will not overlap. ∎

LEMMA 5.12. *The total distance traveled during all invocations of* JumpDownRight *is $O(kL)$.*

Proof. Consider an invocation of JumpDownRight (d) at line 17 of Raise (i, q). We will present a charging scheme where different invocations of JumpDownRight (d) will be charged to distinct portions of the fence-tree. When JumpDownRight (d) is invoked, $\mathrm{Up}(d+1)$ is true (this is a $\mathrm{PreJDR}(d)$ condition). Subsequent to this invocation of JumpDownRight (d), Raise $(d+1, q)$ is invoked, and when that procedure completes, the $\mathrm{PostRaise}(d+1, q)$ conditions imply that $\mathrm{Up}(d+1)$ is no longer true. This means that Raise $(d+1, q)$ has found a collection N of new obstacles on fence F_{d+1}, and then $|F_{d+1}|$ would equal $|F_d| = m$ (this is a $\mathrm{PostRaise}(d+1, q)$ condition). The edge-following motion in line 3 of JumpDownRight (d) can be charged to the set E of edges that are followed. The vertical motion in line 2 of JumpDownRight (d) is clearly no more than the lengths of the edges E plus the heights of the obstacles N, so this motion can be charged to the edge-set E and the obstacle-set N (see Figure 5.11).

Note that since $\mathrm{Up}(d+1)$ is not true after the Raise procedure completes just after this invocation JumpDownRight (d), the next invocation of JumpDownRight (d) can only occur from obstacle $m+1$ or later of F_d. Since presently $X_d < X_q$, and NOT $\mathrm{Up}(q+1)$ and $\mathrm{Eq}(q+1, d)$ hold, the obstacle $m+1$ of F_d will be to the right of X_q. So the edge-set E followed by any future invocation of JumpDownRight (d) will be distinct from the set E of the present invocation. Also, since N is the set of *new* obstacles discovered just after the present invocation of JumpDownRight (d), the set N of any future call to JumpDownRight (d) will not overlap with the set N of the present one. In fact, since we are charging the motion of JumpDownRight (d) only to obstacles and edges associated with fence F_{d+1}, the sets N and E of any future call to JumpDownRight (d') will also not overlap with the sets N and E of the present call. ∎

6. Extension to arbitrary axis-parallel rectangular obstacles.

We now show how to extend the search algorithm to scenes with arbitrary axis-parallel rectangular obstacles (for brevity we call such scenes *general* scenes). That is, we will show how to explore a distance of $O(L\sqrt{nk})$ and find a path of length $O(L\sqrt{n/k})$. Fortunately it turns out that algorithm FindFenceTree, interpreted appropriately, can be used unchanged for these scenes. However, the procedures JumpDownRight and JumpDownLeft must be modified since for general scenes vertical motion is not always unobstructed. In fact if all obstacles have width 1 (but arbitrary heights and positions), then even these procedures remain unchanged. In the next two subsections, we define the notions of *τ-post* and a *τ-fence,* which are the analogues of "obstacle" and "fence" for general scenes. As stated earlier, we will assume throughout that all obstacles have their corners at integer coordinates.

6.1. τ-posts.

Throughout this section we will denote the value $L/\sqrt{nk}$ by τ. We assume $k \le n$ so $\tau \ge 1$. Recall that in a simple scene if the obstacles have height less than $2\tau = 2L/\sqrt{nk}$, then they can be considered "small"—in the sense that the simple strategy of just moving horizontally forward (walking around any obstacles on the way by the shortest route) achieves the optimal ratio of $O(\sqrt{n/k})$ on *each* trip. This motivates the following definition in a general scene. A point P on the left side of an obstacle is called a *τ-post* if the obstacle extends vertically at least τ above and

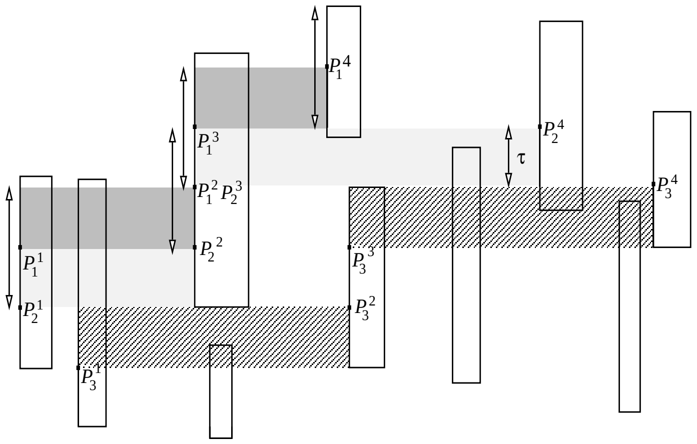


FIG. 6.1. *A collection of three disjoint fences with four posts each. The solid rectangles are the obstacles. The bands of different fences are shaded differently. For convenience, post $F_i(m)$ is denoted P_i^m.*

below P. We will use the term τ-post to refer either to the entire segment of height 2τ or just to the center of that segment. Roughly speaking when the robot encounters a τ-post P while moving horizontally, the obstacle encountered is "big," otherwise it is "small."

6.2. τ-fences. We define a τ-fence as the generalization of the fence defined in section 5.1. The definitions (and notations) for *up τ-fence*, *down τ-fence*, *band*, as well as the definition of a point being *left* or *right* of a fence, and of a path *crossing* a fence, remain the same as in simple scenes, except that we replace the word "obstacle" with "τ-post" throughout and replace h by τ in the relations (5.1) and (5.2) of section 5.1. Note that consecutive τ-posts of a fence may lie on the *same* obstacle, since the inequality (5.1) is not strict. The band between two such posts is empty. We say two τ-fences are *disjoint* if their *nonempty* bands are disjoint. Thus a collection of k disjoint τ-fences costs at least $k\tau$ to cross. In Figure 6.1, the sequence of τ-posts $\langle F_1^1, F_1^2, F_1^3, F_1^4\rangle$ form a τ-fence F_1. Note that F_1^2, F_1^3 are on the same obstacle and that the three fences in the figure are disjoint.

For future reference we define a (right) *τ-path* as the path of the robot when it moves to the right along a fixed horizontal line $y = y_0$ until it hits a τ-post or the wall, moving around any nonpost obstacle on its way. For instance, in Figure 6.2, the path from A to τ-post $F_1(2)$ is a τ-path. Observe that a τ-path has vertical motion at most 2τ at every (integer) x-coordinate on the path, so we have the following.

FACT 6.1. *A τ-path between two points (x, y) and $(x + \delta x, y)$ has length at most $\delta x + 2\tau\ \delta x$.*

6.3. The initial search trip. Roughly speaking, a general scene is treated as if it is a simple scene with obstacles of height $2h = 2\tau = 2L/\sqrt{nk}$. Recall that for simple scenes, the initial trip consists of building groups of G fences of M obstacles each (where $G = \lceil \frac{k\tau}{h} \rceil$ and $M = \lceil \frac{k\tau}{h} \rceil + \lceil \frac{2L}{h} \rceil$), where each group must be built "cheaply" (i.e., with cost $O(kL)$) and must have a known "short" (cost $O(L)$) path crossing it. Analogously for general scenes, we have $h = \tau$ and we would like to build groups of k τ-fences with $M = k + \lceil \frac{2L}{\tau} \rceil$ τ-posts each. We will now pay more

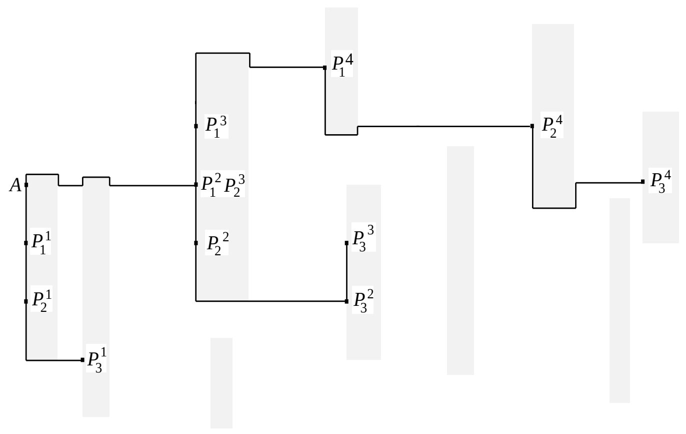


FIG. 6.2. *A 3×4 τ-fence-tree. The shaded rectangles are the obstacles, and solid lines are tree edges. The fences corresponding to this tree are shown in Figure* 6.1. *For convenience, post $F_i(m)$ is denoted P_i^m.*

attention to our progress in the x direction and will build each group with cost at most $O(kL + k\tau\ \Delta x)$ and give each group a group-crossing path of length $O(L + \tau\ \Delta x)$. Here Δx is the x-distance between the leftmost τ-post $F_1(1)$ and rightmost τ-post $F_k(M)$ in the group.

These bounds are sufficient for our purposes for the following reason. Since a fence costs $\tau = L/\sqrt{nk}$ to cross, there can be at most $\sqrt{nk}$ disjoint τ-fences in the window between s and t, so the algorithm will find at most $\sqrt{n/k}$ groups of k fences each. Since the x-motions do not overlap between the groups of fences, the Δx terms add to at most n, so the total distance traveled is $O(kL\sqrt{n/k} + nk\tau) = O(L\sqrt{nk})$. In addition, the concatenation of the group-crossing paths has total cost $O(L\sqrt{n/k} + n\tau) = O(L\sqrt{n/k})$.

6.4. Extending FindFenceTree to general scenes. Once we fix the τ-post $F_1(1)$ in a scene, the three fence-tree definition rules given for simple scenes can be used in a general scene to define a group of $G = k$ τ-fences with M τ-posts each, with the following interpretation. First, "τ-post" replaces the word "obstacle" everywhere. Second, an up-right edge from a τ-post $F_i(m)$ is simply a path that goes up to the top of the τ-post, then right along a τ-path until a τ-post is reached; this τ-post is $F_i(m+1)$. A down-right edge is a similar path that leads to the τ-post $F_{i+1}(m)$. With this interpretation, the algorithm FindFenceTree can be used unchanged for general scenes; only the jump procedures must be changed to handle arbitrary obstacle widths, since the vertical motions (in line 3 of JumpDownLeft and line 2 of JumpDownRight) may no longer be possible. The modified jump algorithms are described in the next two subsections, and we will show that they work correctly, i.e., that the analogues of Lemmas 5.6 and 5.7 hold. We will also show that these procedures are not expensive, i.e., that the analogues of Lemmas 5.11 and 5.12 hold. Given the correctness of the jump procedures, it is easy to verify that the Raise procedure works correctly, i.e., satisfies Lemma 5.3, and consequently that the procedure FindFenceTree does indeed find a $k \times M$ τ-fence-tree, if it exists, with the given post $F_1(1)$ as root. Furthermore, the invariant AlmostOrd also holds (Lemma 5.9) throughout the execution of FindFenceTree.

We must show that this algorithm is still "cheap" in a general scene. Note that if the robot does not encounter any small obstacles when adding the various edges, then (assuming the jump procedures are cheap) our previous arguments suffice. We begin with the fairly straightforward argument that in general the cost of going around small obstacles is not too large. To do this, we show the analogue of Lemma 5.2, namely, that the total cost of the tree edges and the cost of the path from the root to the rightmost node are both within our required bounds.

LEMMA 6.2. *Suppose there is a $k \times M\tau$-fence-tree consisting of fences $F_1, F_2, \dots, F_k$, with $M = k + \lceil \frac{2L}{\tau} \rceil$, where $\tau = L/\sqrt{nk}$. Let Δx be the x-distance from $F_1(1)$ to $F_k(M)$. Then*

(a) *The unique path in the tree from $F_1(1)$ to $F_k(M)$ has length at most $4L + 3\tau\Delta x$.*

(b) *The total length of all the edges in the fence-tree is at most $k(3L + 3\tau\Delta x)$.*

Proof. Since $k \le n$ it follows that $k\tau = kL/\sqrt{nk} \le L$. This implies $M\tau = 2L + k\tau \le 3L$.

Part (a). There are exactly $(M + k - 2)$ edges in the tree path from $F_1(1)$ to $F_k(M)$. Since the vertical portion of each edge has length τ, and the τ-path portions of the edges do not overlap in the x-direction, the total length of these edges is at most $(M + k - 2)\tau + 2\tau\Delta x + \Delta x$, which is at most $(4L + 3\tau\Delta x)$ from the inequalities above.

Part (b). Note that we can associate each edge with a unique post, namely the one at the right-end of the edge. For any given post $F_i(m)$ other than $F_1(1)$, the x-distance to its parent is at most the x-distance δx to its predecessor $F_i(m-1)$ on the same fence. So the edge associated with this post has length at most $(\tau + 2\tau\delta x + \delta x)$. The sum of the δx terms over all posts of the fence F_i is the x-distance between the first and last posts of F_i, which is at most Δx. So the total length of the edges associated with the M posts of a fence is at most $(M\tau + 2\tau\Delta x + \Delta x)$, which sums to $k(M\tau + 2\tau\Delta x + \Delta x)$ for k fences. This last expression is at most $k(3L + 3\tau\Delta x)$ from the previous inequalities. $\square$

Thus, just as in simple scenes, we need to argue only that the total cost of each jump procedure is at most a constant times the total length of the tree edges plus the heights of all τ-posts in the tree. In the next two subsections we show how these procedures can be modified to handle arbitrary obstacle widths, and we prove that they are not too expensive. As before, our approach will be to argue that for each jump procedure, no portion of the tree is charged too often for different executions of that procedure.

6.5. Modifying JumpDownRight.

To describe the modifications, it will be useful to introduce the notion of a *greedy down-left* path: it is a path that repeatedly goes "*down* till it hits an obstacle, then to the *left* corner of the obstacle." Other greedy paths are defined similarly. Also, a point (x, y) will be said to be *down-left* of another point (x_0, y_0) if $x \le x_0$ and $y \le y_0$. As in the case of simple scenes, we will find it convenient to associate an edge in the fence-tree with the post at its right end. Our modified procedure is shown in Figure 6.3, and a typical path walked while executing this procedure is shown in Figure 6.4.

We first establish the correctness of this modified procedure, i.e., the analogue of Lemma 5.7.

LEMMA 6.3. *If the general procedure* JumpDownRight (d) *is called when the PreJDR(d) conditions hold, then after the procedure is completed, the robot will be at the last known τ-post of F_{d+1}.*

```
Procedure JumpDownRight (d)
    1. Let $m = |F_d|$ and $p = |F_{d+1}|$.
    2. Move greedy down-left until down-left of top of $F_{d+1}(p)$.
    3. Move greedy right-down until:
        • at $\tau$-post $F_{d+1}(p)$, or
        • on a tree edge. In this case, follow tree-edges to the right until at $F_{d+1}(p)$.
```

FIG. 6.3. *General procedure for jumping down from F_d to F_{d+1} when $X_d \leq X_{d+1}$.*

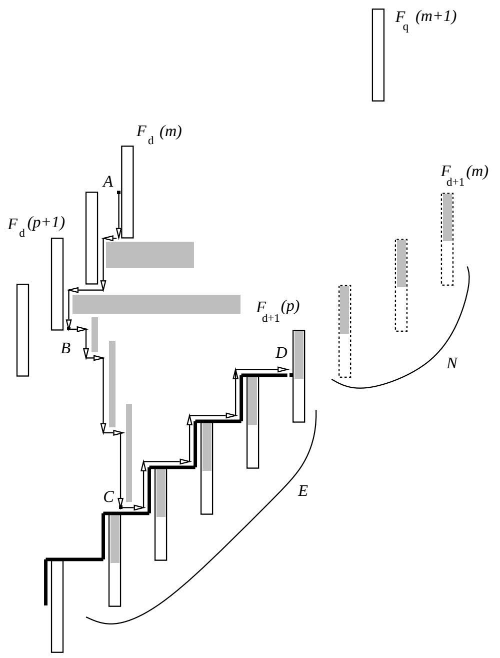


FIG. 6.4. *A use of the generalized procedure* JumpDownRight *to jump from $F_d(m)$ to $F_{d+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are nodes found so far in the tree. Dotted-boundary rectangles are posts on F_{d+1} that will be found immediately following this procedure. Thick solid lines are tree edges. The thin arrow line shows the path followed when executing the procedure.*

Proof. It will be useful to consult Figure 6.4 which shows a typical path followed while executing this procedure. From the PreJDR(d) conditions, the robot is initially at (the last known τ-post of) F_d, so the initial y-coordinate is $Y_d(m)$. The PreJDR(d) conditions $X_d \leq X_{d+1}$ and Up($d+1$) also imply that $|F_{d+1}| < |F_d|$, or $p < m$, which means that the bottom of the τ-post $F_d(m)$ is no lower than the top of the τ-post $F_{d+1}(p)$. Therefore the greedy down-left path in step 2 will not encounter any tree-edges, since even a down-right leading to the destination τ-post $F_{d+1}(p)$ can only originate at τ-post number p or lower of F_d, which must be lower than $F_d(m)$. Note that the path in step 2 is bounded on the left by the τ-posts of F_i. Also, at the end of this step, the robot's y-coordinate is the same as the top of the τ-post $F_{d+1}(p)$ to which the robot is jumping. Even if the greedy right-down path of step 3 goes

```
Procedure JumpDownLeft ($j$)
    1. Let $m = |F_j|$, $p = |F_{j+1}|$.
    2. Follow tree edges to the left until $x = X_{j+1}(p)$.
    3. Go greedy down-left until robot is either
        • down-left of top of $F_{j+1}(p)$, or
        • down-left of top of $F_u(m-1)$ for some $u \leq j$. In this case:
            (a) While $u \leq j$ do the following:
                i. Go greedy right-down until at $F_u(m-1)$ or on a tree edge.
                ii. If at a tree edge, follow edges to right until at $F_u(m-1)$.
                iii. $u := u + 1$.
            (b) Go greedy down-left until down-left of top of $F_{j+1}(p)$.
    4. Go greedy right-down until at $F_{j+1}(p)$ or on a tree edge. If at a tree edge, follow
       edges to right until at $F_{j+1}(p)$.
```

FIG. 6.5. *General procedure for jumping down from F_j to F_{j+1} when $X_j > X_{j+1}$.*

only to the right, it will hit this τ-post. In the *worst* case, the motion in step 3 is just vertically down until a tree edge (down-right or up-right) is reached. From the definition of the fence-tree, it is easy to see that following the tree edges to the right must lead to $F_{d+1}(p)$. ∎

In the lemma below, we show that the cost of all calls to JumpDownRight can be charged off to the lengths of all edges in the fence-tree.

LEMMA 6.4. *The total cost of all calls to* JumpDownRight *is at most a constant times the total length of all edges in the fence-tree, plus the heights of all τ-posts in the tree.*

Proof. As in the case of simple scenes (Lemma 5.12) we will present a charging scheme where the cost of different invocations of JumpDownRight is charged to distinct portions of the fence-tree. Consider a particular call to JumpDownRight (d) from Raise (i, q), to jump from $F_d(m)$ to $F_{d+1}(p)$ (that is, when this procedure is called, $|F_d| = m$ and $|F_{d+1}| = p$). See Figure 6.4. By a reasoning similar to the one in the proof of Lemma 5.12, we can see that there is a set N of new obstacles that will be added to F_{d+1} by the Raise ($d+1, q$) procedure that is invoked just after this invocation of JumpDownRight (d). After these new obstacles are added to F_{d+1}, $|F_{d+1}|$ would equal $|F_d| = m$. Let E be the set of edges of F_{d+1} (if any) that are followed in step 3 of JumpDownRight.

Clearly the total horizontal and vertical motion of this procedure (in steps 2 and 3) is no more than twice the total length of the edges in E plus the heights of the obstacles in N. (This bound is actually quite loose but will suffice for our purposes.) As before it is easy to argue that the sets N and E of any future call of JumpDownRight (d) will not overlap with the corresponding sets of the present call. ∎

6.6. Modifying JumpDownLeft.

The general procedure JumpDownLeft is shown in Figure 6.5, and a sample path executed by that procedure is shown in Figure 6.6.

We first establish the correctness of this general procedure, i.e., the analogue of Lemma 5.6.

LEMMA 6.5. *If the general procedure* JumpDownLeft (j) *is called under the conditions PreJDL(j), then the procedure terminates with the robot at the last known τ-post of F_{j+1}.*

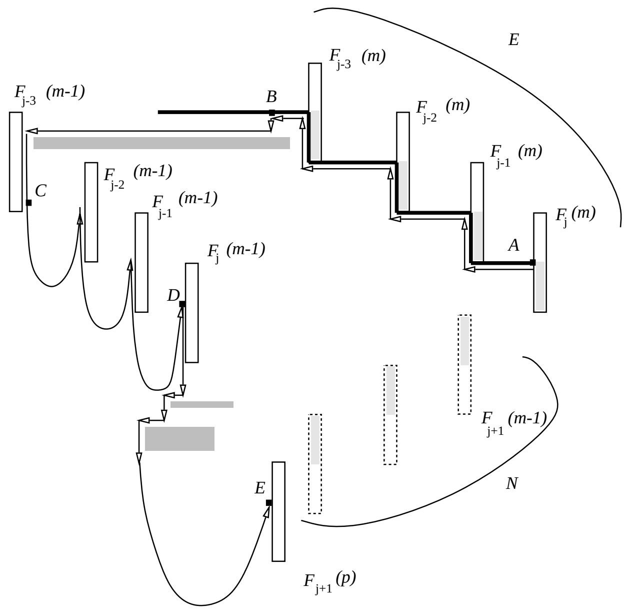


FIG. 6.6. *A use of procedure* JumpDownLeft *to jump from $F_j(m)$ to $F_{j+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are posts found so far in the tree. Dotted-boundary rectangles are posts of F_{j+1} that will be found immediately following this procedure. Thick solid lines are tree edges. The thin arrow line is the path followed when executing the procedure. Curved arrows represent motions executed during steps* 3*(a)(i) and* 4 *of the procedure.*

Proof. Let m and p be the quantities defined in the procedure. See Figure 6.6 for a typical path of this procedure. The PreJDL(j) condition $X_{j+1} < X_j$ implies that following tree edges to the left in step 2 will lead to a point where $x = X_{j+1} = X_{j+1}(p)$. The PreJDL(j) conditions $X_{j+1} < X_j$ and Up($j+1$) imply that $p = |F_{j+1}| < m-1$, so the $(m-1)$st posts of fences F_j and above are higher than the top of the destination post $F_{j+1}(p)$. Suppose F_r is the highest fence reached in step 2, i.e., the last edge retraced has its right end on a τ-post of F_r. By the AlmostOrd invariant, no fence can have more than one post to the right of a lower one, so only the last edge retraced in step 2 can be an up-right edge; the others must be *down-right* edges. The same invariant implies that the x-coordinate of the robot at the end of step 2 (i.e., $X_{i+1}(p)$) lies in the interval $[X_u(m-1), X_u(m)]$, for each $u = r, r+1, \dots, j$. This means that in step 3 when the robot goes greedily down-left, the robot must either reach a point down-left of the top of some post $F_u(m-1)$, or else reach a point down-left of $F_{j+1}(p)$, the destination post. In the former case, the robot enters the while loop of step 3(a). We claim that in each iteration of this while loop, the robot jumps down to post $(m-1)$ of the next lower fence until it reaches $F_j(m-1)$. This motion is similar to that of procedure JumpDownRight, so we can reason as in the proof of Lemma 6.3 (correctness of JumpDownRight) to show this claim. Once the robot is at $F_j(m-1)$, step 3(b) will take the robot to a point down-left of the top of $F_{j+1}(p)$. Finally step 4 is similar to step 3 in the general version of JumpDownRight, so by reasoning as in the proof of Lemma 6.3, we can show that the robot will eventually be at $F_{j+1}(p)$. □

LEMMA 6.6. *The total cost of all calls to* JumpDownLeft *is at most a constant times the total length of all the tree edges, plus the heights of all the τ-posts in the fence-tree.*

Proof. Consider a call to JumpDownLeft (j). As in the case of simple scenes, we can use a charging scheme where different calls to JumpDownLeft will be charged to distinct portions of the fence-tree. As in the proof of Lemma 5.11, let E be the set of edges followed in step 2 of the procedure, and let N be the set of $m-1-p$ "new" obstacles that will be added to F_{j+1} by the Raise procedure invoked just after this procedure completes. By the same reasoning as in that Lemma, the E and N sets of this call to JumpDownLeft will not overlap with the corresponding sets of any future call. The edge-following motion of the robot during step 2 of the procedure can be charged to the edge-set E, and so we only need to account for the motions in the remaining steps of the procedure.

If step 3 takes the robot vertically down to the τ-post $F_{j+1}(p)$, then as in the case of simple scenes this vertical motion can be charged to the edge-set E and the set N. In this case we are done with the proof. However, in general scenes there are two possibilities for the motion of the robot during this step:

- (A) The robot may reach some point down-left of the top of $F_{j+1}(p)$. Let (x_1, y_1) be the coordinates of the robot at this point. In this case, step 3 is done, and we charge the vertical motion to the total heights of the τ-posts in N plus the lengths of the edges in E. Also in this case, while going greedy down-left, the robot cannot go to the left of post $F_j(p+1)$, since the bottom of this post has the same y-coordinate as the top of $F_{j+1}(p)$. (If the robot were forced to go to the left of $F_j(p+1)$, case (B) below would occur.) Therefore the horizontal motion in this case is no more than $X_{j+1}(p) - X_j(p+1)$, which in turn is no more than the lengths of the portions of the edges of F_{j+1} that lie between $x = X_j(p+1)$ and $x = X_{j+1}(p)$; we can charge the horizontal motion to this edge-set E_1. We now claim that the edge-set E_1 of this call to JumpDownLeft (j) will not overlap with the E_1 set of any future call to JumpDownLeft (j). As we argued in the proof of Lemma 5.11, by the time any future call to JumpDownLeft (j) is made, F_{j+1} would have at least $m-1$ obstacles. Since $X_j(m) > X_{j+1}(p)$, the E_1 set of such a call would lie entirely to the right of $x = X_j(m+1) > X_j(m)$, which is to the right of $x = X_{j+1}(p)$ (the right-boundary of the present E_1 set). Therefore the edge-sets E_1 of different calls to JumpDownLeft (j) cannot overlap.
- (B) The robot may reach some point down-left of the top of post $(m-1)$ of some fence F_j or above. In this case the robot enters the while loop of step 3(a), and repeatedly moves down to the $(m-1)$st post of the next lower fence until it reaches $F_j(m-1)$. The motion in each iteration of this while loop is similar to the motion in the JumpDownRight procedure. The only difference is that instead of going from post number m of one fence to a lower-numbered obstacle of the next lower fence, in this case the robot goes to the *same*-numbered post of the next lower fence. Consider some iteration of this while loop, where the robot is jumping down from $F_u(m-1)$ to $F_{u+1}(m-1)$. Let E_2 be the set of edges of F_{u+1} contained in the region between the lines $x = X_u(m-1)$ and $x = X_{u+1}(m-1)$. The vertical motion in step 3(a)(i) is no more than the height of the destination post, plus the heights of the posts associated with the edges E_2. The horizontal motion in steps 3(a)(i) and 3(a)(ii) is no more than the lengths of the edges in E_2. In any future call

to JumpDownLeft the E_2 set corresponding to fence F_{u+1} must lie entirely to the right of $F_u(m)$, which in turn is to the right of $F_{u+1}(m-1)$ (the right-boundary of the present E_2 set). Thus in any future call to JumpDownLeft, the E_2 set associated with F_{u+1} will not overlap with the E_2 set of the present call. Similarly, as argued in the case of simple scenes, in any future call to JumpDownLeft from post m of a fence below F_j the edge-set E followed in step 2 will lie entirely to the right of $X_j(m)$, so none of the E_2 sets of that call will overlap with those of the present call. After exiting the while loop of step 3(a), the robot executes step 3(b): go greedy down-left until down-left of top of $F_{j+1}(p)$. The charging for this step is similar to case (A) above, except that we do not need to charge the vertical motion to the edge-set E since this step starts at $F_j(m-1)$. The same argument as in case (A) establishes that the charge-sets for this step will not overlap with the corresponding charge-sets of any future call to JumpDownLeft.

Finally in step 4, the robot performs a motion similar to the one in the (generalized) procedure JumpDownRight, and we can use a charging-scheme similar to the one used there. The argument to show that the portions of the tree charged for step 4 of this call to JumpDownLeft (j) do not overlap with the corresponding charge-sets of any future call is similar to the one used above for case (B) of step 3. □

7. An incremental algorithm. We describe here an improvement of our cumulative algorithm, so that the per-trip ratio on the ith trip, for all $i \le n$, is $O(\sqrt{n/i})$. Let us for simplicity say that we know L. From the earlier results in this paper, we know that by searching a distance at most $cL\sqrt{nk}$ we can find an s-t path of length at most $c'L\sqrt{n/k}$, for some constants c, c' and any $k \le n$.

Let us suppose that at the end of i trips we know an s-t-path π of length at most $c'L\sqrt{n/i}$ (for the base case, simply use the BRS algorithm). What we now want to do is to search with cost at most $cL\sqrt{n2i}$ and find a path of length at most $c'L\sqrt{n/2i}$. Let us denote by Π the path we would have traveled if we did this entire search in one trip using the algorithm of the previous sections. In order to maintain a per-trip ratio of $O(\sqrt{n/i})$, we spread the work of Π over the next i trips as follows. Each trip consists of two phases: The first is a *search* phase, where we walk an additional portion of Π of length $\frac{1}{i}cL\sqrt{n2i} = cL\sqrt{2n/i}$, starting from where we left off on the previous trip. We can always do this because the fences are in a tree structure, so that the last point in Π during the previous search can always be reached from the start point by a known short path whose length adds only a small constant factor to the total trip length. Once the search phase is completed, we "give up" and enter the *follow* phase, where we complete the trip by joining (by a greedy path) the known path π of length $c'L\sqrt{n/i}$ and following it to t. Thus our trip length is still $O(L\sqrt{n/i})$. Since in each such search-follow trip we traverse a portion of Π of length $cL\sqrt{2n/i}$, and the length of Π is at most $cL\sqrt{2ni}$, after i trips we will have completely walked the path Π. So after the first $2i$ trips we have a path of length at most $c'L\sqrt{n/2i}$. This reestablishes our invariant. Thus, we have the following theorem.

THEOREM 7.1. *There is a deterministic algorithm R that for every $i \le n$ achieves a per-trip ratio on the ith trip, $\rho_i(R, n)$, of $O(\sqrt{n/i})$.*

8. Modification for point-to-point navigation. Our algorithms can be extended to the case where t is a point rather than a wall, with the same bounds, up to constant factors, as follows. Let us assume for simplicity that the shortest path length L is known. As before, if we do not know L, we can use the standard "guessing and doubling" approach and suffer only a constant factor penalty in performance. On the first trip, the robot can get to t using the optimal point-to-point algorithms of [7] or [3], with a single-trip ratio of $O(\sqrt{n})$. Once at t, the robot creates a greedy up-left path and a greedy down-left path from t, within a window of height $4L$ centered at t. Note that the highest post in a $k \times M$ τ-fence-tree is $M\tau \leq 3L$ above the root (which is always distance L below t) and the lowest post is $k\tau \leq L$ below the root. So the robot is guaranteed to stay within a window of height $4L$ centered at t. Thus after the first trip, these greedy paths play the role of a wall; once the robot hits one of these paths, it can reach t with an additional cost that is only a low-order term in the total cost.

9. Modification for a purely tactile robot.

We assumed so far that whenever our robot hits an obstacle, it is told how far the nearest corner of the obstacle is. This information is used only to tell the robot whether or not there is a τ-post at the point of encounter. With only a constant factor penalty (see the analysis in [1]) the robot can obtain this information on its own, using the standard doubling strategy: Move up a distance 1, then down 2, then up 4, and so on, each time moving double the previous distance.

10. Conclusion and open problems.

The core result of this paper is an algorithm that performs a smooth trade-off between search effort and the goodness of the path found. This algorithm may be of interest independently of the performance-improvement problem. For instance when a robot has more time or fuel available, one would like it to spend more effort and find a better route. The fence-tree structure is central to this search algorithm. Intuitively, one can think of the fence-tree as representing the collection of those obstacles in the scene which are responsible for making the scene difficult to cross from s to t. Thus the fence-tree in a sense captures the "essence" of a scene, as far as the difficulty (i.e., cost) of crossing the scene is concerned. It would be interesting to explore whether an analogous structure can be defined in more general scenes. This might lead to a generalization of our results to such scenes.

At a higher level, our approach in designing a "learning" navigation algorithm was to start with an algorithm that achieves the above-mentioned cost/performance trade-off and convert that to a more incremental algorithm by spreading the work over several trips. This high-level idea may well be useful in designing performance-improvement algorithms for other tasks.

There are several other interesting research directions that can be explored. For instance, can randomization provide a better or simpler algorithm? For the one trip problem, the best lower bound known is $\Omega(\log\log n)$ by Karloff, Rabani, and Ravid [11], and the best upper bound is $O(n^{4/9}\log n)$ by Berman et al. [2]. What about extending our multitrip results to more general scenes? Recently, Berman and Karpinski [4] designed a randomized $O(n^{3/4})$-competitive single-trip algorithm for 2-dimensional scenes containing arbitrary convex obstacles within which a unit circle can be inscribed. Achieving an $O(\sqrt{n})$ ratio for such scenes seems considerably harder. A good first step might be to consider scenes with rectangular obstacles in arbitrary orientations (i.e., not necessarily axis-parallel).

A related problem is the question of how the robot can efficiently visit several destinations in a scene, improving performance wherever possible. One difficulty here is devising a useful performance measure (depending on the location of the destinations, one may be able to use previous information to varying degrees) that appropriately captures the essence of the problem.

REFERENCES

[1] R. Baeza-Yates, J. Culberson, and G. Rawlins, *Searching in the plane*, Inform. and Comput., 106 (1993), pp. 234–252.

[2] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosen, and M. Saks, *Randomized Robot Navigation Algorithms*, manuscript.

[3] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan, *On-line navigation in a room*, in Proceedings of the Third ACM-SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 237–249.

[4] P. Berman and M. Karpinski, *Wall Problem with Convex Obstacles*, manuscript, July 1994.

[5] M. Betke, R. Rivest, and M. Singh, *Piecemeal learning of an unknown environment*, in Proceedings of the Sixth ACM Conference on Computational Learning Theory, 1993, pp. 277–286.

[6] A. Blum and P. Chalasani, *An online algorithm for improving performance in navigation*, in Proceedings of the 34th Annual Symposium on Foundations of Computer Science, 1993, pp. 2–11.

[7] A. Blum, P. Raghavan, and B. Schieber, *Navigating in unfamiliar geometric terrain*, in Proceedings of the 23rd ACM Symposium on the Theory of Computing, 1991.

[8] P. Chalasani, *Online Performance-Improvement Algorithms*, Ph.D. thesis, Carnegie–Mellon University, Pittsburgh, PA, 1994.

[9] P. Chen, *Improving path planning with learning*, in Proceedings of the Ninth International Workshop on Machine Learning, 1992.

[10] E.G. Coffman and E.N. Gilbert, *Paths through a maze of rectangles*, Networks, 22 (1992), pp. 349–367.

[11] H. Karloff, Y. Rabani, and Y. Ravid, *Lower bounds for randomized k-server and motion-planning algorithms*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 278–288.

[12] S. Koenig and R.G. Simmons, *Complexity analysis of real-time reinforcement learning*, in Proceedings of the AAAI, 1993, pp. 99–105.

[13] V. Lumelsky, *Algorithmic issues of sensor-based robot motion planning*, in 26th IEEE Conference on Decision, 1987, pp. 1796–1801.

[14] V. Lumelsky, *Algorithmic and complexity issues of robot motion in an uncertain environment*, J. Complexity, 3 (1987), pp. 146–182.

[15] V. Lumelsky and A. Stepanov, *Dynamic path planning for a mobile automaton with limited information on the environment*, IEEE Trans. Automat. Control, 31 (1986), pp. 1058–1063.

[16] M.S. Manasse, L.A. McGeoch, and D.D. Sleator, *Competitive algorithms for online problems*, J. Algorithms, 11 (1990), pp. 208–230.

[17] C. Papadimitriou and M. Yannakakis, *Shortest paths without a map*, in Proceedings of the 16th International Collegian on Automata, Languages, and Programming, 1989.

[18] S. Thrun, *Efficient Exploration in Reinforcement Learning*, Technical Report CMU-CS-92-102, Carnegie–Mellon University, Pittsburgh, PA, 1992.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 1939–1967

ON INTERPOLATION AND AUTOMATIZATION FOR FREGE SYSTEMS[*]

MARIA LUISA BONET[†], TONIANN PITASSI[‡], AND RAN RAZ[§]

Abstract. The interpolation method has been one of the main tools for proving lower bounds for propositional proof systems. Loosely speaking, if one can prove that a particular proof system has the *feasible interpolation* property, then a generic reduction can (usually) be applied to prove lower bounds for the proof system, sometimes assuming a (usually modest) complexity-theoretic assumption. In this paper, we show that this method *cannot* be used to obtain lower bounds for Frege systems, or even for TC^0-Frege systems. More specifically, we show that unless factoring (of Blum integers) is feasible, neither Frege nor TC^0-Frege has the feasible interpolation property. In order to carry out our argument, we show how to carry out proofs of many elementary axioms/theorems of arithmetic in polynomial-sized TC^0-Frege.

As a corollary, we obtain that TC^0-Frege, as well as any proof system that polynomially simulates it, is not automatizable (under the assumption that factoring of Blum integers is hard). We also show under the same hardness assumption that the k-provability problem for Frege systems is hard.

Key words. propositional proof systems, Frege proof systems, threshold circuits, Diffie–Hellman

AMS subject classifications. 03B05, 03F20, 68Q15

PII. S0097539798353230

1. Introduction.

One of the most important questions in propositional proof complexity is to show that there is a family of propositional tautologies requiring superpolynomial-sized proofs in a Frege or extended Frege proof system. The problem is still open, and it is thus a very important question to understand which techniques can be applied to prove lower bounds for these systems, as well as for weaker systems. In recent years, the interpolation method has been one of the most promising approaches for proving lower bounds for propositional proof systems and for bounded arithmetic. Here we show that this method is not likely to work for Frege systems and some weaker systems. The basic idea behind the interpolation method is as follows.

We begin with an unsatisfiable statement of the form $F(x, y, z) = A_0(x, z) \wedge A_1(y, z)$, where z denotes a vector of shared variables, and x and y are vectors of private variables for formulas A_0 and A_1, respectively. Since F is unsatisfiable, it follows that for any truth assignment α to z, either $A_0(x, \alpha)$ is unsatisfiable or $A_1(y, \alpha)$ is unsatisfiable. An interpolation function associated with F is a Boolean function that takes such an assignment α as input, and outputs 0 only if A_0 is unsatisfiable, and 1 only if A_1 is unsatisfiable. (Note that both A_0 and A_1 can be unsatisfiable, in which case either answer will suffice.)

[*]Received by the editors December 16, 1998; accepted for publication (in revised form) June 22, 1999; published electronically April 11, 2000.
http://www.siam.org/journals/sicomp/29-6/35323.html

[†]Department of LSI, Universidad Politécnica de Cataluña, 08034 Barcelona, Spain (bonet@lsi.upc.es). The research of this author was partly supported by EU HCM network console, by ESPRIT LTR project 20244 (ALCOM-IT), and by CICYT TIC98-0410-C02-01.

[‡]Department of Computer Science, University of Arizona, Tucson, AZ 85721 (toni@cs.arizona.edu). The research of this author was supported by NSF grant CCR-9457782, US-Israel BSF grant 95-00238, and grant INT-9600919/ME-103 from NSF and MŠMT (Czech Republic).

[§]Department of Applied Mathematics, Weizmann Institute, Rehovot 76100, Israel (ranraz@wisdom.weizmann.ac.il). The research of this author was supported by US-Israel BSF grant 95-00238.

How hard is it to compute an interpolation function for a given unsatisfiable statement F as above? It has been shown, among other things, that interpolation functions are not always computable in polynomial time unless $P = NP \cap co - NP$ [M1, M2, M3]. Nevertheless, it is possible that such a procedure exists for some special cases. In particular, a very interesting and fruitful question is whether one can find (or whether there exists) a polynomial-sized circuit for an interpolation function in the case where F has a short refutation in some proof system S. We say that a proof system S admits *feasible interpolation* if, whenever S has a polynomial-sized refutation of a formula F (as above), an interpolation function associated with F has a polynomial-sized circuit. Krajíček [K2] was the first to make the connection between proof systems having feasible interpolation and circuit complexity.

There is also a monotone version of the interpolation idea. Namely, for conjunctive normal form formulas A_0 and A_1, $F = A_0(x, z) \wedge A_1(y, z)$ is *monotone* if the variables of z occur only positively in A_1 and only negatively in A_0. In this case, an associated interpolant function is monotone, and we are thus interested in finding a polynomial-sized monotone circuit for an interpolant function. We say that a proof system S admits *monotone feasible interpolation* if whenever S has a polynomial-sized refutation of a monotone F, a monotone interpolation function associated with F has a monotone polynomial-sized circuit.

Beautiful connections exist between circuit complexity and proof systems with feasible interpolation in both (monotone and nonmonotone) cases.

In the monotone case, superpolynomial lower bounds can be proven for a (sufficiently strong) proof system that admits feasible interpolation. This was presented by the sequence of papers [IPU, BPR, K1] and was first used in [BPR] to prove lower bounds for propositional proof systems. (The idea is also implicit in [Razb2].)

In short, the statement F that is used is the clique interpolation formula, $A_0(g, x) \wedge A_1(g, y)$, where A_0 states that g is a graph containing a clique of size k (where the clique is described by the x variables), and A_1 states that g is a graph that can be colored with $k - 1$ colors (where the coloring is described by the y variables). By the pigeonhole principle, this formula is unsatisfiable. However, an associated monotone interpolation function would take as input a graph g and distinguish between graphs containing cliques of size k from those that can be colored with $k - 1$ colors. By [Razb1, AB], when $k = n^{2/3}$, such a circuit is of exponential size. Thus, exponential lower bounds follow for any propositional proof system S that admits feasible monotone interpolation.

Similar ideas also work in the case where S admits feasible interpolation (but not necessarily monotone feasible interpolation). The first such result, by [Razb2], gives *explicit* superpolynomial lower bounds for (sufficiently strong) proof systems S admitting feasible interpolation, under a cryptographic assumption. In particular, it was shown that a (nonmonotone) interpolation function, associated with a certain statement expressing $P \neq NP$, is computable by polynomial-sized circuits only if there do not exist pseudorandom number generators. Therefore, lower bounds follow for any (sufficiently strong) propositional proof system that admits feasible interpolation (conditional on the cryptographic assumption that there exist pseudorandom number generators). It is also possible to prove nonexplicit superpolynomial lower bounds for a (sufficiently strong) proof system under the assumption that NP is not computable by polynomial-sized circuits.

Many researchers have used these ideas to prove lower bounds for propositional proof systems. In particular, in the last five years, lower bounds have been shown for all of the following systems using the interpolation method: resolution [BPR], cutting planes [IPU, BPR, Pud, CH], generalizations of cutting planes [BPR, K1, K3], relativized bounded arithmetic [Razb2], Hilbert's Nullstellensatz [PS], the polynomial calculus [PS], and the Lovasz–Schriver proof system [Pud3].

1.1. Automatizability and *k*-provability. As explained in the previous paragraphs, the existence of feasible interpolation for a particular proof system S gives rise to lower bounds for S. Feasible interpolation, moreover, is a very important paradigm for proof complexity (in general) for several other reasons. In this section, we wish to explain how the lack of feasible interpolation for a particular proof system S implies that S is not automatizable.

We say that a proof system S is *automatizable* if there exists a deterministic procedure D that takes as input a formula f and returns an S-refutation of f (if one exists) in time polynomial in the size of the shortest S-refutation of f. Automatizability is a crucial concept for automated theorem proving: in proof complexity we are mostly interested in the length of the shortest proof, whereas in theorem proving it is also essential to be able to find the proof. While there are seemingly powerful systems for the propositional calculus (such as extended resolution or even axiomatic set theory (ZFC)), they are scarce in theorem proving because it seems difficult to search efficiently for a short proof in such systems. In other words, there seems to be a tradeoff between proof simplicity and automatizability—the simpler the proof system, the easier it is to find the proof.

In this section, we formalize this tradeoff in a certain sense. In particular, we show that if S has no feasible interpolation, then S is not automatizable. This was first observed by Impagliazzo. The idea is to show that if S is automatizable (using a deterministic procedure D), then S has feasible interpolation.

THEOREM 1.1. *If a proof system S does not have feasible interpolation, then S is not automatizable.*

Proof. Suppose that S is automatizable, and suppose D is the deterministic procedure to find proofs, and moreover, D is guaranteed to run in time n^c, where n is the size of the shortest proof of the input formula. Let $A_0(x,z) \wedge A_1(y,z)$ be the interpolant statement, and let α be an assignment to z. We want to output an interpolant function for $A_0(x,\alpha) \wedge A_1(y,\alpha)$. First, we run D on $A_0(x,z) \wedge A_1(y,z)$ to obtain a refutation of size s. Next, we simulate D on $A_0(x,\alpha)$ for $T(s)$ steps, and return 0 if and only if D produces a refutation of $A_0(x,\alpha)$ within time $T(s)$. $T(s)$ will be chosen to be the maximum time for D to produce a refutation for a formula that has a refutation of size s; thus $T(s) = s^c$ in this case. This works because in the case where $A_1(y,\alpha)$ is satisfiable with satisfying assignment γ, we can plug γ into the refutation of $A_0(x,\alpha) \wedge A_1(y,\alpha)$ to obtain a refutation of $A_0(x,\alpha)$ of size s. Therefore S has feasible interpolation. $\square$

Thus, feasible interpolation is a simple measure that formalizes the complexity/search tradeoff: the existence of feasible interpolation implies superpolynomial lower bounds (sometimes modulo complexity assumptions), whereas the nonexistence of feasible interpolation implies that the proof system cannot be automatized.

A concept that is very closely related to automatizability is k-provability. The k-symbol provability problem for a particular Frege system S is as follows. The problem is to determine, given a propositional formula f and a number k, whether or not there is a k-symbol S proof of f. The k-line provability problem for S is to determine whether or not there is a k-line S proof of f. The k-line provability is an undecidable problem for first-order logic [B1]; the first complexity result for the k-provability problem for propositional logic was provided by Buss [B2], who proved the rather surprising fact that the k-symbol propositional provability problem is NP-complete for a particular Frege system. More recently, [ABMP] show that the k-symbol and k-line provability problems cannot be approximated to within linear factors for a variety of propositional proof systems, including resolution and all Frege systems, unless $P = NP$.

The methods in our paper show that both the k-symbol and k-line provability problems cannot be solved in polynomial time for any TC^0-Frege system, Frege system, or extended Frege system, assuming hardness of factoring (of Blum integers). More precisely, using the same idea as above, we can show that if there is a polynomial time algorithm A solving the k-provability problem for S, then S has feasible interpolation: suppose that $F = A_0(x, z) \wedge A_1(y, z)$ is the unsatisfiable statement. We first run A with $k = n, n^2, n^3, \ldots$ on F, until A first verifies that there is a size $s = |F|^c$ proof of F for some fixed value of c. Now let α be an assignment to z. As above, we run A to determine if there is an $O(s)$-symbol (or $O(s)$-line) refutation of $A_0(x, \alpha)$ and return 0 if and only if A accepts. In fact, this proof can be extended easily to show that both the k-symbol and k-line provability problems cannot be approximated to within polynomial factors for the same proof systems (TC^0-Frege, Frege, extended Frege) under the same hardness assumption.

1.2. Interpolation and one way functions. How can one prove that a certain propositional proof system S does not admit feasible interpolation? One idea, due to Krajíček and Pudlák [KP], is to use one way permutations in the following way. Let h be a one way permutation and let $A_0(x, z), A_1(y, z)$ be the following formulas.

The formula A_0:
$h(x) = z$, AND the ith bit of x is 0.
The formula A_1:
$h(y) = z$, AND the ith bit of y is 1.

Since h is one to one, $A_0(x, z) \wedge A_1(y, z)$ is unsatisfiable. Assume that A_0, A_1 can be formulated in the proof system S and that in S there exists a polynomial-sized refutation for $A_0(x, z) \wedge A_1(y, z)$. Then, if S admits feasible interpolation, it follows that given an assignment α to z there exists a polynomial-sized circuit that decides whether $A_0(x, \alpha)$ is unsatisfiable or $A_1(y, \alpha)$ is unsatisfiable. Obviously, such a circuit breaks the ith bit of the input for h. Since A_0, A_1 can be constructed for any i, all bits of the input for h can be broken. Hence, assuming that the input for h is secure, and that in the proof system S there exists a polynomial-sized refutation for $A_0 \wedge A_1$, it follows that S does not admit feasible interpolation.

A major step towards the understanding of feasible interpolation was made by Krajíček and Pudlák [KP]. They considered formulas A_0, A_1 based on the Rivest–Shamir–Adleman (RSA) cryptographic scheme and showed that unless RSA is not secure, extended Frege systems do not have feasible interpolation. It has been open, however, whether or not the same negative results hold for Frege systems and for weaker systems such as bounded depth threshold logic or bounded depth Frege.

1.3. Our results. In this paper, we prove that Frege systems, as well as constant-depth threshold logic (referred to below as TC^0-Frege), do not admit feasible interpolation, unless factoring of Blum integers is computable by polynomial-sized circuits. (Recall that Blum integers are integers P of the type $P = p_1 \cdot p_2$, where p_1, p_2 are both primes such that $p_1 \bmod 4 = p_2 \bmod 4 = 3$.) Thus our result significantly extends [KP] to weaker proof systems. In addition, our cryptographic assumption is weaker.

To prove our result, we use a variation of the ideas of [KP]. In a conversation with Moni Naor [N], he observed that the cryptographic primitive needed here is not a one way permutation as in [KP], but the more general structure of *bit commitment.* Our formulas A_0, A_1 are based on the Diffie–Hellman secret key exchange scheme [DH]. For simplicity, we state the formulas only for the least significant bit. (Our argument works for any bit.)

Informally, our propositional statement DH will be

$$DH_n = A_0(P, g, X, Y, a, b) \wedge A_1(P, g, X, Y, c, d).$$

The common variables are two integers X, Y, and P and g. P represents a number (not necessarily a prime) of length n, and g an element of the group Z_P^*. The private variables for A_0 are integers a, b, and the private variables for A_1 are integers c, d.

Informally, $A_0(P, g, X, Y, a, b)$ will say that $g^a \bmod P = X$, $g^b \bmod P = Y$, and $g^{ab} \bmod P$ is even. Similarly, $A_1(P, g, X, Y, c, d)$ will say that $g^c \bmod P = X$, $g^d mod P = Y$, and $g^{cd} \bmod P$ is odd. The statement $A_0 \wedge A_1$ is unsatisfiable since (informally) if A_0, A_1 are both true we have

$$g^{ab} \bmod P = (g^a \bmod P)^b \bmod P = X^b \bmod P$$

$$= (g^c \bmod P)^b \bmod P = g^{bc} \bmod P = (g^b \bmod P)^c \bmod P$$

$$= Y^c \bmod P = (g^d \bmod P)^c \bmod P = g^{cd} \bmod P.$$

We will show that the above informal proof can be made formal with a (polynomial-sized) TC^0-Frege proof. On the other hand, an interpolant function computes one bit of the secret key exchanged by the Diffie–Hellman procedure. Thus, if TC^0-Frege admits feasible interpolation, then all bits of the secret key exchanged by the Diffie–Hellman procedure can be broken using polynomial-sized circuits, and hence the Diffie–Hellman cryptographic scheme is not secure. Note, that it was proved that for $P = p_1 \cdot p_2$, where p_1, p_2 are both primes such that $p_1 \bmod 4 = p_2 \bmod 4 = 3$ (i.e., P is a Blum integer), breaking the Diffie–Hellman cryptographic scheme is harder than factoring P! (See [BBR] and also [Sh, Mc]).

It will require quite a bit of work to formalize the above statement and argument with a short TC^0-Frege proof. Notice that we want the size of the propositional formula expressing the Diffie–Hellman statement to be polynomially bounded in the number of binary variables. And additionally, we want the size of the TC^0-Frege proof of the statement also to be polynomially bounded. A key idea in order to define the statement and prove it efficiently is to introduce additional common variables to our propositional Diffie–Hellman statement. The bulk of the argument then involves showing how (with the aid of the auxiliary variables) one can formalize the above proof by showing that basic arithmetic facts, including the Chinese remainder theorem, can be stated and proven efficiently within TC^0-Frege.

1.4. Section description. The paper is organized as follows. In section 2, we define our TC^0-Frege system. In section 3, we define the TC^0-formulas used for the proof. In section 4, we define precisely the interpolation formulas which are based on the Diffie–Hellman cryptographic scheme. In section 5, we show how to prove our main theorem, provided we have some technical lemmas that will be proved fully in section 7. In section 6, there is a discussion and some open problems. Finally, in section 7, we prove all the technical lemmas required for the main theorem.

The unusual organization of the paper is due to the many very technical lemmas required to show the result, which are essential to the correctness of the argument but which not every reader might want to go through. Sections 1–6 give an exposition of the result, relying on the complete proofs in the technical part.

2. TC^0-Frege systems.

For clarity, we will work with a specific bounded-depth threshold logic system, which we call TC^0-Frege. However, any reasonable definition of such a system should also suffice. Our system is a sequent-calculus logical system where formulas are built up using the connectives $\vee$, $\wedge$, Th_k, $\neg$, $\oplus_0$, and $\oplus_1$. ($\mathrm{Th}_k(x)$ is true if and only if the number of 1's in x is at least k, and $\oplus_i(x)$ is true if and only if the number of 1's in x is i mod 2.)

Our system is essentially the one introduced in [MP], (which is, in turn, an extension of the system PTK introduced by Buss and Clote [BC, section 10]).

Intuitively, a family of formulas $f_1, f_2, f_3, \ldots$ has polynomial-sized TC^0-Frege proofs if each formula has a proof of size polynomial in the size of the formula, and such that every line in the proof is a TC^0 formula.

DEFINITION 2.1. *Formulas are built up using the connectives* $\wedge$, $\vee$, Th_k, $\oplus_1$, $\oplus_0$, $\neg$. *All connectives are assumed to have unbounded fan-in.* $\mathrm{Th}_k(A_1, \ldots, A_n)$ *is interpreted to be true if and only if the number of true* A_i*'s is at least* k*;* $\oplus_j(A_1, \ldots, A_n)$ *is interpreted to be true if and only if the number of true* A_i*'s is equal to* j mod 2.

The formula $\wedge(A_1, \ldots, A_n)$ denotes the logical AND of the multiset consisting of $A_1, \ldots A_n$, and similarly for $\vee$, $\oplus_j$, and Th_k. Thus commutativity of the connectives is implicit. Our proof system operates on sequents which are sets of formulas of the form $A_1, \ldots, A_i \rightarrow B_1, \ldots, B_j$. The intended meaning is that the conjunction of the A_i's implies the disjunction of the B_j's. A proof of a sequent S in our logic system is a sequence of sequents, $S_1, \ldots, S_q$, such that each sequent S_i either is an initial sequent or follows from previous sequents by one of the rules of inference, and the final sequent, S_q, is S. The size of the proof is $\sum_{1\le i\le q} \mathrm{size}(S_i)$, and its depth is $\max_{1\le i\le q}(\mathrm{depth}(S_i))$.

The *initial sequents* are of the form (1) $A \rightarrow A$, where A is any formula; (2) $\rightarrow \wedge()$; $\vee() \rightarrow$; (3) $\oplus_1() \rightarrow$; $\rightarrow \oplus_0()$; and (4) $\mathrm{Th}_k() \rightarrow$ for $k \geq 1$; $\rightarrow \mathrm{Th}_0(A_1, \ldots, A_n)$ for $n \geq 0$. The rules of inference are as follows. Note that the logical rules are defined for $n \geq 1$ and $k \geq 1$. First we have simple structural rules such as weakening (formulas can always be added to the left or to the right), contraction (two copies of the same formula can be replaced by one), and permutation (formulas in a sequent can be reordered). The remaining rules are the cut rule and logical rules, which allow us to introduce each connective on both the left side and the right side. The cut rule allows the derivation of $\Gamma, \Gamma' \rightarrow \Delta, \Delta'$ from $\Gamma, A \rightarrow \Delta$, and $\Gamma' \rightarrow A, \Delta'$.

The logical rules are as follows.

1. (Negation-left) From $\Gamma \rightarrow A, \Delta$, (for consistency) derive $\neg A, \Gamma \rightarrow \Delta$.
2. (Negation-right) From $A, \Gamma \rightarrow \Delta$, derive $\Gamma \rightarrow \neg A, \Delta$.
3. (And-left) From $A_1, \wedge(A_2, \ldots, A_n), \Gamma \rightarrow \Delta$, derive $\wedge(A_1, \ldots, A_n), \Gamma \rightarrow \Delta$.
4. (And-right) From $\Gamma \rightarrow A_1, \Delta$ and $\Gamma \rightarrow \wedge(A_2, \ldots, A_n), \Delta$, derive $\Gamma \rightarrow \wedge(A_1, \ldots, A_n), \Delta$.
5. (Or-left) From $A_1, \Gamma \rightarrow \Delta$ and $\vee(A_2, \ldots, A_n), \Gamma \rightarrow \Delta$, derive $\vee(A_1, \ldots, A_n), \Gamma \rightarrow \Delta$.
6. (Or-right) From $\Gamma \rightarrow A_1, \vee(A_2, \ldots, A_n), \Delta$, derive $\Gamma \rightarrow \vee(A_1, \ldots, A_n), \Delta$.
7. (Mod-left) From $A_1, \oplus_{1-i}(A_2, \ldots, A_n), \Gamma \rightarrow \Delta$ and $\oplus_i(A_2, \ldots, A_n), \Gamma \rightarrow A_1, \Delta$, derive $\oplus_i(A_1, \ldots, A_n)$, $\Gamma \rightarrow \Delta$.
8. (Mod-right) From $A_1, \Gamma \rightarrow \oplus_{1-i}(A_2, \ldots, A_n), \Delta$ and $\Gamma \rightarrow A_1, \oplus_i(A_2, \ldots, A_n)$,

Δ, derive $\Gamma \rightarrow \oplus_i(A_1, \ldots, A_n), \Delta$.

9. (Threshold-left) From $\mathrm{Th}_k(A_2, \ldots, A_n), \Gamma \rightarrow \Delta$ and $A_1, \mathrm{Th}_{k-1}(A_2, \ldots, A_n), \Gamma \rightarrow \Delta$, derive $\mathrm{Th}_k(A_1, \ldots, A_n), \Gamma \rightarrow \Delta$.
10. (Threshold-right) From $\Gamma \rightarrow A_1, \mathrm{Th}_k(A_2, \ldots, A_n), \Delta$, and $\Gamma \rightarrow \mathrm{Th}_{k-1}(A_2, \ldots, A_n), \Delta$, derive $\Gamma \rightarrow \mathrm{Th}_k(A_1, \ldots, A_n), \Delta$.

A TC^0 proof is a bounded-depth proof in our system of polynomial size. More formally, we have the following definitions.

DEFINITION 2.2. *Let $F = \{(\Gamma_n \rightarrow \Delta_n) : n \in N\}$ be a family of sequents. Then $\{R_n : n \in \mathbf{N}\}$ is a family of TC^0 proofs for F if there exist constants c and d such that the following conditions hold:* (1) *each R_n is a valid proof of* $(\Gamma_n \rightarrow \Delta_n)$ *in our system;* (2) *for all i, the depth of R_n is at most d; and* (3) *for all n, the size of R_n is at most* $(\mathrm{size}(\Gamma_n \rightarrow \Delta_n))^c$.

We note that we have defined a specific proof system for clarity; our result still holds for any reasonable definition of a TC^0-Frege proof. (It can be shown that our system polynomially simulates any Frege-style system.) The difference between a polynomial-sized proof in our system and a polynomial-sized TC^0 proof is similar to the difference between NC^1 and TC^0.

3. The TC^0-formulas.

In this section, we will describe some of the TC^0-formulas needed to formulate and to refute the Diffie–Hellman formula. For simplicity of the description, let us assume that we have a fixed number N which is an upper bound for the *length* of all numbers used in the refutation of the Diffie–Hellman formula. The number N will be used to define some of the formulas below. After seeing the statement and the refutation of the Diffie–Hellman formula, it will be clear that it is enough to take N to be a small polynomial in the *length* of the number P used for the Diffie–Hellman formula.

3.1. Addition and subtraction.

We will use the usual carry-save AC^0-formulas to add two n-bit numbers. Let $x = x_n, \ldots, x_1$ and $y = y_n, \ldots, y_1$ be two numbers. Then $x + y$ will denote the following AC^0-formula: There will be $n + 1$ output bits, $z_{n+1}, \ldots, z_1$. The bit z_i will equal the mod 2 sum of C_i, x_i, and y_i, where C_i is the carry bit. Intuitively, C_i is 1 if there is some bit position less than i that generates a carry that is propagated by all later bit positions until bit i. Formally, C_i is computed by $OR(R_{i(i-1)}, \ldots, R_{i1})$, where $R_{ij} = AND(P_{i-1}, \ldots, P_{j+1}, G_j)$, where $P_k = Mod_2(x_k, y_k)$, and $G_j = AND(x_j, y_j)$. (G_j is 1 if the jth bit position generates a carry, and P_k is 1 if the k^{th} bit position propagates but does not generate a carry.)

As for subtraction, let us show how to compute $z = |x - y|$. Think of x, y as N-bit numbers. Let $s = x + \overline{y} + 1$, and similarly let $t = y + \overline{x} + 1$, where $\overline{y}$ is the complement (modulo 2) of the N bits of y, and $\overline{x}$ is the complement of the N bits of x. Denote $s = s_{N+1}, s_N, \ldots, s_1$, and note that s is equal to $2^N + (x - y)$, and similarly t is equal to $2^N + (y - x)$. If $s_{N+1} = 1$, then we know that $x - y \geq 0$ and thus $s = z$. Otherwise, if $s_{N+1} = 0$, then we know that $y - x > 0$ and thus $t = z$. Thus, for any i, we can compute z_i by $(s_{N+1} \wedge s_i) \vee (\neg s_{N+1} \wedge t_i)$.

3.2. Iterated addition.

We will now describe the TC^0-formula $SUM[x_1, \ldots, x_m]$ that inputs m numbers, each n bits long, and outputs their sum $x_1 + x_2 + \cdots + x_m$ (see [CSV]). We assume that $m \leq N$. The main idea is to reduce the addition of m numbers to the addition of two numbers. Let x_i be $x_{i,n}, \ldots, x_{i,1}$ (in binary representation). Let $l = \lceil \log_2 N \rceil$. Let $r = \frac{n}{2l}$, and assume (for simplicity) that r is an integer.

Divide each x_i into r blocks, where each block has $2l$ bits, and let $S_{i,k}$ be the number in the k^{th} block of x_i. That is,

$$S_{i,k} = \sum_{j=1}^{2l} x_{i,(k-1)\cdot 2l+j} \cdot 2^{j-1}.$$

Now, each $S_{i,k}$ has $2l$ bits. Let $L_{i,k}$ be the low-order half of $S_{i,k}$, and let $H_{i,k}$ be the high-order half. That is, $S_{i,k} = H_{i,k} \cdot 2^l + L_{i,k}$.

Denote

$$H = \sum_{i=1}^{m} \sum_{k=1}^{r} H_{i,k} \cdot 2^l \cdot 2^{(k-1)2l},$$

$$L = \sum_{i=1}^{m} \sum_{k=1}^{r} L_{i,k} \cdot 2^{(k-1)2l}.$$

Then,

$$x_1 + \cdots + x_m = \sum_{i=1}^{m} \sum_{k=1}^{r} S_{i,k} \cdot 2^{(k-1)2l}$$

$$= \sum_{i=1}^{m} \sum_{k=1}^{r} H_{i,k} \cdot 2^l \cdot 2^{(k-1)2l} + \sum_{i=1}^{m} \sum_{k=1}^{r} L_{i,k} \cdot 2^{(k-1)2l} = H + L.$$

Hence, we just have to show how to compute the numbers H, L. Let us show how to compute L; the computation of H is similar.

Denote $L_k = \sum_{i=1}^{m} L_{i,k}$. Then

$$L = \sum_{k=1}^{r} L_k \cdot 2^{(k-1)2l}.$$

Since each $L_{i,k}$ is of length l, each L_k is of length at most $l + \log_2 m$, which is at most $2l$. Hence, the bits of L are just the bits of the L_k's combined. That is, $L = L_r, L_{r-1}, \ldots, L_1$.

As for the computation of the L_k's, note that since each L_k is a polysize sum of logarithmic length numbers, it can be computed using polysize threshold gates.

3.3. Modular arithmetic. Next, we describe our TC_0-formulas that compute the quotient and remainder of a number z modulo p, where z is of length n. The remainder and the inputs for the remainder and the quotient formulas are as follows:

1. the number z,
2. numbers $p_1, p_2, \ldots, p_n$,
3. numbers k_i and r_i for all $1 \leq i \leq n$.

The intended values for the variables k_i and r_i are such that $2^i = p \cdot k_i + r_i$, where $0 \leq r_i < p$ for all $1 \leq i \leq n$. The intended values for the variables p_i are $i \cdot p$.

Suppose that $z = kp + r$, where $0 \leq r < p$, and assume that the input variables k_i, r_i, and p_i take the right values. Then our formula $[z]_p$ will output r, and our formula $div_p(z)$ will output k. The formulas are computed as follows.

Let $z = z_n, \ldots, z_1$; i.e., $z = \sum_{i=1}^{n} 2^{i-1} z_i$. Suppose that the k_i, r_i, and p_i variables satisfy $2^i = k_i \cdot p + r_i$, where $0 \leq r_i < p$, and $p_i = i \cdot p$ for all $1 \leq i \leq n$. Then z satisfies

$$z = \sum_{i=1}^{n} 2^{i-1} z_i = \sum_{i=1}^{n} (p \cdot k_{i-1} + r_{i-1}) z_i$$

$$= p \cdot \sum_{i=1}^{n} k_{i-1} \cdot z_i + \sum_{i=1}^{n} r_{i-1} \cdot z_i.$$

Denote $s = \sum_{i=1}^{n} r_{i-1} \cdot z_i$, and let l be such that $l \cdot p \leq s < (l+1) \cdot p$. Then $[z]_p = s - l \cdot p$ and can therefore be computed by

$$[z]_p = SUM_{i=1}^{n} [r_{i-1} \cdot z_i] - p \cdot l.$$

$div_p(z)$ is computed by $SUM_{i=1}^{n}[k_{i-1} \cdot z_i] + l$.

Notice that if the k_i, r_i, and p_i's are not such that $2^i = k_i \cdot p + r_i$, $0 \leq r_i < p$, and $p_i = i \cdot p$, then the formulas are not required to compute the correct values of the quotient or remainder and can give an arbitrary answer.

3.4. Product and iterated product. We will write $x \cdot y$ to denote the formula $SUM_{i,j}[2^{i+j-2} x_i y_j]$, computing the product of two n-bit numbers x and y. By $2^{i+j-2} x_i y_j$, we mean 2^{i+j-2} if both x_i and y_j are true, and 0 otherwise.

Last, we will describe our TC^0-formula for computing the iterated product of m numbers. This formula is basically the original formula of [BCH] and is articulated as a TC^0-formula in [M].

The iterative product $PROD[z_1, \ldots, z_m]$ gives the product of $z_1, \ldots, z_m$, where each z_i is of length n, and we assume that m, n are both bounded by N. The basic idea is to compute the product modulo small primes using iterated addition and then to use the constructive chinese remainder theorem to construct the actual product from the product modulo small primes.

Let Q be the product of the first t primes $q_1, \ldots, q_t$, where t is the first integer that gives a number Q of length larger than N^2. Since $q_1, \ldots, q_t$ are all larger than 2, t is at most N^2, and by the well-known bounds for the distribution of prime numbers the length of each q_j is at most $O(\log N)$. For each q_j, let g_j be a fixed generator for $Z^*_{q_j}$. Also, for each q_j, let $u_j \leq Q$ be a fixed number with the property that $u_j \bmod q_j = 1$ and for all $i \neq j$, $u_j \bmod q_i = 0$ (such a number exists by the Chinese remainder theorem). $PROD[z_1, \ldots, z_m]$ is computed as follows.

1. First we compute $r_{i,j} = [z_i]_{q_j}$ for all i, j. This is calculated using the modular arithmetic described earlier.
2. For each $1 \leq j \leq t$ we will compute $r_j = (\prod_{i=1}^{m} r_{i,j}) \bmod q_j$ as follows.
   a. Compute a_{ij} such that $(g_j^{a_{ij}}) \bmod q_j = r_{i,j}$. This is done by a table lookup.
   b. Calculate $c_j = SUM_{i=1}^{m}[a_{ij}]_{(q_j-1)}$.
   c. Compute r_j such that $g_j^{c_j} \bmod q_j = r_j$. This is another table lookup.
3. Finally, compute

$$PROD[z_1, \ldots, z_m] = SUM_{j=1}^{t}[u_j \cdot r_j]_Q.$$

We will hardwire the values $u_j \cdot k$ for all $k \leq q_j$. Thus, this computation is obtained by doing a table lookup to compute $u_j \cdot r_j$ followed by an iterated sum followed by a mod Q calculation.

3.5. Equality and inequality. Often we will write $x = y$, where x and y are both vectors of variables or formulas: $x = x_1, \dots, x_n$, and $y = y_1, \dots, y_n$. When we write $x = y$, we mean the formula $\wedge_i((\neg x_i \vee y_i) \wedge (x_i \vee \neg y_i))$. We apply the same conventions when writing $\neq, <, \leq, >, \geq$.

4. The Diffie–Hellman formula.

We are now ready to formally define our propositional statement DH. DH will be the conjunction of A_0 and A_1. The common variables for the formulas will be

(a) P and g representing n-bit integers, and for every $i \leq 2n$, we will also add common variables for g^{2^i} mod P.

(b) X, Y, and for every $i \leq 2n$, we will also add common variables for X^{2^i} mod P and for Y^{2^i} mod P.

(c) We also add variables for $P_2, \dots, P_N$, and for $k_1, \dots, k_N$ and $r_1, \dots, r_N$. These variables are needed to define arithmetic modulo P (see section 3.3).

For $e \in \{0, 1\}$, denote by $g^{2^i \cdot e}$ (respectively, $X^{2^i \cdot e}$, $Y^{2^i \cdot e}$) the following: the common variable g^{2^i} mod P (respectively, X^{2^i} mod P, Y^{2^i} mod P) if $e = 1$, and 1 if $e = 0$. The formula $A_0(P, g, X, Y, a, b)$ will be the conjunction of the following TC^0-formulas:

1.
$$PROD_i \left[g^{2^i \cdot a_i}\right]_P = X,$$
which means g^a mod $P = X$.

2. For every $j \leq n$,
$$PROD_i \left[g^{2^{i+j} \cdot a_i}\right]_P = X^{2^j} \text{ mod } P,$$
which means $(g^{2^j})^a \text{mod} P = X^{2^j} \text{mod} P$. Note that from this, it is easy to prove for $e \in \{0, 1\}$,
$$PROD_i \left[g^{2^{i+j} \cdot a_i \cdot e}\right]_P = X^{2^j \cdot e}.$$

3. Similar formulas for g^b mod $P = Y$, and for $(g^{2^j})^b$ mod $P = Y^{2^j}$ mod P.

4. $PROD_{i,j}[g^{2^{i+j} \cdot a_i \cdot b_j}]_P$ is even, which means g^{ab} mod P is even.

5. For every $i \leq N$, formulas expressing $2^i = P \cdot k_i + r_i$, $1 \leq r_i < P$, and $P_i = i \cdot P$. (These formulas are added to guarantee that the modulo P arithmetic is computed correctly.)

Similarly, the formula $A_1(P, g, X, Y, c, d)$ will be the conjunction of the above formulas, but with a replaced by c, b replaced by d, and the fourth item stating that g^{cd} mod P is odd.

Note that the definition of the iterated product ($PROD$) requires the primes $q_1, \dots, q_t$ (as well as their product Q, and the numbers $u_1, \dots, u_t$), which are fixed for the length n. So we are going to hardwire the numbers $q_1, \dots, q_t, Q, u_1, \dots u_t$, as well as the correct values for the r_i's and k_i's needed for the modulo q_j arithmetic for each one of these numbers.

5. A TC^0-Frege refutation for DH.

We want to describe a TC^0-Frege refutation for DH. As mentioned above, the proof proceeds as follows.

1. Using A_0, show that g^{ab} mod $P = X^b$ mod P.
2. Using A_1, show that X^b mod $P = g^{cb}$ mod P.

3. Show that $g^{cb} \bmod P = g^{bc} \bmod P$.
4. Using A_0, show that $g^{bc} \bmod P = Y^c \bmod P$.
5. Using A_1, show that $Y^c \bmod P = g^{dc} \bmod P$.
6. Show that $g^{dc} \bmod P = g^{cd} \bmod P$.

We can conclude from the above steps that A_0 and A_1 imply that $g^{ab} \bmod P = g^{cd} \bmod P$, but now we can reach a contradiction since A_0 states that $g^{ab} \bmod P$ is even, while A_1 states that $g^{cd} \bmod P$ is odd.

We formulate $g^{ab} \bmod P$ as

$$PROD_{i,j}\left[g^{2^{i+j}\cdot a_i\cdot b_j}\right]_P$$

and $X^b \bmod P$ as

$$PROD_j\left[X^{2^j\cdot b_j}\right]_P.$$

Thus, step 1 is formulated as

$$PROD_{i,j}\left[g^{2^{i+j}\cdot a_i\cdot b_j}\right]_P = PROD_j\left[X^{2^j\cdot b_j}\right]_P,$$

and so on.

Steps 1, 2, 4, and 5 are all virtually identical. Steps 3 and 6 follow easily because our formulas defining g^{ab} make symmetry obvious. Thus the key step is to show step 1; that is, to show how to prove $g^{ab} \bmod P = X^b \bmod P$. As mentioned above, this is formulated as follows:

$$PROD_{i,j}\left[g^{2^{i+j}\cdot a_i\cdot b_j}\right]_P = PROD_j\left[X^{2^j\cdot b_j}\right]_P.$$

We will build up to the proof that $g^{ab} \bmod P$ equals $X^b \bmod P$ by proving many lemmas concerning our basic TC^0-formulas. The final lemma that we need is the following.

LEMMA 5.1. *For every $z_{1,1},\ldots,z_{m,m'}$ and p, there are TC^0-Frege proofs of*

$$PROD_{i,j}[z_{i,j}]_p = PROD_i[PROD_j[z_{i,j}]_p]_p.$$

The proof of the lemma is given in section 7.

Using Lemma 5.1 for the first equality and point 2 from section 4 for the second equality, we can now obtain

$$PROD_{i,j}\left[g^{2^{i+j}\cdot a_i\cdot b_i}\right]_P$$

$$= PROD_j\left[PROD_i\left[g^{2^{i+j}\cdot a_i\cdot b_j}\right]_P\right]_P = PROD_j\left[X^{2^j\cdot b_j}\right]_P,$$

which proves step 1.

Hence, the main goal of section 7 is to show that the statement

$$PROD_{i,j}[z_{i,j}]_p = PROD_i[PROD_j[z_{i,j}]_p]_p$$

has a short TC^0-Frege proof. This is not trivial because our TC^0-Frege formulas are quite complicated (and in particular the formulas for iterated product and modular arithmetic). In order to prove the statement, we will need to carry out a lot of the basic arithmetic in TC^0-Frege. Before we go on to the technical part, we will try to give some intuition on how the proof of the main lemma is built.

We organized the proof as a sequence of lemmas that show how many basic facts of arithmetic can be formulated and proved in TC^0-Frege (using our TC^0-formulas). The proofs of these lemmas require careful analysis of the exact formula used for each operation. The proofs of some of these lemmas are straightforward (using the well-known TC^0-formulas), while the proofs of other lemmas require some new tricks.

In short, the main lemmas that are used for the TC^0-Frege proof of the final statement (Lemma 5.1) are the following:

1. (Lemma 7.34) For every x, y, and p, there are TC^0-Frege proofs of
$$[x \cdot y]_p = [x \cdot [y]_p]_p.$$
2. (Lemma 7.37) For every $z_1, \ldots, z_m$, and every $1 \leq k \leq m$, there are TC^0-Frege proofs of
$$PROD[z_1, \ldots, z_m] = PROD[z_1, \ldots, z_{k-1}, PROD[z_k, \ldots, z_m]].$$
3. (Lemma 7.43) For every z_1, z_2, there are TC^0-Frege proofs of
$$PROD[z_1, z_2] = z_1 \cdot z_2.$$

First, we prove some basic lemmas about addition, subtraction, multiplication, iterative-sum, less-than, and modular arithmetic. Among these lemmas will be Lemma 7.34.

The proof of Lemma 7.37 is cumbersome, but it is basically straightforward, given some basic facts about modular arithmetic. Recall that to do the iterated product we have to first compute the product modulo small primes and then combine all these products to get the right answer using iterated sum. Therefore, many basic facts of the modulo arithmetic need to be proven in advance, as well as some basic facts of the iterated sum.

Once this is done, we need to obtain the same fact modulo p (Lemma 7.44). At this point it is easier to go through the regular product, where the basic facts of modular arithmetic are easier to prove. Therefore it is important to show that TC^0-Frege can prove
$$PROD[z_1, z_2] = z_1 \cdot z_2$$
(Lemma 7.43). In our application, z_1 and z_2 will themselves be iterated products.

To show this fact we use the Chinese remainder theorem. We first prove the equality modulo small primes. This is relatively easy, since the sizes of these primes are sufficiently small ($O(\log N)$), and we can basically check all possible combinations. Once this is done, we apply the Chinese remainder theorem to obtain the equality modulo the product of the primes, and since this product is big enough, we obtain the desired result.

Our TC^0-Frege proof of the Chinese remainder theorem is different than the standard textbook one. The main fact that we need to show is that if for every j, $[R]_{q_j} = [S]_{q_j}$, then there are TC^0-Frege proofs of $[R]_Q = [S]_Q$ ($Q = q_1 \cdot \ldots \cdot q_t$). The usual proofs use some basic facts of division of primes that would be hard to implement here. Instead we prove by induction on $i < t$ that $[R]_{Q_i} = [S]_{Q_i}$, where $Q_i = \prod_{j=1}^{i} q_j$. This method allows us to work with numbers smaller than the q_i's, and again since these numbers are sufficiently small, we can verify all possibilities.

6. Discussion and open problems. We have shown that TC^0-Frege does not have feasible interpolation, assuming that the factoring of Blum integers is not efficiently computable. This implies (under the same assumptions) that TC^0-Frege as well as any system that can polynomially simulate TC^0-Frege is not automatizable. It is interesting to note that our proof and even the definition of the Diffie–Hellman formula itself is nonuniform, essentially due to the nonuniform nature of the iterated product formulas that we use. It would be interesting to know to what extent our result holds in the uniform TC^0 proof setting.

A recent paper [BDGMP] extends our results to prove that bounded-depth Frege does not have feasible interpolation assuming factoring Blum integers is sufficiently hard (actually their assumptions are stronger than ours). As a consequence bounded-depth Frege is not automatizable under somewhat weaker hardness assumptions.

An important question that is still open is whether resolution, or some restricted forms of it, is automatizable. A positive answer to this question would have important applied consequences.

7. Formal proof of the main lemma. The goal of this section is to prove Lemma 5.1. As mentioned earlier, we will build up to the proof of this lemma by showing that basic facts concerning arithmetic, multiplication, iterated multiplication, and modulus computations can be efficiently carried out in our proof system. Before we begin the formal presentation, we would like to note that we will be giving a precise description of a sequence of lemmas that are sufficient in order to carry out a full, formal proof of Lemma 5.1. However, since there are many lemmas and many of them have obvious proofs, we will describe at a meta-level what is required in order to formalize the argument in TC^0-Frege, rather than give an excessively formal TC^0-Frege proof of each lemma.

In what follows, x, y, and z will be numbers. Each one of them will denote a vector of n variables or formulas (representing the number), where $n \leq N$ and x_i (respectively, y_i, z_i) denotes the ith variable of x (representing the ith bit of the number x). When we need to talk about more than three numbers, we will write $z_1, \dots, z_m$ to represent a sequence of m n-bit numbers, (where $m, n \leq N$), and now $z_{i,j}$ is the jth variable of z_i (representing the jth bit in the ith number).

Recall that whenever we say below "there are TC^0-Frege proofs," we actually mean to say "there are polynomial-sized TC^0-Frege proofs." Some trivial properties like $x = y \wedge y = z \rightarrow x = z$ are not stated here.

7.1. Some basic properties of addition, subtraction, and multiplication.

LEMMA 7.1. *For every x, y, there are TC^0-Frege proofs of $x + y = y + x$.*

Proof of Lemma 7.1. The proof is immediate from the fact that the addition formula was defined in a symmetric way. □

LEMMA 7.2. *For every x, y, z, there are TC^0-Frege proofs of $x + (y + z) = (x + y) + z$.*

Proof of Lemma 7.2. By the definition of the addition formula, the ith bit of $((x + y) + z)$ is equal to $\oplus_1(\oplus_1(x_i, y_i, C_i(x, y)), z_i, C_i((x + y), z))$, where $C_i(x, y)$ is the carry bit going into the ith position, when we add x and y, and $C_i((x + y), z)$ is similarly defined to be the carry bit going into the ith position when we add $(x + y)$ and z.

Using basic properties of $\oplus_1$ and the above definitions, there is a simple TC^0-Frege proof that if $\oplus_1(C_i(x, y), C_i((x + y), z)) = \oplus_1(C_i(y, z), C_i(x, (y + z)))$, then it follows that $((x+y)+z) = (x+(y+z))$. Thus it is left to show that for all $i \le n+2$,

$$\oplus_1(C_i(x,y), C_i((x+y),z)) = \oplus_1(C_i(y,z), C_i(x,(y+z))).$$

We will show how to prove the stronger equality

$$C_i(x,y) + C_i((x+y),z) = C_i(y,z) + C_i(x,(y+z)).$$

(It can be verified that this is the strongest equality possible for the four quantities $C_i(x,y), C_i((x+y),z), C_i(y,z), C_i(x,(y+z))$. That is, all six assignments for $C_i(x,y)$, $C_i((x+y),z), C_i(y,z), C_i(x,(y+z))$ that satisfy the above equality are actually possible.)

We will prove this by induction on i. For $i=1$, the carry bits going into the first position are zero, so the above identity holds trivially. To prove the above equality for i, we assume that it holds for $i-1$. We will prove the equality by considering many cases, where a particular case will assume a fixed value to each of the following seven quantities: $x_{i-1}, y_{i-1}, z_{i-1}, C_{i-1}(x,y), C_{i-1}(y,z), C_{i-1}((x+y),z)$, $C_{i-1}(x,(y+z))$, subject to the condition that $C_{i-1}(x,y) + C_{i-1}((x+y),z) = C_{i-1}(y,z) + C_{i-1}(x,(y+z))$. It is easy to check that the number of cases is 48 since there are 2 choices for x_{i-1}; 2 choices for y_{i-1}; 2 choices for z_{i-1}; and 6 choices in total for $C_{i-1}(x,y)$, $C_{i-1}((x+y),z)$, $C_{i-1}(y,z)$, and $C_{i-1}(x,(y+z))$.

Each case will proceed in the same way. We will first show how to compute $C_i(x,y), C_i(y,z), C_i((x+y),z)$, and $C_i(x,(y+z))$ using the above seven values. Then we simply verify that in all 48 cases where the inductive hypothesis holds, the equality is true.

First, we will show that

$$C_i(x,y) = 1 \leftrightarrow x_{i-1} + y_{i-1} + C_{i-1}(x,y) \ge 2.$$

This requires a proof along the following lines. If $x_{i-1} = y_{i-1} = 1$, then the left-hand side of the above statement is true since position $i-1$ generates a carry, and the right-hand side of the statement is also true. Similarly, if $x_{i-1} = y_{i-1} = 0$, then both sides of the above statement are false (since position $i-1$ absorbs a carry). The last case is when $x_{i-1} = 1$ and $y_{i-1} = 0$ (or vice-versa). In this case, position $i-1$ propagates a carry, so the ith carry bit is 1 if and only if there exists a $j < i-1$ such that the jth position generates a carry and all positions between j and $i-1$ propagate carries—but this is exactly the definition of $C_{i-1}(x,y)$. Thus, we have in this last case that both sides of the statement are true if and only if $C_{i-1}(x,y)$ is true.

Using the above fact and also that $(x+y)_i = x_i \oplus y_i \oplus C_i(x,y)$, we have that $C_i((x+y),z) = 1$ if and only if $z_{i-1} + (x_{i-1} \oplus y_{i-1} \oplus C_{i-1}(x,y)) + C_{i-1}((x+y),z) \ge 2$. Identical arguments show that $C_i(y,z) = 1$ and $C_i(x,(y+z)) = 1$ can also be computed as simple formulas of the seven pieces of information. $\square$

LEMMA 7.3. *For every x, y, there are TC^0-Frege proofs of $(x+y) - y = x$.*

Proof of Lemma 7.3. $(x+y) - y$ is computed by taking the first N bits of $(x+y) + \overline{y} + 1$. Note that by the definition of the addition formula it follows easily that all bits of $(y + \overline{y})$ are 1, and hence that $((y+\overline{y}) + 1) = 2^N$. Thus,

$$(x+y) + \overline{y} + 1 = x + ((y+\overline{y}) + 1) = x + 2^N,$$

and hence the first N bits of this number are the same as the first N bits of x. $\square$

LEMMA 7.4. *For every x, y, there are TC^0-Frege proofs of $x \ge y \rightarrow (x-y) + y = x$.*

Proof of Lemma 7.4. $(x-y)$ is computed by taking the first N bits of $x+\overline{y}+1$. By the definition of the addition formula, and since $x \geq y$, it can be proved that the $(N+1)$th bit of $x+\overline{y}+1$ is 1, and hence that

$$(x-y)+2^N = x+\overline{y}+1.$$

Therefore, as in Lemma 7.3,

$$(x-y)+y+2^N = x+\overline{y}+y+1 = x+2^N.$$

In particular, the first N bits of $(x-y)+y+2^N$ are the same as those of $x+2^N$. Thus, $(x-y)+y=x$. ∎

Lemma 7.5. *For every x, y, z, there are TC^0-Frege proofs of $x+z=y+z \rightarrow x=y$.*

Proof of Lemma 7.5. The proof follows immediately from Lemmas 7.3 and 7.2 as follows: $x=(x+z)-z=(y+z)-z=y$. ∎

Lemma 7.6. *For every z, there are TC^0-Frege proofs of*

$$z = SUM_i[2^{i-1}z_i].$$

Proof of Lemma 7.6. We need to show that for every j,

$$z_j = [SUM_i[2^{i-1}z_i]]_j.$$

This is shown by a rather tedious but straightforward proof following the definition of the formula SUM for iterated addition. Namely, we show first that

$$H = z_n..z_{n-l+1}0..0z_{n-2l}..z_{n-3l+1}0..0.....z_{2l}..z_{l+1}0..0$$

and, similarly, that

$$L = 0..0z_{n-l}..z_{n-2l+1}0..0z_{n-3l}..z_{n-4l+1}.....0..0z_l..z_1.$$

Secondly, we show that $[H+L]_j = z_j$, using the definition of $+$. This second step is not difficult because all carry bits are zero. ∎

Lemma 7.7. *For every $z_1, \ldots, z_m$, and every fixed permutation α, there are TC^0-Frege proofs of*

$$SUM[z_1, \ldots, z_m] = SUM[z_{\alpha(1)}, ..., z_{\alpha(m)}].$$

(That is, the iterated sum is symmetric.)

Proof of Lemma 7.7. The proof is immediate from the fact that the formula SUM was defined in a symmetric way. ∎

Lemma 7.8. *For every z, there are TC^0-Frege proofs of*

$$SUM[z] = z.$$

Proof of Lemma 7.8. By definition of the iterated addition formula SUM, it is straightforward to prove that

$$H = z_n..z_{n-l+1}0..0z_{n-2l}..z_{n-3l+1}0..0.....z_{2l}..z_{l+1}0..0$$

and, similarly, that

$$L = 0..0z_{n-l}..z_{n-2l+1}0..0z_{n-3l}..z_{n-4l+1}.....0..0z_l..z_1.$$

Then it is also straightforward to show, using the definition of the formula for +, that $[H+L]_j = z_j$ for every j. (Again, all carry bits are zero.) ∎

LEMMA 7.9. *For every $z_1, \ldots, z_m$, there are TC^0-Frege proofs of*

$$SUM[z_1, \ldots, z_m] = z_1 + SUM[z_2, \ldots, z_m].$$

Proof of Lemma 7.9. Recall that $SUM[z_1, \ldots, z_m]$ is computed by adding two numbers H, L. Recall that L is computed by first computing the numbers $L_k = \sum_{i=1}^m L_{i,k}$, where $L_{i,k}$ is the low-order half of the kth block of z_i. The first equality follows from Lemma 7.9, and similarly, $SUM[z_2, \ldots, z_m]$ is computed by $H' + L'$, where L' is computed by first computing the numbers $L'_k = \sum_{i=2}^m L_{i,k}$. We can also write $z_1 = H'' + L''$, where $H'' = \sum_{k=1}^r H_{1,k} \cdot 2^l \cdot 2^{(k-1)2l}$ and $L'' = \sum_{k=1}^r L_{1,k} \cdot 2^{(k-1)2l}$.

In both L_k, L'_k the sum is computed using polysize threshold gates, e.g., by using the unary representation of each $L_{i,k}$. It is therefore straightforward to prove for each k, $L_k = L'_k + L_{1,k}$, (e.g., by trying all the possibilities for $L'_k, L_{1,k}$, and proving the formula separately for each possibility).

Now consider the formula $L' + L''$. Since in this addition there is no carry flow from one block to the next one, and since the bits of L, L', L'' in each block are just the bits of $L_k, L'_k, L_{1,k}$ (respectively), we can conclude that $L = L' + L''$. Since in a similar way we can prove that $H = H' + H''$, we are now able to conclude

$$SUM[z_1, \ldots, z_m] = H + L$$

$$= (H'' + L'') + (H' + L') = z_1 + SUM[z_2, \ldots, z_m]. \quad ∎$$

LEMMA 7.10. *For every $z_1, \ldots, z_m$, there are TC^0-Frege proofs of*

$$SUM[z_1 + z_2, z_3, \ldots, z_m] = SUM[z_1, z_2, \ldots, z_m].$$

Proof of Lemma 7.10. The lemma can be proved easily from Lemmas 7.9 and 7.2 as follows:

$$SUM[z_1, \ldots, z_m] = z_1 + SUM[z_2, \ldots, z_m]$$

$$= z_1 + (z_2 + SUM[z_3, \ldots, z_m]) = (z_1 + z_2) + SUM[z_3, \ldots, z_m]$$

$$= SUM[z_1 + z_2, z_3, \ldots, z_m]. \quad ∎$$

LEMMA 7.11. *For every $z_1, \ldots, z_m$, and every $1 \le k \le m$, there are TC^0-Frege proofs of*

$$SUM[z_1, \ldots, z_{k-1}, SUM[z_k, \ldots, z_m]] = SUM[z_1, \ldots, z_m].$$

Proof of Lemma 7.11. By Lemmas 7.9, 7.10, and 7.7, we have

$$SUM[z_1, \ldots, z_{k-1}, SUM[z_k, \ldots, z_m]]$$

$$= SUM[z_1, \ldots, z_{k-1}, z_k + SUM[z_{k+1}, \ldots, z_m]]$$

$$= SUM[z_1, \ldots, z_{k-1}, z_k, SUM[z_{k+1}, \ldots, z_m]].$$

The proof now follows by repeating the same argument $m-k$ times, where Lemma 7.8 is used for the base case. $\square$

LEMMA 7.12. *For every x, y, there are TC^0-Frege proofs of*

$$x \cdot y = y \cdot x.$$

Proof of Lemma 7.12. The proof is immediate from the fact that the product formula was defined in a symmetric way. $\square$

LEMMA 7.13. *For every x, y, z, where x is a power of 2, there are TC^0-Frege proofs of*

$$x \cdot (y + z) = x \cdot y + x \cdot z.$$

Proof of Lemma 7.13. It is straightforward to prove that $2^i \cdot y$, where y is any sequence of bits, consists of adding to the end of y, i 0's. The lemma easily follows. $\square$

LEMMA 7.14. *For every $z_1, \dots, z_m$, and every x where x is a power of 2, there are TC^0-Frege proofs of*

$$x \cdot SUM[z_1, \dots, z_m] = SUM[x \cdot z_1, \dots, x \cdot z_m].$$

Proof of Lemma 7.14. The proof of this lemma is like the proof of Lemma 7.17 but uses Lemma 7.13 instead of 7.16. $\square$

LEMMA 7.15. *For every x, y, z, where x, y are powers of 2, there are TC^0-Frege proofs of*

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z.$$

Proof of Lemma 7.15. The proof is the same as the proof of Lemma 7.13. $\square$

The following three lemmas are generalizations of the previous three lemmas.

LEMMA 7.16. *For every x, y, z there are TC^0-Frege proofs of*

$$x \cdot (y + z) = x \cdot y + x \cdot z.$$

Proof of Lemma 7.16. By definition of the product formula,

$$x \cdot (y + z) = SUM_{i,j}[2^{i+j-2} x_i (y + z)_j].$$

Similarly,

$$x \cdot y + x \cdot z = SUM_{i,j}[2^{i+j-2} x_i y_j] + SUM_{i,j}[2^{i+j-2} x_i z_j].$$

By iterative application of Lemma 7.11 (also using Lemma 7.7),

$$SUM_{i,j}[2^{i+j-2} x_i (y + z)_j] = SUM_i[SUM_j[2^{i+j-2} x_i (y + z)_j]].$$

Similarly (also using Lemmas 7.9 and 7.10),

$$SUM_{i,j}[2^{i+j-2} x_i y_j] + SUM_{i,j}[2^{i+j-2} x_i z_j]$$

$$= SUM_i[SUM_j[2^{i+j-2} x_i y_j + 2^{i+j-2} x_i z_j]],$$

and in the same way (using the same lemmas)

$$SUM_i[SUM_j[2^{i+j-2}x_iy_j + 2^{i+j-2}x_iz_j]]$$

$$= SUM_i[SUM_j[2^{i+j-2}x_iy_j] + SUM_j[2^{i+j-2}x_iz_j]].$$

Thus, we have to prove

$$SUM_i[SUM_j[2^{i+j-2}x_i(y+z)_j]] = SUM_i[SUM_j[2^{i+j-2}x_iy_j] + SUM_j[2^{i+j-2}x_iz_j]].$$

We will prove this by proving that for every i,

$$SUM_j[2^{i+j-2}x_i(y+z)_j] = SUM_j[2^{i+j-2}x_iy_j] + SUM_j[2^{i+j-2}x_iz_j].$$

If $x_i = 0$, this is trivial. Otherwise, $x_i = 1$, and using Lemmas 7.15, 7.14, and 7.6 we have

$$SUM_j[2^{i+j-2}x_i(y+z)_j] = 2^{i-1}SUM_j[2^{j-1}(y+z)_j] = 2^{i-1}\cdot(y+z).$$

In the same way (also using Lemma 7.13),

$$SUM_j[2^{i+j-2}x_iy_j] + SUM_j[2^{i+j-2}x_iz_j] = 2^{i-1}SUM_j[2^{j-1}y_j] + 2^{i-1}SUM_j[2^{j-1}z_j]$$

$$= 2^{i-1}\cdot y + 2^{i-1}\cdot z = 2^{i-1}\cdot(y+z). \qquad \square$$

LEMMA 7.17. *For every $z_1,\dots,z_m$, and every x, there are TC^0-Frege proofs of*

$$x\cdot SUM[z_1,\dots,z_m] = SUM[x\cdot z_1,\dots,x\cdot z_m].$$

Proof of Lemma 7.17. We will show that for every i,

$$\begin{aligned} x\cdot SUM[z_1,\dots,z_i] + SUM[x\cdot z_{i+1},\dots,x\cdot z_m] = x\cdot &SUM[z_1,\dots,z_{i+1}] \\ &+ SUM[x\cdot z_{i+2},\dots,x\cdot z_m]. \end{aligned}$$

The lemma then follows by the combination of all these equalities. The case $i = 0$ is proven as follows:

$$\begin{aligned} SUM[x\cdot z_1,\dots,x\cdot z_m] &= x\cdot z_1 + SUM[x\cdot z_2,\dots,x\cdot z_m] \\ &= x\cdot SUM[z_1] + SUM[x\cdot z_2,\dots,x\cdot z_m]. \end{aligned}$$

The first equality follows by applying Lemma 7.9, and the second equality by applying Lemma 7.8.

For the general step,

$$\begin{aligned} &x\cdot SUM[z_1,\dots,z_i] + SUM[x\cdot z_{i+1},\dots,x\cdot z_m] \\ &= x\cdot SUM[z_1,\dots,z_i] + x\cdot z_{i+1} + SUM[x\cdot z_{i+2},\dots,x\cdot z_m] \\ &= x\cdot(SUM[z_1,\dots,z_i] + z_{i+1}) + SUM[x\cdot z_{i+2},\dots,x\cdot z_m] \\ &= x\cdot SUM[z_1,\dots,z_{i+1}] + SUM[x\cdot z_{i+2},\dots,x\cdot z_m]. \end{aligned}$$

The first equality follows from Lemmas 7.9, the second equality follows from Lemma 7.16, and the third equality follows from Lemma 7.9. $\square$

LEMMA 7.18. *For every x, y, z, there are TC^0-Frege proofs of*

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z.$$

Proof of Lemma 7.18. We will show that $x \cdot (y \cdot z)$ is equal to $SUM_{i,j,k}[2^{i+j+k-3} x_i y_j z_k]$. The same will be true for $(x \cdot y) \cdot z$, and the lemma follows.

By the definition of the product,

$$y \cdot z = SUM_{j,k}[2^{j+k-2} y_j z_k].$$

Hence, by Lemma 7.6 and two applications of Lemma 7.17 (and freely using Lemma 7.12),

$$x \cdot (y \cdot z) = SUM_i[2^{i-1} x_i] \cdot SUM_{j,k}[2^{j+k-2} y_j z_k]$$

$$= SUM_i[(2^{i-1} x_i) \cdot SUM_{j,k}[2^{j+k-2} y_j z_k]]$$

$$= SUM_i[SUM_{j,k}[(2^{i-1} x_i) \cdot 2^{j+k-2} y_j z_k]].$$

Since it can be easily verified that $(2^{i-1} x_i) \cdot 2^{j+k-2} y_j z_k = 2^{i+j+k-3} x_i y_j z_k$, the above is equal to

$$SUM_i[SUM_{j,k}[2^{i+j+k-3} x_i y_j z_k]],$$

and by an iterative application of Lemma 7.11 (using also Lemma 7.7) the above is equal to

$$SUM_{i,j,k}[2^{i+j+k-3} x_i y_j z_k]. \quad \square$$

7.2. Some basic properties of less-than.

LEMMA 7.19. *For every x, y, there are TC^0-Frege proofs of $x > y \vee y > x \vee x = y$ and also of $x > 0 \vee x = 0$.*

Proof of Lemma 7.19. Either there is a bit i such that i is the most significant bit where x and y differ, or not. If all bits are equal, then $x = y$. But if there is i such that it is the most significant bit where they differ, then if $x_i = 1$ and $y_i = 0$, then $x > y$, and if $x_i = 0$ and $y_i = 1$, then $y > x$. $\square$

LEMMA 7.20. *For every x, y, there are TC^0-Frege proofs of $x > y \rightarrow (x - y) > 0$.*

Proof of Lemma 7.20. By Lemma 7.19, $(x - y) = 0 \vee (x - y) > 0$. Suppose for a contradiction that $x - y = 0$. Then $x = (x - y) + y = y$, and we get $x > x$ (which is easily proved to be false). So $x - y > 0$. $\square$

LEMMA 7.21. *For every x, y, z, there are TC^0-Frege proofs of $x > y \ \wedge \ y \geq z \rightarrow x > z$; and also $x \geq y \ \wedge y > z \rightarrow x > z$.*

Proof of Lemma 7.21. If $y = z$, then the proof of the first statement is obvious. Otherwise, suppose that i is the most significant bit where $x_i \neq y_i$ and that $x_i = 1$ and $y_i = 0$. Similarly, suppose that j is the most significant bit where $y_j \neq z_j$ and that $y_j = 1$ and that $z_j = 0$. If $i \geq j$, then it is easy to show that i is the most significant bit where $x_i \neq z_i$, $x_i = 1$, $z_i = 0$, and thus $x > z$. Similarly, if $j > i$, then j is the most significant bit where $x_j \neq z_j$, $x_j = 1$, $z_j = 0$, and thus $x > z$. Similar reasoning also implies the second statement in the lemma. $\square$

LEMMA 7.22. *For every x, z, there are TC^0-Frege proofs of $x + z \geq x$; and also $z > 0 \rightarrow x + z > x$.*

Proof of Lemma 7.22. If $z = 0$, then it is clear that $x + z = x$. For $z > 0$, we will show inductively for decreasing k that

$$x + SUM_{i \geq k}[2^{i-1}z_i] > x.$$

Then when $k = 1$, we have $x + SUM_i[2^{i-1}z_i] = x + z > x$, by Lemma 7.6.

Assuming that $z > 0$, let $z_{i'}$ be the most significant bit such that $z_{i'} = 1$. The base case of the induction will be to show that $x + SUM_{i \geq i'}[2^{i-1}z_i] > x$. Because $z_i = 0$ for all $i > i'$, and applying Lemma 7.8, it suffices to show that $x + z' > x$, where $z' = 2^{i'-1}z_{i'}$. There are two cases. If $x_{i'} = 0$, then $x + z'$ is equal to $x_n x_{n-1} \cdots x_{i'+1} 1 x_{i'-1} \cdots x_1$, and so clearly $x + z' > x$. The other case is when $x_{i'} = 1$. Let j be the most significant bit position greater than i' such that $x_j = 0$. One clearly exists because $x_{n+1} = 0$. Then $(x + z')_j = 1$, $x_j = 0$, and all higher bits are equal, and thus $x + z' > x$ as desired.

For the inductive step, we assume that $x + SUM_{i \geq k}[2^{i-1}z_i] > x$ and want to show that $x + SUM_{i \geq k-1}[2^{i-1}z_i] > x$. Using the same argument as in the base case, one can prove that (a) $x + SUM_{i \geq k}[2^{i-1}z_i] + 2^{k-2}z_{k-1} \geq x + SUM_{i \geq k}[2^{i-1}z_i]$. By the inductive hypothesis, (b) $x + SUM_{i \geq k}[2^{i-1}z_i] > x$. Applying Lemma 7.21 to (a) and (b), we obtain $x + SUM_{i \geq k}[2^{i-1}z_i] + 2^{k-2}z_{k-1} > x$. By Lemma 7.9, this implies $x + SUM_{i \geq k-1}[2^{i-1}z_i] > x$, as desired. □

LEMMA 7.23. *For every x, y, z there are TC^0-Frege proofs of $x > y \rightarrow x + z > y$.*

Proof of Lemma 7.23. If $z = 0$, then $x + z = x > y$. Otherwise, $z > 0 \rightarrow x + z > x$ by Lemma 7.22. Then by Lemma 7.21, $x + z > y$ as desired. □

LEMMA 7.24. *For every x, y, z there are TC^0-Frege proofs of $x > y \rightarrow x + z > y + z$.*

Proof of Lemma 7.24.

$$\begin{aligned} x + z &= (x - y) + y + z \\ &> y + z. \end{aligned}$$

The first equality follows from Lemma 7.4, and the second follows from Lemma 7.22 and the fact that $x > y \rightarrow x - y > 0$. □

LEMMA 7.25. *For every x, y, there are TC^0-Frege proofs of $y > 0 \rightarrow x \leq x \cdot y$.*

Proof of Lemma 7.25. $x \cdot y = SUM_{i,j}[2^{i+j-2}x_i y_j]$ by definition. Also, since $y > 0$, there is a bit of y that is 1, and suppose that it is y_l. Then

$$\begin{aligned} x \cdot y &= SUM_{i,j}[2^{i+j-2}x_i y_j] \\ &= SUM_i[2^{i+l-2}x_i y_l] + SUM_{i,j,j \neq l}[2^{i+j-2}x_i y_j] \\ &\geq SUM_i[2^{i+l-2}x_i y_l] \\ &= 2^{l-1}SUM_i[2^{i-1}x_i] \\ &= 2^{l-1} \cdot x \geq x. \qquad \square \end{aligned}$$

7.3. Some basic properties of modular arithmetic. Recall that the formulas for $[z]_p$ and $div_p(z)$ take as inputs not only the variables p and z, but also variables k_i, r_i (for every $1 \leq i \leq n$), and variables $p_1, \ldots, p_n$. The formulas give the right output if $2^i = p \cdot k_i + r_i$, $r_i < p$, and $p_i = i \cdot p$ (for all $1 \leq i \leq n$). So the following theorems will all have the hypothesis that the values for the variables k_i, r_i, and p_i are correct, and that there are short TC^0-Frege proofs for $2^i = p \cdot k_i + r_i$, $r_i < p$, and $p_i = i \cdot p$. We will state this hypothesis for the first lemma and omit it afterwards for simplicity. For simplicity, we will also use the notations $k_0 = 0$ and $r_0 = 1$, thus $2^0 = p \cdot k_0 + r_0$.

The lemmas will be used with either $p = P$, where P is the number used for the DH formula, or with $p = q$, where q is some fixed hardwired value (e.g., $q = q_j$ or $q = Q$, where q_j is one of the primes used for the iterated product formula, and Q is the product of all these primes). If $p = q$ for some hardwired q, then k_i, r_i, and $p_1, \ldots p_n$ can also be hardwired. Hence, their values are correct and it is straightforward to check (i.e., to prove) that the nonvariable formulas $2^i = p \cdot k_i + r_i$, $r_i < p$, and $p_i = i \cdot p$ are all correct. If $p = P$, then k_i, r_i, and $p_1, \ldots, p_n$ are inputs for the DH formula itself, and the requirements $2^i = p \cdot k_i + r_i$, $r_i < p$, and $p_i = i \cdot p$ are part of the requirements in the DH formula.

LEMMA 7.26. *Let z and p be n-bit numbers. Then there are TC^0-Frege proofs of*

$$2^i = k_i \cdot p + r_i, 0 \leq r_i < p, p_i = i \cdot p \text{ (for all } 0 \leq i \leq n) \longrightarrow z = SUM_i[(r_{i-1} + p \cdot k_{i-1})z_i].$$

Proof of Lemma 7.26. From Lemma 7.6, and if $2^i = r_i + p \cdot k_i$,

$$z = SUM_i[2^{i-1} z_i] = SUM_i[(r_{i-1} + p \cdot k_{i-1})z_i]. \qquad \square$$

LEMMA 7.27. *For every z and p, there are TC^0-Frege proofs of*

$$z = [z]_p + \mathrm{div}_p(z) \cdot p.$$

Also, the following uniqueness property has a TC^0-Frege proof: If $z = x + y \cdot p$ where $0 \leq x < p$, then $x = [z]_p$ and $y = \mathrm{div}_p(z)$.

Proof of Lemma 7.27. From the previous lemma, we can express z as $SUM_i[(r_{i-1} + p \cdot k_{i-1})z_i]$. Let l be the same as in the definition of the modulo formulas. Then

$$\begin{aligned}
[z]_p + p \cdot \mathrm{div}_p(z) &= (SUM_i[r_{i-1}z_i] - p \cdot l) + p \cdot (SUM_i[k_{i-1}z_i] + l) \\
&= (SUM_i[r_{i-1}z_i] - p \cdot l) + (p \cdot SUM_i[k_{i-1}z_i] + p \cdot l) \\
&= ((SUM_i[r_{i-1}z_i] - p \cdot l) + p \cdot l) + p \cdot SUM_i[k_{i-1}z_i] \\
&= SUM_i[r_{i-1}z_i] + p \cdot SUM_i[k_{i-1}z_i] \\
&= SUM_i[r_{i-1}z_i] + SUM_i[p \cdot k_{i-1}z_i] \\
&= SUM[SUM_i[r_{i-1}z_i], SUM_i[p \cdot k_{i-1}z_i]] \\
&= SUM[r_0 z_1, .., r_{n-1}z_n, p \cdot k_0 z_1, .., p \cdot k_{n-1}z_n] \\
&= SUM_i[(r_{i-1} + p \cdot k_{i-1})z_i] \\
&= z.
\end{aligned}$$

The first equality follows from the definitions of the formulas $[z]_p$ and $\mathrm{div}_p(z)$. The remaining equalities follow from Lemmas 7.16, 7.2, 7.4, 7.17, 7.9, 7.11, 7.10, and 7.26.

Let us now prove the uniqueness part. Suppose $z = [z]_p + \mathrm{div}_p(z) \cdot p = x + y \cdot p$, where $0 \leq x, [z]_p < p$. If $\mathrm{div}_p(z) = y$, then we are finished. But if $\mathrm{div}_p(z) > y$, then by the claim below $x \geq p$, which is a contradiction. (A similar argument holds when $\mathrm{div}_p(z) < y$.)

CLAIM 7.28. *If $x + y \cdot p = u + v \cdot p$ and $v > y$, then $x \geq p$.*

Proof of the claim. Since $v > y$, by Lemmas 7.4 and 7.20, $y + (v - y) = v$, and $v - y > 0$. Then $x + y \cdot p = u + (y + (v - y)) \cdot p$, and by Lemma 7.16, $x + y \cdot p = u + y \cdot p + (v - y) \cdot p$. By Lemma 7.5 we get that $x = u + (v - y) \cdot p$. Therefore by Lemmas 7.25 and 7.22,

$$p \leq (v - y) \cdot p \leq u + (v - y) \cdot p = x,$$

and by Lemma 7.21 we get $p \leq x$. ∎

LEMMA 7.29. *For every z, k, and p, there are TC^0-Frege proofs of*

$$[z]_p = [z + k \cdot p]_p.$$

Proof of Lemma 7.29. Let $x = z + k \cdot p$. Then $x = [x]_p + \mathrm{div}_p(x) \cdot p$, and $z = [z]_p + div_p(z) \cdot p$ (by Lemma 7.27).

So, $z + k \cdot p = x = [x]_p + \mathrm{div}_p(x) \cdot p$. Therefore, $[z]_p + \mathrm{div}_p(z) \cdot p + k \cdot p = [x]_p + \mathrm{div}_p(x) \cdot p$. By the uniqueness part of Lemma 7.27 applied to x, $[z]_p = [x]_p$. ∎

LEMMA 7.30. *For every x, y, z, and p, there are TC^0-Frege proofs of*

$$[x + y]_p = [[x]_p + [y]_p]_p,$$

$$[x + y]_p = [[x]_p + y]_p,$$

and

$$[x + y + z]_p = [[x]_p + [y]_p + z]_p.$$

Proof of Lemma 7.30. By Lemma 7.27, $x = [x]_p + \mathrm{div}_p(x) \cdot p$ and $y = [y]_p + \mathrm{div}_p(y) \cdot p$. Hence,

$$[x + y]_p = [[x]_p + [y]_p + (\mathrm{div}_p(x) + \mathrm{div}_p(y)) \cdot p]_p,$$

and by Lemma 7.29,

$$[x + y]_p = [[x]_p + [y]_p]_p.$$

A similar argument shows that $[x + y]_p = [[x]_p + y]_p$ and $[x + y + z]_p = [[x]_p + [y]_p + z]_p$. ∎

LEMMA 7.31. *For every $z_1, \dots, z_m$ and p, there are TC^0-Frege proofs of*

$$SUM[z_1, \dots, z_m]_p = [[z_1]_p + SUM[z_2, \dots, z_m]_p]_p.$$

Proof of Lemma 7.31. The lemma follows easily from Lemmas 7.9 and 7.30. ∎

LEMMA 7.32. *For every x, y, and p, there are TC^0-Frege proofs of*

$$[x]_p = [y]_p \longrightarrow [x + z]_p = [y + z]_p.$$

Proof of Lemma 7.32.

$$[x + z]_p = [[x]_p + [z]_p]_p = [[y]_p + [z]_p]_p = [y + z]_p.$$

The first equality follows from Lemma 7.30, the next equality follows from the assumption that $[x]_p = [y]_p$, and the third equality follows from Lemma 7.30. ∎

LEMMA 7.33. *For every x, y, z, and p, there are TC^0-Frege proofs of*

$$[x + z]_p = [y + z]_p \longrightarrow [x]_p = [y]_p.$$

Proof of Lemma 7.33. Assuming that $[x+z]_p = [y+z]_p$, it follows from the above Lemma 7.32 that $[x + z + (p - [z]_p)]_p = [y + z + (p - [z]_p)]_p$. The left side of the equation is equal to

$$\begin{aligned}[x + z + (p - [z]_p)]_p &= [[x]_p + [z]_p + (p - [z]_p)]_p \\ &= [[x]_p + p]_p \\ &= [x]_p.\end{aligned}$$

The first equality follows from Lemma 7.30; the second equality follows from Lemmas 7.1, 7.2, and 7.3; and the third equality follows from Lemma 7.29. Similarly, it can be shown that $[y + z + (p - [z]_p)]_p = [y]_p$ and thus the lemma follows. □

LEMMA 7.34. *For every x, y, and p, there are TC^0-Frege proofs of*

$$[x \cdot y]_p = [x \cdot [y]_p]_p.$$

Proof of Lemma 7.34.

$$\begin{aligned}[x \cdot y]_p &= [x \cdot ([y]_p + \mathrm{div}_p(y) \cdot p)]_p \\ &= [x \cdot [y]_p + x \cdot \mathrm{div}_p(y) \cdot p]_p \\ &= [x \cdot [y]_p]_p,\end{aligned}$$

where the last equality follows from Lemma 7.29. □

LEMMA 7.35. *Let A, B, C be fixed numbers such that $A = BC$. Then for every z, there are TC^0-Frege proofs of*

$$[z]_B = [[z]_A]_B.$$

This lemma will be used in situations where $A = Q$ and $B = q_i$ for some i. Recall that the numbers $Q, q_1, \ldots q_t$ are hardwired, along with their corresponding k_i, r_i, and the variables for the $j \cdot q_i$'s. Hence, we think of A, B, C as hardwired.

Proof of Lemma 7.35. Using Lemmas 7.27, 7.29, and 7.18, we get

$$[z]_B = [[z]_A + A \cdot \mathrm{div}_A(z)]_B = [[z]_A + B \cdot (C \cdot \mathrm{div}_A(z))]_B = [[z]_A]_B. \qquad □$$

7.4. Some basic properties of iterative product.

LEMMA 7.36. *For every $z_1, \ldots, z_m$ and every fixed permutation α, there are TC^0-Frege proofs of*

$$PROD[z_1, \ldots, z_m] = PROD[z_{\alpha(a)}, \ldots, z_{\alpha(m)}].$$

(That is, the iterated product is symmetric.)

Proof of Lemma 7.36. The proof of this lemma is immediate from the symmetric definition of $PROD$. □

LEMMA 7.37. *For every $z_1, \ldots, z_m$ and every $1 \le k \le m$, there are TC^0-Frege proofs of*

$$PROD[z_1, \ldots, z_m] = PROD[z_1, \ldots, z_{k-1}, PROD[z_k, \ldots, z_m]].$$

Proof of Lemma 7.37. Recall that we have hard-coded the numbers u_j, such that $u_j \bmod q_j = 1$ and for all $i \neq j$, $u_j \bmod q_i = 0$. For all primes q_j dividing Q and for all m, $1 \le m \le q_j$, we can verify the following statements: $[u_j \cdot m]_{q_j} = m$, and for all $i \neq j$, $[u_j \cdot m]_{q_i} = 0$. (Note that these statements are variable-free and hence they can be easily proven by doing a formula evaluation.)

Recall that for any k, the iterated product of the numbers $z_k, \dots, z_m$ is calculated as follows:

$$PROD[z_k, \dots, z_m] = SUM_{j=1}^{t}[u_j \cdot r_j^{[k,..,m]}]_Q,$$

where $r_j^{[k,\dots,m]}$ is computed like r_j as defined in section 3.4 but using r_{ij} only for i such that $k \leq i \leq m$.

In the same way,

$$PROD[z_1, \dots, z_{k-1}, PROD[z_k, \dots, z_m]]$$

$$= SUM_{j=1}^{t}[u_j \cdot r_j^{[1,\dots,k-1,[k,\dots,m]]}]_Q,$$

where $r_j^{[1,\dots,k-1,[k,\dots,m]]}$ is calculated as before by the following steps:

1. For $i < k$, calculate $r_{i,j} = [z_i]_{q_j}$, and also calculate $r_{*,j} = PROD[z_k, \dots, z_m]_{q_j}$.
2. For $i < k$, calculate $a_{i,j}$ such that $(g_j^{a_{i,j}}) \bmod q_j = r_{i,j}$, and also $a_{*,j}$ such that $(g_j^{a_{*,j}}) \bmod q_j = r_{*,j}$ by table lookup.
3. Calculate $c'_j = SUM[a_{1,j}, \dots, a_{k-1,j}, a_{*,j}]_{(q_j-1)}$.
4. Calculate $r_j^{[1,\dots,k-1,[k,\dots,m]]}$ such that $g^{c'_j} \bmod q_j = r_j^{[1,\dots,k-1,[k,..,m]]}$ by table lookup.

Therefore, all we have to do is to show that

$$SUM_{j=1}^{t}[u_j \cdot r_j^{[1,\dots,m]}]_Q = SUM_{j=1}^{t}[u_j \cdot r_j^{[1,\dots,k-1,[k,\dots,m]]}]_Q.$$

Hence, all we need to do to prove Lemma 7.37 is to show the following claim.

CLAIM 7.38. *For every j, there are TC^0-Frege proofs of $r_j^{[1,\dots,k-1,[k,\dots,m]]} = r_j^{[1,\dots,m]}$.*

The first step is to prove the following claim:

CLAIM 7.39. *There are TC^0-Frege proofs of $PROD[z_k, \dots, z_m]_{q_j} = r_j^{[k,\dots,m]}$.*

Claim 7.39 is proven as follows.

$$PROD[z_k, \dots, z_m]_{q_j}$$

$$= [SUM_{i=1}^{t}[u_i \cdot r_i^{[k,\dots,m]}]_Q]_{q_j} = SUM_{i=1}^{t}[u_i \cdot r_i^{[k,\dots,m]}]_{q_j}$$

$$= [[u_j \cdot r_j^{[k,\dots,m]}]_{q_j} + SUM_{i \neq j}[u_i \cdot r_i^{[k,\dots,m]}]_{q_j}]_{q_j}$$

$$= [r_j^{[k,\dots,m]} + 0]_{q_j} = r_j^{[k,\dots,m]}.$$

The second equality follows by Lemma 7.35; the third equality follows by Lemmas 7.31 and 7.7. To prove the fourth equality, we need to use the fact that $[u_j \cdot r_j^{[k,\dots,m]}]_{q_j} = r_j^{[k,\dots,m]}$, and also for all $i \neq j$, $[u_i \cdot r_i^{[k,\dots,m]}]_{q_j} = 0$. These facts can be easily proved just by checking all possibilities for $r_i^{[k,\dots,m]}$ (proving the statement for each possibility is easy, because these statements are variable-free and hence they can be easily proven by doing a formula evaluation). In order to prove the fourth equality formally, we can show that $SUM_{i\neq j}[u_i \cdot r_i^{[k,\ldots,m]}]_{q_j}$ equals zero by induction on the number of terms in the sum. $\square$

We can now turn to the proof of Claim 7.38. The quantity $r_j^{[1,\ldots,m]}$ is obtained by doing a table lookup to find the value equal to $g_j^{c_j} \bmod q_j$, where $c_j = SUM_{i=1}^m[a_{i,j}]_{(q_j-1)}$. Similarly, the quantity $r_j^{[1,\ldots,k-1,[k,\ldots,m]]}$ is obtained by doing a table lookup to find the value equal to $g_j^{c'_j} \bmod q_j$, where

$$c'_j = SUM[a_{1,j}, a_{2,j}, .., a_{k-1,j}, a_{*,j}]_{(q_j-1)}.$$

Hence, it is enough to prove that $c_j = c'_j$. Using previous lemmas,

$$c_j = [SUM_{i=1}^{k-1}[a_{i,j}]_{(q_j-1)} + SUM_{i=k}^m[a_{i,j}]_{(q_j-1)}]_{(q_j-1)},$$

$$c'_j = [SUM_{i=1}^{k-1}[a_{i,j}]_{(q_j-1)} + a_{*,j}]_{(q_j-1)}.$$

Thus, it suffices to show that

$$SUM_{i=k}^m[a_{i,j}]_{(q_j-1)} = a_{*,j}.$$

Recall that $a_{*,j}$ is the value obtained by table lookup such that $(g_j^{a_{*,j}}) \bmod q_j = r_{*,j}$, and by Claim 7.39, we have that $r_{*,j} = r_j^{[k,\ldots,m]}$. Now $r_j^{[k,\ldots,m]}$, in turn, is the value obtained by table lookup to equal $(g_j^d) \bmod q_j$, where $d = SUM_{i=k}^m[a_{i,j}]_{(q_j-1)}$.

Now it is easy to verify that our table lookup is one to one. That is, for every $x, y, z \leq q_j$, if $g_j^x \bmod q_j = z$, and $g_j^y \bmod q_j = z$, then $x = y$. Using this property (with $x = SUM_{i=k}^m[a_{i,j}]_{(q_j-1)}$, $y = a_{*,j}$ and $z = r_{*,j} = r_j^{[k,\ldots,m]}$), it follows that

$$SUM_{i=k}^m[a_{i,j}]_{(q_j-1)} = a_{*,j}. \qquad \square$$

7.5. The Chinese remainder theorem and other properties of iterative product. The heart of our proof is a TC^0-Frege proof for the following lemma, which gives the hard direction of the Chinese remainder theorem (a TC^0-Frege proof for the other direction is simpler).

LEMMA 7.40. *Let R, S be two integers, such that for every j, $[R]_{q_j} = [S]_{q_j}$. Then there are TC^0-Frege proofs of*

$$[R]_Q = [S]_Q,$$

where $q_1, \ldots, q_t$ are the fixed primes used for the $PROD$ formula (i.e., the first t primes), and Q is their product.

Proof of Lemma 7.40. Without loss of generality, we can assume that $0 \leq R, S \leq Q - 1$, and prove that $R = S$. Otherwise, define $R' = [R]_Q$, and $S' = [S]_Q$, and use Lemma 7.35 to show that for every j, $[R']_{q_j} = [S']_{q_j}$. Since $0 \leq R', S' \leq Q - 1$, we can then conclude that

$$[R]_Q = R' = S' = [S]_Q.$$

For every k, let Q_k denote $\prod_{j=1}^k q_j$. Note that the numbers Q_k can be hardwired, and that one can easily prove the following statements. (These statements are variable-free and hence they can be easily proven by doing a formula evaluation.)

For every i, $Q_{i+1} = Q_i \cdot q_{i+1}$.

The proof of the lemma is by induction on t (the number of q_j's). For $t = 1$, $Q = q_1$, and the lemma is trivial. Assume therefore by the induction hypothesis that

$$[R]_{Q_{t-1}} = [S]_{Q_{t-1}},$$

and hence

$$[[R]_{Q_{t-1}}]_{q_t} = [[S]_{Q_{t-1}}]_{q_t}.$$

Denote, $D_R = \text{div}_{Q_{t-1}}[R]$, and $D_S = \text{div}_{Q_{t-1}}[S]$. Then by Lemma 7.27,

$$R = D_R \cdot Q_{t-1} + [R]_{Q_{t-1}},$$

and

$$S = D_S \cdot Q_{t-1} + [S]_{Q_{t-1}},$$

and since we know that $[R]_{q_t} = [S]_{q_t}$, we have

$$[D_R \cdot Q_{t-1} + [R]_{Q_{t-1}}]_{q_t} = [D_S \cdot Q_{t-1} + [S]_{Q_{t-1}}]_{q_t},$$

and by $[R]_{Q_{t-1}} = [S]_{Q_{t-1}}$ and Lemma 7.33,

$$[D_R \cdot Q_{t-1}]_{q_t} = [D_S \cdot Q_{t-1}]_{q_t}.$$

Since R, S are both lower than Q, it follows that D_R, D_S are both lower than q_t. Hence, by Claim 7.41, $D_R = D_S$. Therefore, we can conclude that

$$R = D_R \cdot Q_{t-1} + [R]_{Q_{t-1}} = D_S \cdot Q_{t-1} + [S]_{Q_{t-1}} = S. \qquad \square$$

CLAIM 7.41. *For every i, there are TC^0-Frege proofs of: if $d_1, d_2 < q_i$, and $[d_1 \cdot Q_{i-1}]_{q_i} = [d_2 \cdot Q_{i-1}]_{q_i}$; then $d_1 = d_2$.*

Proof. Since $d_1, d_2 < q_i$, there are only $O(\log n)$ possibilities for d_1, d_2. Therefore, one can just check all the possibilities for d_1, d_2. Proving the statement for each possibility is easy, because these statements are variable-free and hence they can be easily proven by doing a formula evaluation.

Alternatively, one can define the function $f(x) = [x \cdot Q_{i-1}]_{q_i}$, in the domain $\{0, \dots, q_i\}$, and prove that $f(x)$ is onto the range $\{0, \dots, q_i\}$. Then, by applying the propositional pigeonhole principle, which is efficiently provable in TC^0-Frege, it follows that f is one to one. $\square$

We are now able to prove the following lemmas.

LEMMA 7.42. *For every z, there are TC^0-Frege proofs of*

$$PROD[z] = z.$$

Proof of Lemma 7.42. Recall that $PROD[z]$ is calculated as follows:

$$PROD[z] = SUM_{j=1}^{t}[u_j \cdot r_j]_Q,$$

where r_j is computed by $r_j = [z]_{q_j}$.

By Claim 7.39, for every i, $PROD[z]_{q_i} = r_i$. We thus have for every i, $PROD[z]_{q_i} = [z]_{q_i}$. The proof of the lemma now follows by Lemma 7.40. $\square$

LEMMA 7.43. *For every z_1, z_2, there are TC^0-Frege proofs of*

$$PROD[z_1, z_2] = z_1 \cdot z_2.$$

Proof of Lemma 7.43. Let us prove that for every i,

$$[PROD[z_1, z_2]]_{q_i} = [z_1 \cdot z_2]_{q_i}.$$

The proof of the lemma then follows by Lemma 7.40. By two applications of Lemma 7.34, it is enough to prove for every i,

$$[PROD[z_1, z_2]]_{q_i} = [[z_1]_{q_i} \cdot [z_2]_{q_i}]_{q_i}.$$

Recall that $PROD[z_1, z_2]$ is calculated as follows:

$$PROD[z_1, z_2] = SUM_{j=1}^{t}[u_j \cdot r_j^{[1,2]}]_Q,$$

where $r_j^{[1,2]}$ is computed like r_j as defined in section 3.4. By Claim 7.39, for every i,

$$PROD[z_1, z_2]_{q_i} = r_i^{[1,2]}.$$

Recall that $[z_1]_{q_i} = r_{1,i}$, and $[z_2]_{q_i} = r_{2,i}$. Therefore, all we have to prove is that for every i,

$$r_i^{[1,2]} = [r_{1,i} \cdot r_{2,i}]_{q_i}.$$

By the definitions: $r_{1,i} = (g_i^{a_{1,i}}) \bmod q_i$, and $r_{2,i} = (g_i^{a_{2,i}}) \bmod q_i$, and therefore,

$$[r_{1,i} \cdot r_{2,i}]_{q_i} = [(g_i^{a_{1,i}}) \bmod q_i \cdot (g_i^{a_{2,i}}) \bmod q_i]_{q_i}.$$

Also,

$$r_i^{[1,2]} = (g_i^{SUM[a_{1,i}, a_{2,i}]_{(q_i - 1)}}) \bmod q_i.$$

Therefore, one can just check all the possibilities for $a_{1,i}, a_{2,i}$. $\square$

Using the previous lemmas, we are now able to prove the following.

LEMMA 7.44. *For every $z_1, \dots, z_m$, every $k \leq m - 1$, and every p, there are TC^0-Frege proofs of*

$$PROD[z_1, \dots, z_m]_p$$

$$= PROD[z_1, \dots, z_k, PROD[z_{k+1}, \dots, z_m]_p]_p$$

(as before, given that $2^i = k_i \cdot p + r_i, 0 \leq r_i < p, p_i = i \cdot p$ for all i).

Proof of Lemma 7.44.

$$PROD[z_1,\dots,z_k,PROD[z_{k+1},\dots,z_m]_p]_p$$

$$= PROD[PROD[z_1,\dots,z_k],PROD[z_{k+1},\dots,z_m]_p]_p$$

$$= [PROD[z_1,\dots,z_k]\cdot PROD[z_{k+1},\dots,z_m]_p]_p$$

$$= [PROD[z_1,\dots,z_k]\cdot PROD[z_{k+1},\dots,z_m]]_p$$

$$= PROD[PROD[z_1,\dots,z_k],PROD[z_{k+1},\dots,z_m]]_p$$

$$= PROD[z_1,\dots,z_k,PROD[z_{k+1},\dots,z_m]]_p$$

$$= PROD[z_1,\dots,z_k,z_{k+1},\dots,z_m]_p.$$

The lemmas used for each equality in turn are Lemmas 7.37, 7.43, 7.34, 7.43, 7.37, and 7.37. □

We are now ready to prove Lemma 5.1: For every $z_{1,1},\dots,z_{m,m'}$ and p, there are TC^0-Frege proofs of

$$PROD_{i,j}[z_{i,j}]_p = PROD_i[PROD_j[z_{i,j}]_p]_p,$$

(given that $2^i = k_i\cdot p + r_i, 0\le r_i < p, p_i = i\cdot p$ for all i).

Proof of Lemma 5.1. This lemma is proved by an iterative application of the previous lemma. □

Acknowledgments. We are very grateful to Omer Reingold and Moni Naor for collaboration at early stages of this work and in particular for suggesting the use of the Diffie–Hellman cryptographic scheme. We also would like to thank Uri Feige for conversations and for his insight about extending this result to bounded-depth Frege. Part of this work was done at Dagstuhl during the complexity of Boolean functions workshop (1997).

REFERENCES

[AB] N. Alon and R. Boppana, *The monotone circuit complexity of Boolean functions*, Combinatorica, 7 (1987), pp. 1–22.

[ABMP] A. Aleknovich, S. Buss, S. Moran, and T. Pitassi, *Minimal propositional proof length is NP-hard to linearly approximate*, J. Symbolic Logic, to appear.

[BBR] E. Biham, D. Boneh, and O. Reingold, *Generalized Diffie–Hellman modulo a composite is not weaker than factoring*, Theory of Cryptography Library, Record 97-14, available from http://theory.lcs.mit.edu/tcryptol/homepage.html.

[BCH] P. Beame, S. Cook, and H. J. Hoover, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[BDGMP] M. Bonet, C. Domingo, R.Gavaldá, A.Maciel, and T.Pitassi, *Non-automatizability of bounded-depth Frege proofs*, in Proceedings IEEE Symposium on Complexity, Atlanta, GA, 1999, pp. 15–23.

[BPR] M. Bonet, T. Pitassi, and R. Raz, *Lower bounds for cutting planes proofs with small coefficients*, in Proceedings ACM Symposium on the Theory of Computing, Las Vegas, NV, 1995, pp. 575–584. Also in J. Symbolic Logic, 62 (1997), pp. 708–728.

[B1] S. Buss, *The undecidability of k-provability*, Ann. Pure Appl. Logic, 53 (1991), pp. 75–102.

[B2] S. Buss, *On Godel's theorems on lengths of proofs* II: *Lower bounds for recognizing k symbol provability*, in Feasible Mathematics II, P. Clote and J. Remmel, eds., Birkhäuser Boston, Cambridge, MA, 1995, pp. 57–90.

[BC] S. Buss and P. Clote, *Cutting planes, connectivity and threshold logic*, Arch. Math. Logic, 35 (1996), pp. 33–62.

[CH] S. Cook and A. Haken, *An exponential lower bound for the size of monotone real circuits*, J. Comput. System Sci., 58 (1999), pp. 326–335.

[CSV] A. K. Chandra, L. Stockmeyer, and U. Vishkin, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[DH] W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Trans. Inform. Theory, 22 (1976), pp. 644–654.

[IPU] R. Impagliazzo, T. Pitassi, and A. Urquhart, *Upper and lower bounds for tree-like cutting planes proofs*, in Proceedings IEEE Symposium on Logic in Computer Science, Paris, France, 1994, pp. 220–228.

[K1] J. Krajíček, *Interpolation theorems, lower bounds for proof systems and independence results for bounded arithmetic*, J. Symbolic Logic, 62 (1997), pp. 457–486.

[K2] J. Krajíček, *Lower bounds to the size of constant-depth propositional proofs*, J. Symbolic Logic, 59 (1994), pp. 73–86.

[K3] J. Krajíček, *Discretely ordered modules as a first-order extension of the cutting planes proof system*, J. Symbolic Logic, 63 (1998), pp. 1582–1596.

[KP] J. Krajíček and P. Pudlak, *Some consequences of cryptographical conjectures for S_2^1 and EF*, in Logic and Computational Complexity, D. Leivant, ed., Lecture Notes in Comput. Sci. 960, Springer-Verlag, New York, 1995, pp. 210–220.

[M] A. Maciel, *Threshold Circuits of Small Majority Depth*, Ph.D. thesis, McGill University, Montreal, PQ, Canada, 1995.

[MP] A. Maciel and T. Pitassi, *On $ACC^0[p^k]$-Frege proofs*, in Proceedings ACM Symposium on Theory of Computing, El Paso, TX, 1997, pp. 720–729.

[Mc] K. McCurley, *A key distribution system equivalent to factoring*, J. Cryptology, 1 (1988), pp. 95–105.

[M1] D. Mundici, *Complexity of Craig's interpolation*, Fund. Inform., 5 (1982), pp. 261–278.

[M2] D. Mundici, *A lower bound for the complexity of Craig's interpolants in sentential logic*, Arch. Math. Logik Grundlag, 23 (1983), pp. 27–36.

[M3] D. Mundici, *Tautologies with a unique Craig interpolant, uniform vs. non-uniform complexity*, Ann. Pure Appl. Logic, 27 (1984), pp. 265–273.

[N] M. Naor, *Personal Communication*, 1996.

[Pud] P. Pudlak, *Lower bounds for resolution and cutting planes proofs and monotone computations*, J. Symbolic Logic, 62 (1997), pp. 981–998.

[PS] P. Pudlák and J. Sgall, *Algebraic models of computation and interpolation for algebraic proof systems*, in Proof Complexity and Feasible Arithmetic, S. Buss, ed., Lecture Notes in Comput. Sci. 39, Springer-Verlag, New York, 1998, pp. 279–295.

[Pud3] P. Pudlak, *On the complexity of propositional calculus*, in Logic Colloquium 97, Cambridge University Press, Cambridge, UK, 1999, pp. 197–218.

[Razb1] A. Razborov, *Lower bounds for the monotone complexity of some Boolean functions*, Dokl. Akad. Nauk. SSSR, 281 (1985), pp. 798–801 (in Russian). English translation in Soviet Mathematics Doklady, 31 (1985), pp. 354–357.

[Razb2] A. Razborov, *Unprovability of lower bounds on the circuit size in certain fragments of bounded arithmetic*, Izv. Ross. Akad. Nauk. Ser. Mat., 59 (1995), pp. 201–224.

[Sh] Z. Shmuely, *Composite Diffie–Hellman Public-Key Generating Systems Are Hard to Break*, Technical report 356, Computer Science Department, Technion, Israel, 1985.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 1968–1996

SPACE-TIME TRADEOFFS FOR EMPTINESS QUERIES[*]

JEFF ERICKSON[†]

Abstract. We develop the first nontrivial lower bounds on the complexity of online hyperplane and halfspace emptiness queries. Our lower bounds apply to a general class of geometric range query data structures called *partition graphs*. Informally, a partition graph is a directed acyclic graph that describes a recursive decomposition of space. We show that any partition graph that supports hyperplane emptiness queries implicitly defines a halfspace range query data structure in the Fredman/Yao semigroup arithmetic model, with the same asymptotic space and time bounds. Thus, results of Brönnimann, Chazelle, and Pach imply that any partition graph of size s that supports hyperplane emptiness queries in time t satisfies the inequality $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$. Using different techniques, we improve previous lower bounds for Hopcroft's problem—Given a set of points and hyperplanes, does any hyperplane contain a point?—in dimensions four and higher. Using this offline result, we show that for online hyperplane emptiness queries, $\Omega(n^d/\operatorname{polylog} n)$ space is required to achieve polylogarithmic query time, and $\Omega(n^{(d-1)/d}/\operatorname{polylog} n)$ query time is required if only $O(n \operatorname{polylog} n)$ space is available. These two lower bounds are optimal up to polylogarithmic factors. For two-dimensional queries, we obtain an optimal continuous tradeoff $st^2 = \Omega(n^2)$ between these two extremes. Finally, using a lifting argument, we show that the same lower bounds hold for both offline and online halfspace emptiness queries in $\mathbb{R}^{d(d+3)/2}$.

Key words. lower bounds, range searching, space-time tradeoff, partition graph

AMS subject classifications. Primary, 68Q25; Secondary, 68U05

PII. S0097539798337212

1. Introduction.

A geometric range searching data structure stores a finite set of points so that we can quickly compute some function of the points inside an arbitrary region of space, or *query range*. For example, a reporting query asks for the list of points in the query range, and a counting query asks for their number. Perhaps the simplest type of query is an *emptiness* query (also called an *existential* query [6, 45]), which asks whether the query range contains any points in the set. Emptiness query data structures have been used to solve several geometric problems, including point location [20], ray shooting [2, 20, 41, 44], nearest and farthest neighbor queries [2], linear programming queries [40, 11], depth ordering [25], collision detection [19], and output-sensitive convex hull construction [40, 12].

This paper presents the first nontrivial lower bounds on the complexity of data structures that support online emptiness queries, where the query ranges are either arbitrary hyperplanes or arbitrary halfspaces. Most of our results take the form of tradeoffs between space and query time; that is, we prove lower bounds on the size of the data structure as a function of its worst-case query time, or vice versa. We also prove tradeoffs between preprocessing time and query time. These are the first such lower bounds for any range searching problem in any model of computation; earlier models do not even define preprocessing time.

[*]Received by the editors April 13, 1998; accepted for publication (in revised form) June 8, 1999; published electronically April 11, 2000. This research was done at the Computer Science Division, U.C. Berkeley, and at the Center for Geometric Computing, Department of Computer Science, Duke University, with the support of a Graduate Assistance in Areas of National Need fellowship, NSF grant DMS-9627683, and U.S. Army Research Office MURI grant DAAH04-96-1-0013. Portions of this paper were presented at the 37th IEEE Symposium on Foundations of Computer Science, 1996, and at the 13th ACM Symposium on Computational Geometry, 1997.

http://www.siam.org/journals/sicomp/29-6/33721.html

[†]Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61820 (jeffe@cs.uiuc.edu).

1.1. Previous results. Most geometric range searching lower bounds are presented in the Fredman/Yao *semigroup arithmetic* model [33, 56]. In this model, the points are given weights from a semigroup, and the goal of a range query is to determine the total weight of the points in a query region. A data structure in this model can be informally regarded as a set of precomputed partial sums in the underlying semigroup. The size of such a data structure is the number of partial sums, and the query time is the minimum number of semigroup additions performed on these partial sums to obtain the required answer. (We define the model more formally in section 2.) Lower bounds have been established in this model for several types of query ranges [10, 14, 16, 56], in many cases matching the complexities of the corresponding data structures, at least up to polylogarithmic factors.

Unfortunately, emptiness queries are completely trivial in the semigroup arithmetic model. If the query range is empty, we perform no additions; conversely, if we perform even a single addition, the query range must not be empty. Similar arguments apply to Tarjan's *pointer machine* model [54], which has been used to derive output-sensitive lower bounds for several types of reporting queries [15, 22]. In fact, the only lower bounds previously known for hyperplane emptiness queries are essentially trivial. The size of any range searching data structure must be $\Omega(n)$, since it must store each of the points. The time to answer any range query must be at least $\Omega(\log n)$, even for a fixed one-dimensional point set, in any reasonable model of computation such as algebraic decision trees [50], algebraic computation trees [8], pointer machines [54], or real RAMs [49].[1] Since there are both linear-size data structures (with large query times) [38] and data structures with logarithmic query time (with large space requirements) [17, 42], any better lower bound must take the form of a tradeoff between space and time.

The only nontrivial lower bound previously known for *any* class of online emptiness queries, in *any* model of computation, is due to Andersson and Swanson [6]. They show that $\Omega(n \log n / \log t)$ space is required to answer axis-aligned rectangular emptiness queries in time t, in the so-called layered partition model. In particular, $\Omega(n \log n / \log \log n)$ space is required to achieve polylogarithmic query time in this model (but see [13] for better upper bounds in the integer RAM model).

Very recently, Borodin, Ostrovsky, and Rabani [9] derived lower bounds for hyperplane and halfspace emptiness, nearest-neighbor, point-location, and related queries in high-dimensional spaces, in Yao's extremely general *cell probe* model [57]. (See also [46].) In the cell probe model, a data structure is an array of s cells, each containing b bits. A query is answered by probing t of these cells in sequence; the address of each probe may depend arbitrarily on the query and the results of previous probes. Borodin, Ostrovsky, and Rabani show that for hyperplane queries among n points in the d-dimensional Hamming cube $\{0,1\}^d$, either $t \log s = \Omega(\log n \log d)$ or $tb = \Omega(n^{1-\varepsilon})$ for any fixed $\varepsilon > 0$. Unfortunately, this lower bound is trivial for any fixed dimension d, since it requires $n \ll 2^d$. Even when the dimension is allowed to vary, the bound is extremely weak unless the number of probes t is nearly constant.

1.2. New results. We derive our new lower bounds with respect to a general class of geometric range query data structures called *partition graphs*. Informally, a

[1]Sublogarithmic or even constant query times can be obtained for axis-aligned rectangular queries in models of computation that allow bit manipulation and require integer inputs within a known bounded universe; see, for example, [4, 5, 13, 46, 47, 55]. No such result is known for nonorthogonal ranges, however. We will take the traditional computational-geometric view that geometric objects are represented by arbitrary real coordinates, for which bit manipulation is impossible.

Table 1
Best known upper bounds for online hyperplane emptiness queries.

Space	Preprocessing	Query Time	Source
$O(n^d/\log^d n)$	$O(n^d/\log^{d-\varepsilon} n)$	$O(\log n)$	[17, 42]
$O(n)$	$O(n^{1+\varepsilon})$	$O(n^{1-1/d})$	[42]
$O(n)$	$O(n \log n)$	$O(n^{1-1/d} \operatorname{polylog} n)$	[38]
$n \le s \le n^d/\log^d n$	$O(n^{1+\varepsilon} + s\log^{\varepsilon} n)$	$O(n/s^{1/d})$	[17, 42]

partition graph is a directed acyclic graph that describes a recursive decomposition of space into connected regions. This recursive decomposition provides a natural search structure that is used both to preprocess points and to answer queries. (A formal definition is provided in section 3.) Our model is powerful enough to describe most, if not all, known hyperplane range searching data structures.[2] Partition graphs were originally introduced to study the complexity of Hopcroft's problem—Given a set of points and hyperplanes, does any hyperplane contain a point?—and similar offline geometric searching problems [31].

We summarize our results below. In each of these results and throughout the paper, s denotes space, p denotes preprocessing time, and t denotes worst-case query time. For comparison, the best known upper bounds are listed in Table 1. For a thorough overview of range searching techniques, results, and applications, see the surveys by Matoušek [43] and by Agarwal and Erickson [1].

- Any partition graph that supports hyperplane queries requires $\Omega(n)$ space, $\Omega(n \log n)$ preprocessing time, and $\Omega(\log n)$ query time.
- Any partition graph that supports hyperplane emptiness queries implicitly defines a halfspace range query data structure in the Fredman/Yao semigroup arithmetic model, with the same time and space bounds. Thus, results of Brönnimann, Chazelle, and Pach [10] immediately imply that $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$. This lower bound applies with high probability to a randomly generated set of points. This is the first nontrivial lower bound for hyperplane emptiness queries in any model of computation.
- We generalize earlier lower bounds on the complexity of Hopcroft's problem [31] for the special case of *polyhedral* partition graphs. Specifically, we prove that in the worst case, the time to preprocess n points in $\mathbb{R}^d$ and perform k hyperplane emptiness queries is

$$\Omega(n \log k + n^{1-2/d(d+1)}k^{2/(d+1)} + n^{2/(d+1)}k^{1-2/d(d+1)} + k \log n),$$

even if the query hyperplanes are specified in advance. This lower bound was previously known in dimensions less than four and in arbitrary dimensions for offline counting and reporting queries, for arbitrary partition graphs [31].
- The previous result implies the worst-case tradeoffs $pt^{(d+2)(d-1)/2} = \Omega(n^d)$ and $pt^{2/(d-1)} = \Omega(n^{(d+2)/d})$. These lower bounds match known upper bounds up to polylogarithmic factors when $d = 2$, $p = O(n \operatorname{polylog} n)$, or $t = O(\operatorname{polylog} n)$ [38, 42]. These results apply to polyhedral partition graphs when $d \ge 4$ and to all partition graphs when $d \le 3$. Under a mild assumption about the partition graphs, these bounds imply similar tradeoffs between

[2]Difficulties in directly modeling existing range searching data structures as partition graphs are discussed in [31, section 3.5].

space and query time, improving the earlier space-time tradeoffs whenever $s = \Omega(n^{d-1})$ or $s = O(n^{1+2/(d^2+d)})$ and giving us the optimal lower bound $st^2 = \Omega(n^2)$ in two dimensions.

- Finally, using a lifting argument, we show that all of these lower bounds also apply to online and offline halfspace emptiness queries in $\mathbb{R}^{d(d+3)/2}$, at least for *semialgebraic* partition graphs. The lower bounds we obtain match existing upper bounds up to polylogarithmic factors in dimensions 2, 3, and 5. In most other cases, our bounds are extremely weak; nevertheless, they are an improvement over all previous (trivial) lower bounds for halfspace emptiness searching.

All our lower bounds for hyperplane emptiness queries in $\mathbb{R}^d$ also apply to hyperplane and halfspace counting, reporting, or semigroup queries in $\mathbb{R}^d$.

We also show that lower bounds in the semigroup arithmetic model imply lower bounds for the corresponding counting or reporting queries in the partition graph model. Thus, any partition graph supporting halfspace counting or reporting queries satisfies $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$, even in the average case. We also derive the worst-case lower bound $st^{d(d+1)/2} = \Omega(n^d)$ for hyperplane queries in the semigroup model (no lower bound was previously known in this model) and thus for hyperplane counting or reporting in the partition graph model as well. Surprisingly, in two dimensions, the lower bound $st^3 = \Omega(n^2)$ is tight in the semigroup model.

From a practical standpoint, our results are quite strong. Even for very simple query ranges, and even if we only want to know whether the range is empty, range searching algorithms based on geometric divide-and-conquer techniques cannot be significantly faster than the naïve linear-time algorithm that simply checks each point individually, unless the dimension is very small or we have almost unlimited storage.[3] For example, answering 10-dimensional hyperplane emptiness queries in, say, $\sqrt{n}$ time—quite far from the desired $O(\log n)$ time bound—requires $\Omega(n^4)$ space, which is simply impossible for large data sets. In practice, we can only afford to use linear space, and this drives the worst-case query time up to roughly $n^{9/10}$. This behavior is unfortunately not a feature of some pathological input; most of our space-time tradeoffs hold in the average case, so in fact, most point sets are this difficult to search. The lower bounds that do not (as far as we know) hold in the average case apply to extremely simple point sets, such as regular lattices.

1.3. Outline. The rest of the paper is organized as follows. In section 2, we review the definition of the semigroup arithmetic model, state a few useful results, and derive new bounds on the complexity of hyperplane queries in this model. Section 3 defines partition graphs, describes how they are used to answer hyperplane and halfspace queries, and states a few of their basic properties. We prove our new space-time tradeoff lower bounds for hyperplane emptiness queries in section 4. In section 5, we define *polyhedral covers* and develop bounds on their worst-case complexity. Using these combinatorial bounds, in section 6, we (slightly) improve earlier lower bounds on the complexity of Hopcroft's offline point-hyperplane incidence problem in dimensions four and higher. From these offline results, we derive new tradeoffs between preprocessing and query time for online hyperplane queries in section 7. Section 8 describes a reduction argument that implies lower bounds for halfspace emptiness queries in both the online and offline settings. Finally, in section 9, we offer our conclusions.

[3]This "curse of dimensionality" can sometimes be avoided by requiring only an approximation of the correct output; see, for example, [7, 24, 37].

2. Semigroup arithmetic.

2.1. Definitions. We begin by reviewing the definition of the semigroup arithmetic model, originally introduced by Fredman to study dynamic range searching problems [33] and later refined for the static setting by Yao [56].

A *semigroup* $(S, +)$ is a set S equipped with an associative addition operator $+ : S \times S \to S$. A semigroup is *commutative* if the equation $x + y = y + x$ is true for all $x, y \in S$. A *linear form* is a sum of variables over the semigroup, where each variable can occur multiple times, or equivalently, a homogeneous linear polynomial with positive integer coefficients. A commutative semigroup is *faithful* if any two identically equal linear forms have the same set of variables, although not necessarily with the same set of coefficients.[4] For example, $(\mathbb{Z}, +)$ and $(\{\text{true}, \text{false}\}, \vee)$ are faithful semigroups, but $(\{0, 1\}, + \bmod 2)$ is not faithful.

A semigroup is *idempotent* if $x + x = x$ for all semigroup elements x, and *integral* if $x \neq \alpha x$ for all semigroup elements x and all integers $\alpha > 1$. For example, the semigroups $(\{\text{true}, \text{false}\}, \vee)$ and $(\mathbb{Z}, \max)$ are idempotent, $(\mathbb{Z}, +)$ and $(\mathbb{R}, \times)$ are integral, and $(\mathbb{C}, \times)$ is neither. (All these semigroups are faithful.)

Let P be a set of n points in $\mathbb{R}^d$, let $(S, +)$ be a faithful commutative semigroup, and let $w : P \to S$ be a function that assigns a weight $w(p)$ to each point $p \in P$. For any subset $P' \subseteq P$, let $w(P') = \sum_{p \in P'} w(P)$, where addition is taken over the semigroup.[5] The range searching problem considered in the semigroup model is to preprocess P so that $w(P \cap q)$ can be calculated quickly for any query range q.

Let $x_1, x_2, \ldots, x_n$ be a set of n variables over S. A *generator* $g(x_1, \ldots, x_n)$ is a linear form $\sum_{i=1}^{n} \alpha_i x_i$, where the α_i's are nonnegative integers, not all zero. Given a class Q of query ranges (subsets of $\mathbb{R}^d$), a *storage scheme* for (P, Q, S) is a collection of generators $\{g_1, g_2, \ldots, g_s\}$ with the following property: For any query range $q \in Q$, there is a set of indices $I_q \subseteq \{1, 2, \ldots, s\}$ and an indexed set of nonnegative integers $\{\beta_i \mid i \in I_q\}$ such that

$$w(P \cap q) = \sum_{i \in I_q} \beta_i g_i(w(p_1), w(p_2), \ldots, w(p_n))$$

holds for *any* weight function $w : P \to S$. The size of the smallest such set I_q is the *query time* for q.

We emphasize that although a storage scheme can take advantage of special properties of the semigroup S or the point set P, it must work for *any* assignment of weights to P. In particular, this implies that lower bounds in the semigroup model do *not* apply to the problem of counting the number of points in the query range, even though $(\mathbb{Z}, +)$ is a faithful semigroup, since a storage scheme for that problem only needs to work for the particular weight function $w(p) = 1$ for all $p \in P$ [14]. For the same reason, even though the semigroups $(\{\text{true}, \text{false}\}, \vee)$ and $(2^P, \cup)$ are faithful, the semigroup model cannot be used to prove lower bounds for emptiness

[4]More formally, $(S, +)$ is faithful if for each $n > 0$, for any sets of indices $I, J \subseteq \{1, \ldots, n\}$ where $I \neq J$, and for all indexed sets of positive integers $\{\alpha_i \mid i \in I\}$ and $\{\beta_j \mid j \in J\}$, there are semigroup elements $s_1, s_2, \ldots, s_n \in S$ such that

$$\sum_{i \in I} \alpha_i s_i \neq \sum_{j \in J} \beta_j s_j.$$

[5]Since S need not have an identity element, we may need to assign a special value *nil* to the empty sum $w(\varnothing)$.

or reporting queries. Emptiness queries can also be formulated as queries over the one-element semigroup $(\{*\}, * + * = *)$, but this semigroup is not faithful.

For any linear form $\sum_{1=1}^{n} \alpha_i x_i$, we call the set of points $\{p_i \mid \alpha_i \neq 0\}$ its *cluster*. The faithfulness of the semigroup S implies that the union of the clusters of the generators used to determine $w(P \cap q)$ is precisely $P \cap q$ (since each $\alpha_i > 0$):

$$\bigcup_{i \in I_q} \text{cluster}(g_i) = P \cap q.$$

Thus, we can think of a storage scheme as a collection of clusters, such that any set of the form $P \cap q$ can be expressed as the (not necessarily disjoint) union of several of these clusters. The size of a storage scheme is the number of clusters, and the query time for a range q is the minimum number of clusters whose union is $P \cap q$. This is the formulation actually used to prove lower bounds in the semigroup arithmetic model[6] [10, 14, 16, 56].

Whether or not the clusters used to answer a query must be disjoint depends on the semigroup. If the semigroup is integral, the clusters must be disjoint for every query; on the other hand, if the semigroup is idempotent, clusters can overlap arbitrarily. Thus, upper bounds developed for integral semigroups and lower bounds developed for idempotent semigroups apply to all other semigroups as well.

Some of our lower bounds derive from the following result.

THEOREM 2.1 (see Brönnimann, Chazelle, Pach [10]). *Let P be a uniformly distributed set of points in the d-dimensional unit hypercube $[0,1]^d$. With high probability, any storage scheme of size s that supports halfspace queries over P in time t satisfies the inequality $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$.*

2.2. Unreasonably good bounds for hyperplane queries. Although lower bounds are known for offline hyperplane searching in the semigroup model [18, 31], we are unaware of any previous results for online hyperplane queries. In particular, Chazelle's lower bound $st^d = \Omega(n^d/\log^d n)$ for simplex range searching [14], which holds when the query ranges are slabs bounded by two parallel hyperplanes, does not apply when the ranges are hyperplanes; Chazelle's proof requires a positive lower bound on the width of the slabs.

We easily observe that for any set of points in general position, the smallest possible storage scheme, consisting of n singleton sets, allows hyperplane queries to be answered correctly in constant "time." We can obtain better lower bounds by considering degenerate point sets, as follows.

First consider the two-dimensional case. Let $\mathcal{C}$ be (the set of clusters associated with) an optimal storage scheme of size s that supports line queries for some n-point set P in the plane. The storage scheme $\mathcal{C}$ must contain n singleton sets, one containing each point in P. Without loss of generality, each of the other $s - n$ clusters in $\mathcal{C}$ is a maximal colinear subset of P; that is, each has the form $P \cap \ell$ for some line ℓ. (If a cluster contains three noncolinear points, it can be discarded. If a cluster contains at least two points on a line ℓ, but not every point on ℓ, then adding the missing points decreases the query time for ℓ without changing the number of clusters or the query time for any other line.) Thus, the query time for any line ℓ is 1 if $P \cap \ell \in \mathcal{C}$, and $|P \cap \ell|$ otherwise. It follows that an optimal storage scheme of size s consists of n singleton sets plus the $s - n$ largest maximal colinear subsets of P, and the worst-case query time is the size of the $(s - n + 1)$th largest maximal colinear subset of P.

[6]Despite the complete absence of both semigroups and arithmetic!

Theorem 2.2. *A storage scheme of size $s \geq 2n$ that supports line queries in time t satisfies the inequality $st^3 = \Omega(n^2)$ in the worst case.*

Proof. Let P be a $\sqrt{n} \times \sqrt{n}$ integer lattice of points. Erdős showed that for any integer k, there is a set of k lines L such that the number of point-line incidences between P and L is $\Omega(n^{2/3}k^{2/3})$; see [33] or [48, p. 177]. In particular, we can take L to be the lines containing the k largest colinear subsets of P. Erdős's lower bound implies that the kth largest maximal colinear subset of P contains at least $\Omega(n^{2/3}/k^{1/3})$ points. Thus, the worst-case query time for any storage scheme of size s is $\Omega(n^{2/3}/(s-n+1)^{1/3}) = \Omega(n^{2/3}/s^{1/3})$. □

Surprisingly, this lower bound is optimal for most values of s.

Theorem 2.3. *For any set P of n points in the plane and any integer s with $2n \leq s \leq n^2$, there is a storage scheme of size s that supports line queries for P in time t, where $st^3 = O(n^2)$.*

Proof. Szemerédi and Trotter [53] proved that there are at most $O(n+n^{2/3}k^{2/3}+k)$ incidences between any set of n points and any set of k lines. (See also [19, 52].) Thus, for any k in both $\Omega(\sqrt{n})$ and $O(n^2)$, the kth largest maximal colinear subset of any n-point set has at most $O(n^{2/3}/k^{1/3})$ elements. The theorem follows by setting $k = s - n + 1$. □

In order to derive bounds in higher dimensions, we focus our attention on *restricted* point sets [35], in which any d points lie on a unique hyperplane. The optimal storage scheme for a restricted point set again consists of n singleton sets plus the $s - n$ largest subsets of the form $P \cap h$ for some hyperplane h, and the worst-case query time is the size of the $(s-n+1)$th largest subset of the form $P \cap h$. Of course, lower bounds for restricted point sets also apply to the general case, but the upper bounds do not similarly generalize.

Given a set P of points and a set H of hyperplanes, let $I(P, H)$ denote the number of incidences between P and H, that is, point-hyperplane pairs $(p, h) \in P \times H$ such that $p \in h$. Our higher-dimensional lower bounds, both here and later in the paper, use the following generalization of the Erdős point-line construction.

Lemma 2.4 (see Erickson [31]). *For any integers n and k with $n > \lfloor k^{1/d} \rfloor$, there is a restricted set P of n points and a set H of k hyperplanes in $\mathbb{R}^d$, such that $I(P, H) = \Omega(n^{2/(d+1)}k^{1-2/d(d+1)})$.*

Theorem 2.5. *A storage scheme of size s that supports d-dimensional hyperplane queries in time t satisfies the inequality $st^{d(d+1)/2} = \Omega(n^d)$ in the worst case.*

Using probabilistic counting techniques of Clarkson et al. [23], Guibas, Overmars, and Robert [35] prove that for any restricted set P of n points and any set H of k hyperplanes, $I(P, H) = O(n+n^{d/(2d-1)}k^{(2d-2)/(2d-1)}+k)$. The following upper bound follows immediately from their result.

Theorem 2.6. *For any restricted set P of n points in $\mathbb{R}^d$ and any integer s such that $2n \leq s \leq n^d$, there is a storage scheme of size s that supports hyperplane queries for P in time t, where $st^{2d-1} = O(n^d)$.*

The general case is unfortunately not so straightforward. Optimal storage schemes could have clusters contained in lower-dimensional flats, in which case the query time for a hyperplane h is no longer necessarily either 1 or $|P \cap h|$. If the semigroup is idempotent, every cluster in an optimal storage scheme still has the form $P \cap h$ for some hyperplane h, but this may not be true for all semigroups. We leave further generalization of our upper bounds as an open problem.

Except when s is near n^d, our upper bounds in the semigroup model are significantly better than the best known upper bounds in more realistic models of computation. The most efficient data structure known satisfies the upper bound $st^d = O(n^d)$ [17, 42], and this is believed to be optimal, especially in light of Chazelle's simplex range searching lower bounds. We are not suggesting that this data structure can be significantly improved, but rather that the semigroup model is too powerful to permit tight lower bounds for this range searching problem. This raises the frustrating possibility that closing the existing gaps between upper and lower bounds for other types of ranges, such as halfspaces [10], will be impossible unless more realistic computational models are considered.

In the remainder of this paper, we derive tighter lower bounds by considering a model that more accurately describes the behavior of geometric range searching algorithms.

3. Partition graphs.

3.1. Definitions. A *partition graph* is a directed acyclic (multi-)graph, with one source, called the *root*, and several sinks, called *leaves*. Associated with each nonleaf node v is a set $\mathcal{R}_v$ of *query regions* satisfying three conditions.

1. $\mathcal{R}_v$ contains at most Δ query regions, for some constant $\Delta \geq 2$.
2. Every query region is a connected subset of $\mathbb{R}^d$.
3. The union of the query regions in $\mathcal{R}_v$ is $\mathbb{R}^d$.

We associate an outgoing edge of v with each query region in $\mathcal{R}_v$. Thus, the outdegree of the graph is at most Δ. The indegree can be arbitrarily large. In addition, every internal node v is labeled either a *primal node* or a *dual node*, depending on whether its query regions $\mathcal{R}_v$ are interpreted as a partition of primal or dual space. The query regions associated with primal (resp., dual) nodes are called primal (resp., dual) query regions.

We do not require the query regions to be disjoint. In the general case, we do not require the query regions to be convex, semialgebraic, simply connected, of constant complexity, or even computable in any sense. However, a few of our results only hold for partition graphs with particular types of query regions. If all the query regions in a partition graph are constant-complexity polyhedra, we call it a *polyhedral* partition graph. If all the query regions are constant-complexity semialgebraic sets (also called *Tarski cells*), we call it a *semialgebraic* partition graph.

Given a partition graph, we preprocess a set P of points for hyperplane queries as follows. We preprocess each point $p \in P$ individually by performing a depth-first search of the partition graph, using the query regions to determine which edges to traverse. Whenever we reach a primal node v, we traverse the edges corresponding to the query regions in $\mathcal{R}_v$ that contain p. Whenever we reach a dual node v, we traverse the edges corresponding to the query regions in $\mathcal{R}_v$ that intersect the dual hyperplane p^*. The same point may enter or leave a node along several different edges, but we only test the query regions at a node once for each point. Thus, each point traverses a given edge at most once. For each leaf ℓ, we maintain a *leaf subset* P_ℓ containing the points that reach ℓ. See Figure 1(a).

To answer a hyperplane query, we use almost exactly the same algorithm as to preprocess a point: a depth-first search of the partition graph, using the query regions to determine which edges to traverse. The only difference is that the behavior at the primal and dual nodes is reversed. See Figure 1(b).

Whenever the query algorithm reaches a leaf ℓ, it *examines* the corresponding leaf subset P_ℓ. The output of the query algorithm is computed from the examined subsets, by assuming that the query hyperplane contains each examined subset. For example, the output of an emptiness query is "yes" if and only if all the examined subsets are

```
Preprocess($p$):
at each primal node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ contains $p$:
            traverse the edge corresponding to $R$
at each dual node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ intersects $p^*$:
            traverse the edge corresponding to $R$
at each leaf $\ell$:
    add $p$ to $P_\ell$
```

(a)

```
Query($h$):
at each dual node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ contains $h^*$:
            traverse the edge corresponding to $R$
at each primal node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ intersects $h$:
            traverse the edge corresponding to $R$
at each leaf $\ell$:
    examine $P_\ell$
```

(b)

FIG. 1. *Preprocessing and query algorithms for hyperplane searching.*

empty. (In fact, we can assume that if the algorithm ever examines a nonempty leaf subset, it immediately halts and answers "no.") The output of a counting query is the sum of the sizes of the examined subsets; here, examining a leaf subset means adding its size to a running counter. The output of a reporting query is the union of the examined subsets; here, "examine" simply means "output." More generally, if the points are given weights from a (*not* necessarily faithful) semigroup $(S, +)$, the output is the semigroup sum over all examined subsets P_ℓ of the total weight of the points in P_ℓ. (We will not explicitly consider semigroup queries in the rest of the paper.)

By modifying the preprocessing algorithm slightly, we can also use partition graphs to answer halfspace queries. For any hyperplane h, let h^+ denote its closed upper halfspace and h^- its closed lower halfspace.[7] Recall that the standard duality transformation $(a_1, a_2, \ldots, a_d) \longleftrightarrow x_d + a_d = a_1x_1 + a_2x_2 + \cdots + a_{d-1}x_{d-1}$ preserves incidences and relative orientation between points and hyperplanes: If a point p is above (on, below) a hyperplane h, then the dual point h^* is above (on, below) the dual hyperplane p^*.

To support halfspace queries, we associate one or two subsets of P with every query region, called *internal subsets.* With each primal region $R \in \mathcal{R}_v$, we associate a single internal subset P_R, which contains the points that reach v and lie inside R. With each dual region $R \in \mathcal{R}_v$, we associate two internal subsets P_R^+ and P_R^-, which contain the points that reach v and whose dual hyperplanes lie below and above R, respectively. Our modified preprocessing and halfspace query algorithms are shown in Figure 2. Note that the modified preprocessing algorithm can still be used for hyperplane searching.

For purposes of proving lower bounds, the *size* of a partition graph is the number of edges in the graph, the *query time* for a particular hyperplane is the number of edges traversed by the query algorithm, and the *preprocessing time* is the total number of edge traversals during the preprocessing phase. We ignore, for example, the time required in practice to construct the graph, the complexity of the query regions, the time required to determine which query regions intersect a hyperplane or contain a point, the sizes of the subsets P_R, P_R^+, P_R^-, and P_ℓ, the time required to maintain and test these subsets, and the size of the output (in the case of reporting queries).

[7]We assume throughout the paper that query halfspaces are closed and that no query halfspace is bounded by a vertical hyperplane. Handling open halfspaces involves only trivial modifications to our query algorithm, which have almost no effect on our analysis. Vertical halfspace queries can be handled either through standard perturbation techniques or by using a lower-dimensional data structure.

```
Preprocess($p$):
at each primal node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ contains $p$:
            add $p$ to $P_R$
            traverse the edge corresponding to $R$
at each dual node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ intersects $p^*$:
            traverse the edge corresponding to $R$
        else if $R$ is above $p^*$:
            add $p$ to $P_R^+$
        else if $R$ is below $p^*$:
            add $p$ to $P_R^-$
at each leaf $\ell$:
    add $p$ to $P_\ell$
```

(a)

```
Query($h^\pm$):
at each dual node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $R$ contains $h^*$:
            examine $P_R^\pm$
            traverse the edge corresponding to $R$
at each primal node $v$:
    for each query region $R \in \mathcal{R}_v$:
        if $h$ intersects $R$:
            traverse the edge corresponding to $R$
        else if $h^\pm$ contains $R$:
            examine $P_R$
at each leaf $\ell$:
    examine $P_\ell$
```

(b)

FIG. 2. *Modified preprocessing algorithm and query algorithm for halfspace searching.*

We emphasize that since we never charge for the construction of the partition graph itself, the graph and its query regions can depend arbitrarily on the input point set P and on the types of queries we expect to receive. Our preprocessing algorithm has "time" to construct the *optimal* partition graph for any given input, and even very similar inputs may result in radically different partition graphs.

We will also consider offline range searching problems, where we are given both a set P of points and a set H of ranges (either hyperplanes or halfspaces), and are asked, for example, whether any range contains a point. A *partitioning algorithm* constructs a partition graph, which can depend arbitrarily on the input, preprocesses each point in P, and performs a query for each range in H. The running time of the partitioning algorithm is the sum of the preprocessing and query times. In the original definition [31], the preprocessing and queries were performed concurrently, but this has no effect on the overall running time.[8] Again, since we ignore the time required in practice to construct the partition graph, partitioning algorithms have the full power of nondeterminism.

3.2. Basic properties. Partition graphs have several properties that are very useful in proving lower bounds.

LEMMA 3.1. *In any partition graph, if a point lies in a query range, it also lies in at least one of the subsets P_R, P_R^+, P_R^-, or P_ℓ examined during a query.*

Proof. First suppose some point p lies on some hyperplane h. Clearly, any primal query region that contains p also intersects h, and any dual query region that contains h^* also intersects p^*. Thus, there is at least one path from the root to a leaf ℓ that is traversed both while preprocessing p (so $p \in P_\ell$) and while querying h (so P_ℓ is examined).

Now suppose some point p lies in some upper halfspace h^+. (The argument for lower halfspaces is symmetric.) Let v be a node farthest from the root that is reached both while preprocessing p and while querying h^+. If v is a leaf, we are done. If v is a primal node, then some query region $R \in \mathcal{R}_v$ contains p but does not intersect h.

[8]However, if we perform all the searches concurrently using the *streaming* technique of Edelsbrunner and Overmars [28], we only need to maintain a single root-to-leaf path in the graph at any time. Thus, the space used by an offline partitioning algorithm is more reasonably modeled by the *depth* of its partition graph. We will not pursue this idea further in this paper.

Since $p \in R$, the subset P_R contains p, and since $p \in h^+$, R must lie above h, so P_R is examined. Finally, if v is a dual node, some query region $R \in \mathcal{R}_v$ contains the dual point h^* but does not intersect the dual hyperplane p^*. Since $h^* \in R$, the subset P_R^+ is examined. Since $p \in h^+$, the dual point h^* and thus the query region R lie above the dual hyperplane p^*, so $p \in P_R^+$. ∎

LEMMA 3.2. *In any partition graph, if an internal subset P_R, P_R^+, or P_R^- is examined during a halfspace query, then the query halfspace contains every point in that set.*

Proof. Suppose we are performing a query for the upper halfspace h^+. (Again, the argument for lower halfspaces is symmetric.) If the primal subset P_R is examined, then $P_R \subset R \subseteq h^+$. If the dual subset P_R^+ is examined, then every point $p \in P_R^+$ is above the hyperplane h, since the dual point $h^* \in R$ is above each dual hyperplane p^*. ∎

Lemma 3.1 implies that partition graphs are "conservative"—the output of a reporting query contains every point in the query range, the output of a counting query is never smaller than the actual number of points in the query range, and an emptiness query never reports that a nonempty query range is empty. Lemma 3.2 further implies that the output of a query can be incorrect only if the query algorithm examines a leaf subset that contains a point outside the query range.

The following lemma follows immediately from a close examination of the query algorithm.

LEMMA 3.3. *A counting query is correct if and only if the points in the query range are the disjoint union of the subsets examined by the query algorithm. A reporting query is correct if and only if the points in the query range are the (not necessarily disjoint) union of the subsets examined by the query algorithm.*

We say that a partition graph *supports* a particular class of online range queries for a given set of points if, after the points are preprocessed, any query in the class is answered correctly. Even though we have a single preprocessing algorithm, a single partition graph need not support all query types. However, in several cases, support for one type of query automatically implies support for another type of query, with the same (or possibly smaller) worst-case query time. These implications are summarized in the following lemma.

LEMMA 3.4. *The following hold for any partition graph.*

(a) *If a counting query is answered correctly, then a reporting query for the same range is also answered correctly.*

(b) *If a reporting query is answered correctly, then an emptiness query for the same range is also answered correctly.*

(c) *For any hyperplane h, if counting (resp. reporting) queries for the halfspaces h^+ and h^- are answered correctly, then a counting (resp., reporting) query for h is also answered correctly.*

(d) *For any hyperplane h, if a reporting query for h is answered correctly, then reporting queries for the halfspaces h^+ and h^- are also answered correctly.*

Proof. Parts (a) and (b) follow immediately from Lemma 3.3.

Fix a hyperplane h, and let $P^+ = P \cap h^+$, $P^- = P \cap h^-$, and $P^\circ = P \cap h$. Suppose reporting queries for the halfspaces h^+ and h^- are answered correctly. The reporting query algorithms for h^+, h^-, and h traverse precisely the same set of edges and reach precisely the same set of nodes. Let $\mathcal{L}$ be the set of leaves reached by any of these three queries, and let $P_{\mathcal{L}} = \bigcup_{\ell \in \mathcal{L}} P_\ell$. By Lemma 3.3, both P^+ and P^- are reported as the union of several internal subsets and $P_{\mathcal{L}}$. It follows that $P_{\mathcal{L}} \subseteq P^+ \cap P^- = P^\circ$.

Lemma 3.1 implies that every point in P° lies in some leaf subset P_ℓ where $\ell \in \mathcal{L}$, so $P^\circ \subseteq P_{\mathcal{L}}$. Thus, a reporting query for h is also answered correctly. The argument for counting queries is identical, except that the examined subsets are disjoint. This proves part (c).

To prove part (d), suppose a reporting query for h is answered correctly. A reporting query for the halfspace h^+ traverses exactly the same edges and visits exactly the same nodes as the reporting query for h. In particular, the leaf subsets P_ℓ examined by the halfspace query algorithm contain only points on the hyperplane h. Lemma 3.2 implies that every internal subset examined by the reporting query algorithm lies in h^+, and Lemma 3.1 implies that every point in P^+ lies in some examined subset. It follows that precisely the points in P^+ are reported. □

3.3. "Trivial" lower bounds. We conclude this section by proving "trivial" lower bounds on the size, preprocessing time, and query time of any partition graph, for points in any dimension.

THEOREM 3.5. *Any partition graph that supports hyperplane emptiness queries has size $\Omega(n)$, preprocessing time $\Omega(n \log n)$, and worst-case query time $\Omega(\log n)$.*

Proof. Let P be an arbitrary set of n points in $\mathbb{R}^d$. Without loss of generality, all the points in P have distinct x_d coordinates; otherwise, rotate the coordinate system slightly. Let H be a set of n hyperplanes normal to the x_d-axis, with each hyperplane just above (farther in the x_d-direction than) one of the points in P. Thus, for all $1 \le i \le n$, there is a hyperplane in H with i points below it and $n-i$ points above it. Any partition graph that correctly answers hyperplanes queries for P must at least detect that every hyperplane in H is empty.

For each point in P, call the hyperplane in H just above it its *partner*. We say that a point is *active* at a node v of the partition graph if both the point and its partner reach v. We say that a node v *deactivates* a point p if both p and its partner h reach v but no edge out of v is traversed by both p and h. Every point in P must be deactivated by some node in the partition graph, since otherwise some active point p and its partner h would reach a common leaf, so a query for h would be answered incorrectly.

Any primal query region R contains at most one active point whose partner does not intersect R. Similarly, for any dual query region R, there is at most one active point whose dual hyperplane misses R and whose partner's dual point lies in R. Thus, any node deactivates at most Δ points. Moreover, since every point in P must be deactivated, the partition graph must have at least n query regions and thus at least n edges.

The *level* of a node is its distance from the root. There are at most Δ^k nodes at level k. At least $n - \sum_{i=0}^{k-1} \Delta^{k+1} \ge n - \Delta^{k+2}$ points are active at some node at level k. In particular, at least $n(1-1/\Delta)$ points are active at level $\lfloor \log_\Delta n - 3 \rfloor$. It follows that at least $n(1-1/\Delta)$ points in P each traverse at least $\lfloor \log_\Delta n - 2 \rfloor$ edges, so the total preprocessing time is at least

$$n(1-1/\Delta)\lfloor \log_\Delta n - 2 \rfloor = \Omega(n \log n).$$

Similarly, at least $n(1-1/\Delta)$ hyperplanes in H each traverse at least $\lfloor \log_\Delta n - 2 \rfloor$ edges, so the worst-case query time is $\Omega(\log n)$. □

Lemma 3.4 implies that the same lower bounds also apply to counting and reporting queries, both for hyperplanes and for halfspaces. Theorem 3.5 also implies that any partitioning algorithm, given n points and k hyperplanes (or halfspaces, if we are not performing emptiness queries), requires at least $\Omega(n \log k + k \log n)$ time in the worst case; this was previously proved in [31], using essentially the same argument. We will prove similar lower bounds for halfspace emptiness queries in section 8.

4. Space-time tradeoffs. We now present our space-time tradeoff lower bounds for hyperplane emptiness and related queries in the partition graph model. All of these bounds are derived from results in the semigroup arithmetic model, which we described in section 2, using the following theorem.

THEOREM 4.1. *Let P be a set of points. Given a partition graph of size s that supports hyperplane (resp., halfspace) counting or reporting queries for P in time t, we can construct a storage scheme of size $O(s)$ that supports hyperplane (resp., halfspace) queries for P, over any idempotent faithful semigroup, in time $O(t)$.*

Proof. For each query region R and leaf ℓ, define the subsets P_R, P_R^-, P_R^+, and P_ℓ by the preprocessing algorithm in Figure 2. We claim that these subsets of P form the clusters of the required storage scheme. There are at most $3s$ of these clusters: at most two for each of the s query regions, plus one for each of the $\leq s$ leaves. To prove the theorem, it suffices to show that the points in any query range can be expressed as the union of $O(t)$ clusters.

Suppose the partition graph supports hyperplane reporting queries. By Lemma 3.3, the points on any hyperplane h are reported as the union of several leaf subsets P_ℓ. Since the query algorithm reaches at most t leaves, the set $P \cap h$ is the union of at most t clusters.

Similarly, if the partition graph supports halfspace reporting queries, then the points in any halfspace $h^\pm$ are reported as the union of at most t subsets P_R, at most Δt subsets $P_R^\pm$, and at most t subsets P_ℓ. Thus, the set $P \cap h^\pm$ is the union of at most $(2+\Delta)t = O(t)$ clusters.

The argument for counting queries is identical, except that the $O(t)$ clusters used to answer any query are disjoint. (In fact, the resulting storage scheme works for *any* faithful semigroup.) □

This theorem implies that lower bounds for range queries in the semigroup arithmetic model are also lower bounds for the corresponding counting and reporting queries in the partition graph model. The following results now immediately follow from Theorems 2.1 and 2.5.

COROLLARY 4.2. *Let P be a uniformly generated set of n points in $[0,1]^d$. With high probability, any partition graph of size s that supports halfspace counting or reporting queries for P in time t satisfies the inequality $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$.*

COROLLARY 4.3. *Any partition graph of size s that supports d-dimensional hyperplane counting or reporting queries in time t satisfies the inequality $st^{d(d+1)/2} = \Omega(n^d)$ in the worst case.*

One way to determine if a hyperplane is empty is by counting or reporting the points in its two halfspaces—the hyperplane is empty if any only if every point in the original point set is counted or reported exactly once. Thus, any halfspace counting or reporting data structure also supports hyperplane emptiness queries. The following result implies that the reverse is *almost* true in our model of computation.

THEOREM 4.4. *Let P be a set of points. Given a partition graph of size s that supports hyperplane emptiness queries for P in time t, we can construct a storage scheme of size $O(s)$ that supports halfspace queries for P, over any idempotent faithful semigroup, in time $O(t)$.*

Proof. Suppose a partition graph G supports hyperplane emptiness queries for the set P in time t. Clearly, G also supports reporting queries for any *empty* hyperplane in time t, since all the examined subsets are empty. Then by Lemma 3.4 (d), G correctly answers any halfspace reporting query in time at most t, provided the boundary of the query halfspace is empty. However, for any halfspace, there is another halfspace with empty boundary that contains precisely the same points. It follows that every set of the form $P \cap h^+$ or $P \cap h^-$ can be expressed as the union of at most $(2+\Delta)t = O(t)$ internal subsets. $\square$

The following lower bound now follows immediately from Theorem 2.1.

COROLLARY 4.5. *Let P be a uniformly generated set of n points in $[0,1]^d$. With high probability, any partition graph of size s that supports hyperplane emptiness queries for P in time t satisfies the inequality $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$.*

Lemma 3.4 implies that the same lower bound applies to hyperplane counting or reporting queries. This improves the lower bound in Corollary 4.3 whenever $s = O(n^{d-1})$ or $t = \Omega(n^{2/d(d+1)})$. Similarly, this lower bound also applies to halfspace counting or reporting queries, giving us a rather roundabout proof of Corollary 4.2. However, none of these results applies immediately to halfspace emptiness queries; we will derive lower bounds for these queries in section 8.

5. Polyhedral covers.

5.1. Definitions. In order to derive our improved offline lower bounds and preprocessing-query time tradeoffs, we first need to define a combinatorial object called a *polyhedral cover*. The formal definition is fairly technical, but intuitively, one can think of a polyhedral cover of a set P of points and a set H of hyperplanes as a collection of constant-complexity convex polytopes such that for every point $p \in P$ and hyperplane $h \in H$, some polytope in the collection contains p and does not intersect h. A polyhedral cover of P and H provides a compact representation of the relative orientation of every point in P and every hyperplane in H.

Our combinatorial bounds rely heavily on certain properties of convex polytopes and polyhedra. Many of these properties are more easily proved, and have fewer special cases, if we state and prove them in projective space rather than affine Euclidean space. In particular, developing these properties in projective space allows us to more easily deal with unbounded polyhedra, degenerate polyhedra, and duality transformations. Everything we define in this subsection can be formalized algebraically in the language of polyhedral cones and linear subspaces one dimension higher; we will give a less formal, purely geometric treatment. For more technical details, we refer the reader to the first two chapters of Ziegler's lecture notes [58] or to the survey by Henk, Richter-Gebert, and Ziegler [36].

The d-dimensional real projective space $\mathbb{RP}^d$ can be defined as the set of lines through the origin in the $(d+1)$-dimensional real vector space $\mathbb{R}^{d+1}$. Every k-dimensional linear subspace of $\mathbb{R}^{d+1}$ induces a $(k-1)$-dimensional *flat* f in $\mathbb{RP}^d$, and its orthogonal complement induces the $(d-k-1)$-dimensional *dual flat* f^*. Hyperplanes are $(d-1)$-dimensional flats, points are 0-dimensional flats, and the empty set is the unique (-1)-flat.

Any finite set H of (at least two) hyperplanes in $\mathbb{RP}^d$ defines a regular cell complex called an *arrangement*, each of whose cells is the closure of a maximal connected subset of $\mathbb{RP}^d$ contained in the intersection of a fixed subset of H and not intersecting any other hyperplane in H. The largest cells in the arrangement are the closures of the connected components of $\mathbb{RP}^d \setminus \bigcup H$; the intersection of any pair of cells is another cell of lower dimension.

A *projective polyhedron* is a single cell, not necessarily of full dimension, in an arrangement of hyperplanes in $\mathbb{RP}^d$. A *projective polytope* is a simply-connected polyhedron, or equivalently, a polyhedron that is disjoint from some hyperplane (not necessarily in the polyhedron's defining arrangement). Every projective polyhedron is (the closure of) the image of a convex polyhedron under some projective transformation, and every projective polytope is the projective image of a convex polytope. Every flat is also a projective polyhedron.

The *(projective) span* of a polyhedron $\Pi \subseteq \mathbb{RP}^d$, denoted span$(\Pi)$, is the flat of minimal dimension that contains it. The *dimension* of a polyhedron is the dimension of its span. The *relative interior* (resp., *relative boundary*) of a polyhedron is its interior (resp., boundary) in the subspace topology of its span. A hyperplane *supports* a polyhedron if it intersects the polyhedron but not its relative interior. In particular, a flat has no supporting hyperplanes.

A *proper face* of a polyhedron is the intersection of the polyhedron and one or more supporting hyperplanes. Every proper face of a polyhedron is a lower-dimensional polyhedron. A *face* of a polyhedron is either a proper face or the entire polyhedron. We write $|\Pi|$ to denote the number of faces of a polyhedron Π, and $\phi \leq \Pi$ to denote that ϕ is a face of Π. The faces of a polyhedron form a graded lattice under inclusion. Every projective polyhedron has a face lattice isomorphic to that of a convex polytope, possibly of lower dimension.

The *dual* of a polyhedron Π, denoted Π^*, is the set of points whose dual hyperplanes intersect Π in one of its faces:

$$\Pi^* \triangleq \{p \mid (p^* \cap \Pi) \leq \Pi\}.$$

In other words, $p \in \Pi^*$ if and only if p^* either contains Π, supports Π, or completely misses Π. This definition generalizes both the polar of a convex polytope containing the origin and the projective dual of a flat. We easily verify that Π^* is a projective polyhedron whose face lattice is the inverse of the face lattice of Π. In particular, Π and Π^* have the same number of faces. See [58, pp. 59–64] and [51, pp. 143–150] for similar definitions.

We say that a polyhedron Π *separates* a set P of points and a set H of hyperplanes if Π contains P and the dual polyhedron Π^* contains the dual points H^*, or equivalently, if any hyperplane in H either contains Π or is disjoint from its relative interior. In particular, if Π is of full dimension, then the hyperplanes in H avoid the interior of Π. Both P and H may intersect the relative boundary of Π, and points in P may lie on hyperplanes in H. See Figure 3. Note that Π separates P and H if and only if Π^* separates H^* and P^*. We say that P and H are *r-separable*, denoted $P \bowtie_r H$, if there is a projective polyhedron with at most r faces that separates them. We write $P \not\bowtie_r H$ if P and H are not r-separable.

Finally, an *r-polyhedral cover* of a set P of points and a set H of hyperplanes is an indexed set of subset pairs $\{(P_i, H_i)\}$ with the following properties.

- $P_i \subseteq P$ and $H_i \subseteq H$ for all i.
- If $p \in P$ and $h \in H$, then $p \in P_i$ and $h \in H_i$ for some i.
- $P_i \bowtie_r H_i$ for all i.

We emphasize that the subsets P_i are not necessarily disjoint, nor are the subsets H_i. We refer to each subset pair (P_i, H_i) in a polyhedral cover as a *r-polyhedral minor*. The *size* of a cover is the sum of the sizes of the subsets P_i and H_i.

Let $\pi_r(P, H)$ denote the size of the smallest r-polyhedral cover of P and H. Let $\pi^\circ_{d,r}(n, k)$ denote the maximum of $\pi_r(P, H)$ over all sets P of n points and H of k hyperplanes in $\mathbb{RP}^d$, such that no point lies on any hyperplane. In all our terminology and notation, whenever the parameter r is omitted, we take it to be a fixed constant. In the remainder of this section, we derive asymptotic lower bounds for $\pi^\circ_d(n, k)$.

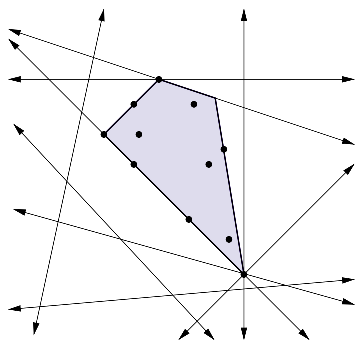


FIG. 3. *A polygon separating a set of points and a set of lines in* $\mathbb{RP}^2$.

5.2. Topological properties. The lower bound proofs in [31] relied on the following trivial observation: If we perturb a set of points and hyperplanes just enough to remove any point-hyperplane incidences, and every point is above every hyperplane in the perturbed set, then no point was below a hyperplane in the original set. In this section, we establish the corresponding, but no longer trivial, property of separable sets and polyhedral covers. Informally, if a set of points and hyperplanes is not separable, then arbitrarily small perturbations cannot make it separable. Similarly, arbitrarily small perturbations of a set of points and hyperplanes cannot decrease its minimum polyhedral cover size.

We start by proving a more obvious property of convex polytopes, namely, that infinitesimally perturbing a set of points can only increase the complexity of its convex hull.

LEMMA 5.1. *For any integers n and r, the set of n-point configurations in $\mathbb{R}^d$ whose convex hulls have at most r faces is topologically closed.*

Proof. Let $A = \{a_1, a_2, \dots, a_n\}$ and $B = \{b_1, b_2, \dots, b_n\}$ be two n-point configurations (i.e., indexed sets of n points) in $\mathbb{R}^d$. We say that A *is simpler than* B, written $A \sqsubseteq B$, if for any subset of B contained in a facet of $\operatorname{conv}(B)$, the corresponding subset of A is contained in a facet of $\operatorname{conv}(A)$.[9] Equivalently, $A \sqsubseteq B$ if and only if for any subset of $d+1$ points in B, d of whose vertices lie on a facet of $\operatorname{conv}(B)$, the corresponding simplex in A either has the same orientation or is degenerate. Simpler point sets have less complex convex hulls: if $A \sqsubseteq B$, then $|\operatorname{conv}(A)| \le |\operatorname{conv}(B)|$. If both $A \sqsubseteq B$ and $B \sqsubseteq A$, then the convex hulls of A and B are combinatorially equivalent.

If B is fixed, then the relation $A \sqsubseteq B$ can be encoded as the conjunction of a finite number of algebraic inequalities of the form

$$\begin{vmatrix} 1 & a_{i_0 1} & a_{i_0 2} & \cdots & a_{i_0 d} \\ 1 & a_{i_1 1} & a_{i_1 2} & \cdots & a_{i_1 d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{i_d 1} & a_{i_d 2} & \cdots & a_{i_d d} \end{vmatrix} \Diamond 0,$$

where $\Diamond$ is either $\ge$, $=$, or $\le$, and a_{ij} denotes the jth coordinate of $a_i \in A$. In every such inequality, the corresponding points $b_{i_1}, b_{i_2}, \dots, b_{i_d} \in B$ lie on a single facet of

[9]Every set of points is simpler than itself. It would be more correct, but also more awkward, to say "A is at least as simple as B."

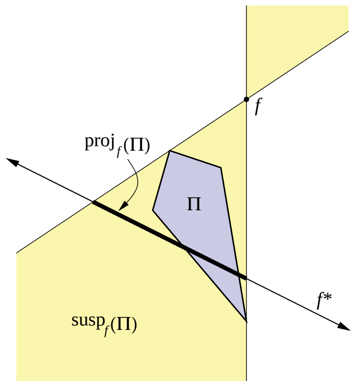


FIG. 4. *Suspension (double wedge) and projection (line segment) of a polygon by a point in* $\mathbb{RP}^2$.

$\mathrm{conv}(B)$. For every d-tuple of points in B contained in a facet of $\mathrm{conv}(B)$, there are $n-d$ such inequalities, one for every other point in B. (If we replaced the loose inequalities $\leq, \geq$ with strict inequalities $<, >$, the resulting expression would encode the combinatorial equivalence of $\mathrm{conv}(A)$ and $\mathrm{conv}(B)$.) Thus, for any *fixed* n-point set B, the set $\{A \in (\mathbb{R}^d)^n \mid A \sqsubseteq B\}$ is the intersection of a finite number of closed algebraic halfspaces, and is thus a closed semialgebraic set.

There are only finitely many equivalence classes of convex polytopes with a given number of faces or vertices [34]. Thus, there is a *finite* set $\mathcal{B} = \{B^1, B^2, \ldots\}$ of n-point configurations, one of each possible combinatorial type, such that if $\mathrm{conv}(A)$ has at most r faces, then $A \sqsubseteq B^i$ for some configuration $B^i \in \mathcal{B}$. It follows that the set of n-point configurations whose convex hulls have at most r faces is the union of a finite number of closed sets and is therefore closed. $\square$

Before continuing, we need to introduce one more important concept. For any subset $X \subseteq \mathbb{RP}^d$ and any flat f, the *suspension of* X *by* f, denoted $\mathrm{susp}_f(X)$, is formed by replacing each point in X by the smallest flat containing that point and f:

$$\mathrm{susp}_f(X) \triangleq \bigcup_{p \in X} \mathrm{span}(p \cup f).$$

The suspension of a subset of projective space roughly corresponds to an infinite cylinder over a subset of an affine space, at least when the suspending flat f is on the hyperplane at infinity. The *projection of* X *by* f, denoted $\mathrm{proj}_f(X)$, is the intersection of the suspension and the dual flat f^*:

$$\mathrm{proj}_f(X) \triangleq \mathrm{susp}_f(X) \cap f^*.$$

In particular, $\mathrm{susp}_f(X)$ is the set of all points in $\mathbb{RP}^d$ whose projection by f is in $\mathrm{proj}_f(X)$. The projection of a subset of projective space corresponds to the orthogonal projection or shadow of a subset of affine space onto a lower-dimensional flat. See Figure 4. For any polyhedron Π and any flat f, $\mathrm{susp}_f(\Pi)$ and $\mathrm{proj}_f(\Pi)$ are also polyhedra. These two polyhedra have the same number of faces (in fact, they have isomorphic face lattices), although in general they have fewer faces than Π. In particular, $\mathrm{susp}_f(f) = f$ and $\mathrm{proj}_f(f) = \varnothing$.

LEMMA 5.2. *Let* H *be a set of hyperplanes in* $\mathbb{RP}^d$. *For any integers* r *and* n, *the set* $\mathrm{Sep}_r(H, n)$ *of* n*-point configurations* $P \in (\mathbb{RP}^d)^n$ *such that* P *and* H *are* r*-separable is topologically closed.*

Proof. There are two cases to consider: Either the hyperplanes in H do not have a common intersection, or they intersect in a common flat. The proof of the second case relies on the first.

Case 1 ($\bigcap H = \varnothing$). Any polyhedron that separates P and H must be completely contained in a closed d-dimensional cell of the arrangement of H. Since there are only finitely many such cells, it suffices to show, for each closed d-cell $\mathcal{C}$, that the set $\operatorname{Sep}_r(H,n) \cap \mathcal{C}^n$ of n-point configurations contained in $\mathcal{C}$ and r-separable from H is topologically closed. We will actually show that $\operatorname{Sep}_r(H,n) \cap \mathcal{C}^n$ is a compact semialgebraic set.

Fix a cell $\mathcal{C}$. Since the hyperplanes in H do not have a common intersection, both $\mathcal{C}$ and any polyhedra it contains must be polytopes. By choosing an appropriate hyperplane "at infinity" that misses $\mathcal{C}$, we can treat $\mathcal{C}$ and any polytopes it contains as *convex* polytopes in $\mathbb{R}^d$. Any separating polytope with r or fewer faces is the (closed) convex hull of some set of r points, all contained in $\mathcal{C}$. Thus, we can write

$$\operatorname{Sep}_r(H,n) \cap \mathcal{C}^n = \left\{P \in \mathcal{C}^n \;\middle|\; \exists\, A \in \mathcal{C}^r \colon P \subset \operatorname{conv}(A) \,\wedge\, |\operatorname{conv}(A)| \le r\right\}.$$

Lemma 5.1 implies that the set

$$\left\{(P,A) \in \mathcal{C}^{n+r} \;\middle|\; |\operatorname{conv}(A)| \le r\right\} = \left\{A \in \mathcal{C}^r \;\middle|\; |\operatorname{conv}(A)| \le r\right\} \times \mathcal{C}^n$$

is compact (closed and bounded). The set

$$\left\{(P,A) \in \mathcal{C}^{n+r} \;\middle|\; P \subset \operatorname{conv}(A)\right\}$$

can be rewritten as

$$\left\{(P,A) \in \mathcal{C}^{n+r} \;\middle|\; \bigwedge_{i=1}^{n} \left(\exists \lambda_i \in [0,1]^r \colon \sum_{j=1}^{r} \lambda_{ij} a_j = p_i \;\wedge\; \sum_{j=1}^{r} \lambda_{ij} = 1\right)\right\}.$$

(Here p_i is the ith point in P, a_j is the jth point in A, λ_i is the vector of barycentric coordinates for the point p_i, and λ_{ij} is its jth component.) This set is an orthogonal projection of the compact semialgebraic set

$$\left\{(P,A,\Lambda) \in \mathcal{C}^{n+r} \times [0,1]^{r\times n} \;\middle|\; \bigwedge_{i=1}^{n} \left(\sum_{j=1}^{r} \lambda_{ij} a_j = p_i \;\wedge\; \sum_{j=1}^{r} \lambda_{ij} = 1\right)\right\}$$

and is thus also compact. (Here Λ is the $n \times r$ matrix of barycentric coordinates λ_{ij}.) It follows that $\operatorname{Sep}_r(H,n) \cap \mathcal{C}^n$ is an orthogonal projection of the intersection of two compact sets and so must be compact.

Case 2 ($\bigcap H \neq \varnothing$). The previous argument does not work in this case, because the cells in the arrangement of H are not simply connected and thus are not polytopes. However, they are combinatorially equivalent to polytopes of lower dimension. To prove that $\operatorname{Sep}_r(H,n)$ is closed, we essentially project everything down to a lower-dimensional subspace in which the hyperplanes do not have a common intersection and apply our earlier argument.

We will actually prove that the complement of $\operatorname{Sep}_r(H,n)$ is open. Let P be an arbitrary set of n points in $\mathbb{RP}^d$ such that P and H are *not* r-separable. To prove the lemma, it suffices to show that there is an open set $\mathcal{U} \subseteq (\mathbb{RP}^d)^n$ with $P \in \mathcal{U}$, such that $Q \not\bowtie H$ for all $Q \in \mathcal{U}$.

Let $f = \bigcap H$, and let f^* be the flat dual to f. Without loss of generality, suppose that $P \setminus f = \{p_1, p_2, \ldots, p_m\}$ and $P \cap f = \{p_{m+1}, \ldots, p_n\}$ for some integer m. (Either of these two subsets may be empty.) The lower-dimensional hyperplanes $H \cap f^*$ do not have a common intersection, so by our earlier argument, $\mathrm{Sep}_r(H \cap f^*, m)$ is closed.

If some polyhedron Π separated P and H, then its projection $\mathrm{proj}_f(\Pi) \subseteq f^*$ would separate the projected points $\mathrm{proj}_f(P)$ and the lower-dimensional hyperplanes $H \cap f^*$. Conversely, if any polyhedron $\Pi \subseteq f^*$ separated $\mathrm{proj}_f(P)$ and $H \cap f^*$, then its suspension $\mathrm{susp}_f(\Pi)$ would separate P and H. Thus, $P \bowtie H$ if and only if $\mathrm{proj}_f(P) \bowtie (H \cap f^*)$. Moreover, since $\mathrm{proj}_f(P) = \mathrm{proj}_f(P \setminus f)$, it follows that $P \bowtie H$ if and only if $(P \setminus f) \bowtie H$. (Note that this argument is not valid for arbitrary flats f, but only for $f = \bigcap H$.)

Since by assumption $P \not\bowtie H$, it follows that $\mathrm{proj}_f(P) \not\bowtie (H \cap f^*)$. Thus, by Case 1, there is an open set $\mathcal{U}' \subseteq (f^*)^m$, with $\mathrm{proj}_f(P) \in \mathcal{U}'$, such that $S \not\bowtie H$ for any $S \in \mathcal{U}'$. Let $\mathcal{U}'' \subseteq (\mathbb{RP}^d \setminus f)^m$ be the set of m-point configurations R with $\mathrm{proj}_f(R) \in \mathcal{U}'$. Since $\mathbb{RP}^d \setminus f \cong f^* \times \mathbb{R}^{\dim f}$, we have $\mathcal{U}'' \cong \mathcal{U}' \times (\mathbb{R}^{\dim f})^m$, so $\mathcal{U}''$ is an open neighborhood of $P \setminus f$. For any $R \in \mathcal{U}''$, since $\mathrm{proj}_f(R) \not\bowtie (H \cap f^*)$, we have $R \not\bowtie H$. Finally, let $\mathcal{U} = \mathcal{U}'' \times (\mathbb{RP}^d)^{n-m}$; clearly, $\mathcal{U}$ is an open neighborhood of P. For every configuration $Q \in \mathcal{U}$, there is a subset $R \subseteq Q$ such that $R \not\bowtie H$, so $Q \not\bowtie H$. □

LEMMA 5.3. *Let P be a set of n points and H a set of k hyperplanes in $\mathbb{RP}^d$. For all $Q \in (\mathbb{RP}^d)^n$ in an open neighborhood of P, $\pi_r(Q, H) \geq \pi_r(P, H)$.*

Proof. For any indexed set of objects (points or hyperplanes) $X = \{x_1, x_2, \ldots\}$ and any set of indices $I \subseteq \{1, 2, \ldots, |X|\}$, let X_I denote the subset $\{x_i \mid i \in I\}$.

Fix two sets of indices $I \subseteq \{1, 2, \ldots, n\}$ and $J \subseteq \{1, 2, \ldots, k\}$, and consider the corresponding subsets $P_I \subseteq P$ and $H_J \subseteq H$. If $P_I \bowtie H_J$, define $\mathcal{U}_{I,J} = (\mathbb{RP}^d)^n$. Otherwise, define $\mathcal{U}_{I,J} \subseteq (\mathbb{RP}^d)^n$ to be an open neighborhood of P such that $Q_I \not\bowtie H_J$ for all $Q \in \mathcal{U}_{I,J}$. Lemma 5.2 implies the existence of such an open neighborhood.

Let $\mathcal{U}$ be the intersection of the $2^n 2^k$ open sets $\mathcal{U}_{I,J}$. Since each $U_{I,J}$ is an open neighborhood of P, $\mathcal{U}$ is also an open neighborhood of P. For all $Q \in \mathcal{U}$ and for all index sets I and J, if $Q_I \bowtie H_J$, then $P_I \bowtie H_J$. In other words, every r-polyhedral minor of Q and H corresponds to a r-polyhedral minor of P and H. Thus, for any r-polyhedral cover of Q and H, there is a corresponding r-polyhedral cover of P and H with exactly the same size. □

5.3. Lower bounds.

We are finally in a position to prove our combinatorial lower bounds. As in section 2, let $I(P, H)$ denote the number of point-hyperplane incidences between P and H.

LEMMA 5.4. *Let P be a set of n points and let H be a set of k hyperplanes, such that no subset of a hyperplanes contains b points in its intersection. If P and H are r-separable, then $I(P, H) \leq r(a + b)(n + k)$.*

Proof. Let Π be a polyhedron with r faces that separates P and H. For any point $p \in P$ and hyperplane $h \in H$ such that p lies on h, there is some face ϕ of Π that contains p and is contained in h. For each face $\phi \leq \Pi$, let P_ϕ denote the points in P that are contained in ϕ, and let H_ϕ denote the hyperplanes in H that contain ϕ. Every point in P_ϕ lies on every hyperplane in H_ϕ.

Since no set of a hyperplanes can all contain the same b points, it follows that for all ϕ, either $|P_\phi| < b$ or $|H_\phi| < a$. Thus, we can bound $I(P, H)$ as follows.

$$I(P, H) \leq \sum_{\phi \leq \Pi} I(P_\phi, H_\phi) = \sum_{\phi \leq \Pi} \bigl(|P_\phi| \cdot |H_\phi|\bigr) \leq (a + b) \sum_{\phi \leq \Pi} \bigl(|P_\phi| + |H_\phi|\bigr).$$

Since Π has r faces, the last sum counts each point in P and each hyperplane in H at most r times. □

THEOREM 5.5. $\pi_d^\circ(n,k) = \Omega\left(n^{1-2/d(d+1)}k^{2/(d+1)} + n^{2/(d+1)}k^{1-2/d(d+1)}\right)$.

Proof. Let P be a restricted set of n points and H a set of k hyperplanes in $\mathbb{R}^d$, such that $I(P,H) = \Omega(n^{2/(d+1)}k^{1-2/d(d+1)})$ as described by Lemma 2.4. Consider any subsets $P_i \subseteq P$ and $H_i \subseteq H$ such that $P_i \bowtie_r H_i$. Applying Lemma 5.4 with $s = 2$ and $t = d$, we have $I(P_i, H_i) \le (2+d)r(|P_i| + |H_i|)$. It follows that any collection of r-polyhedral minors that includes every incidence between P and H must have size at least $I(P,H)/(2+d)r$. Thus, $\pi_r(P,H) = \Omega(n^{2/(d+1)}k^{1-2/d(d+1)})$. Finally, Lemma 5.3 implies that we can perturb P slightly, removing all the incidences, without decreasing the polyhedral cover size.

The symmetric lower bound $\Omega(n^{1-2/d(d+1)}k^{2/(d+1)})$ follows by considering the dual points H^* and the dual hyperplanes P^*. □

When $d \le 3$, this result follows from earlier bounds on the complexity of monochromatic covers derived in [31]. (In a monochromatic minor, either every point lies above every hyperplane, or every point lies below every hyperplane.)

Our d-dimensional lower bound only improves our $(d-1)$-dimensional lower bound when $k = O(n^{2/(d-1)})$ or $k = \Omega(n^{(d-1)/2})$. We can combine the lower bounds from all dimensions $1 \le i \le d$ into a single expression, as in [31, 30]:

$$\pi_d^\circ(n,k) = \Omega\left(\sum_{i=0}^{d}\left(n^{1-2/i(i+1)}k^{2/(i+1)} + n^{2/(i+1)}k^{1-2/i(i+1)}\right)\right).$$

If the relative growth rates of n and k are fixed, this entire sum reduces to a single term. In particular, when $k = n$, the best lower bound we can prove is $\pi_d^\circ(n,n) \ge \pi_2^\circ(n,n) = \Omega(n^{4/3})$, the proof of which requires only the original point-line configuration of Erdős. (See Theorem 2.2.)

We conjecture that $\pi_d^\circ(n,n) = \Theta(n^{2d/(d+1)})$. The lower bound would follow from a construction of n points and n hyperplanes with $\Omega(n^{2d/(d+1)})$ incident pairs, such that no d points lie on the intersection of d hyperplanes, or in other words, such that the bipartite incidence graph of P and H does not have $K_{d,d}$ as a subgraph. (The results of Clarkson et al. [23] and of Guibas, Overmars, and Robert [35] imply that this is the smallest forbidden subgraph for which the desired lower bound is possible.) An upper bound of $\pi_d^\circ(n,n) = O(n^{2d/(d+1)}2^{O(\log^* n)})$ follows from the running time of Matoušek's algorithm for Hopcroft's problem [42], using the results in the next section.

6. Better offline lower bounds.

Recall that a *partitioning algorithm*, given a set of points and hyperplanes, constructs a partition graph (which may depend arbitrarily on the input, at no cost), preprocesses the points, and queries the hyperplanes, using the algorithms in Figure 1. For a *polyhedral* partitioning algorithm, the partition graph's query regions are all convex (or projective) polyhedra, each with at most r faces, where r is some fixed constant.[10]

In [31], it was shown that the worst-case running time of any partitioning algorithm that solves Hopcroft's point-hyperplane incidence problem, given n points and k hyperplanes as input, requires time $\Omega(n \log k + n^{2/3}k^{2/3} + k \log n)$ when $d = 2$, or $\Omega(n \log k + n^{5/6}k^{1/2} + n^{1/2}k^{5/6} + k \log n)$ for any $d \ge 3$. Here, by restricting our attention to polyhedral partitioning algorithms, we derive (slightly) better lower bounds in arbitrarily high dimensions.

[10]In most actual partitioning algorithms, every query region is either a simplex ($r = 2^{d+1}$) or a combinatorial cube ($r = 3^d + 1$).

Theorem 6.1. *Let P be a set of points and H a set of hyperplanes, such that no point lies on any hyperplane. The running time of any polyhedral partitioning algorithm that solves Hopcroft's problem, given P and H as input, is at least $\Omega(\pi(P, H))$.*

Proof. For any partitioning algorithm $\mathcal{A}$, let $T_\mathcal{A}(P, H)$ denote its running time given the points P and hyperplanes H as input. Recall that $T_\mathcal{A}(P, H)$ is defined as follows:

$$\begin{aligned} T_\mathcal{A}(P,H) &\triangleq \sum_{p\in P} \#\text{edges traversed by } p + \sum_{h\in H} \#\text{edges traversed by } h \\ &= \sum_{\text{edge } e} \big(\#\text{points traversing } e + \#\text{hyperplanes traversing } e\big). \end{aligned}$$

We say that a point or hyperplane *misses* an edge from v to w if it reaches v but does not traverse the edge. (It might still reach w by traversing some other edge.) Recall that each node in the partition graph has at most Δ outgoing edges, for some fixed constant Δ. Thus, for every edge that a point or hyperplane traverses, there are at most $\Delta - 1$ edges that it misses.

$$\begin{aligned} \Delta\cdot T_\mathcal{A}(P,H) \geq \sum_{\text{edge } e} \big(&\#\text{points traversing } e + \#\text{points missing } e \\ &+ \#\text{hyperplanes traversing } e + \#\text{hyperplanes missing } e\big). \end{aligned}$$

Call any edge that leaves a primal node a primal edge, and any edge that leaves a dual node a dual edge.

$$\begin{aligned} \Delta\cdot T_\mathcal{A}(P,H) \geq &\sum_{\substack{\text{primal}\\ \text{edge } e}} \big(\#\text{points traversing } e + \#\text{hyperplanes missing } e\big) \\ &+ \sum_{\substack{\text{dual}\\ \text{edge } e}} \big(\#\text{hyperplanes traversing } e + \#\text{points missing } e\big) \end{aligned}$$

For each primal edge e, let P_e be the set of points that traverse e, and let H_e be the set of hyperplanes that miss e. The edge e is associated with a query region Π, a polyhedron with at most r faces. The polyhedron Π separates P_e and H_e, since every point in P_e is contained in Π, and every hyperplane in H_e is disjoint from Π.

Similarly, for each dual edge e, let H_e be the set of hyperplanes that traverse it, and let P_e be the points that miss it. The associated polyhedral query region Π separates the dual points H_e^* and the dual hyperplanes P_e^*. By the definitions of separation and dual polyhedra, Π^* separates P_e and H_e.

Now our argument is similar to the proof of Theorem 3.5. Say that a node v *splits* a point p and a hyperplane h if both p and h reach v but no edge out of v is traversed by both p and h. For every point $p \in P$ and hyperplane $h \in H$, some node must split p and h, since otherwise p and h would both reach a leaf, and the output of the algorithm would be incorrect. Thus, for some outgoing edge e of this node, we have $p \in P_e$ and $h \in H_e$.

It follows that the collection of subset pairs $\{(P_e, H_e)\}$ is an r-polyhedral cover of P and H. The size of this cover is at least $\Delta \cdot T_\mathcal{A}(P, H)$ and, by definition, at most $\pi_r(P, H)$. ∎

We emphasize that in order for this lower bound to hold, no point can lie on a hyperplane. If some point lies on a hyperplane, then the trivial partitioning algorithm, whose partition graph consists of a single leaf, correctly "detects" the incident pair at no cost. This is consistent with the intuition that it is trivial to prove that some point lies on some hyperplane, but proving that no point lies on any hyperplane is more difficult.

COROLLARY 6.2. *The worst-case running time of any polyhedral partitioning algorithm that solves Hopcroft's problem, given n points and k hyperplanes in $\mathbb{R}^d$, is*

$$\Omega\left(n^{1-2/d(d+1)}k^{2/(d+1)} + n^{2/(d+1)}k^{1-2/d(d+1)}\right).$$

Again, our d-dimensional bound improves our $(d-1)$-dimensional bound only for certain values of n and k. We can combine the lower bounds for different dimensions into the following single expression:

$$\Omega\left(n\log k + \sum_{i=0}^{d}\left(n^{1-2/i(i+1)}k^{2/(i+1)} + n^{2/(i+1)}k^{1-2/i(i+1)}\right) + k\log n\right).$$

This lower bound was previously known for *arbitrary* partitioning algorithms for counting or reporting versions of Hopcroft's problem—Given a set of points and lines, return the number of point-hyperplane incidences, or a list of incident pairs—as well as for offline *halfspace* counting and reporting problems [31].

7. Preprocessing-query time tradeoffs.

Based on the offline results in the previous section, we now establish tradeoff lower bounds between preprocessing and query time for online hyperplane emptiness and related queries. These are the first such lower bounds for any range searching problem in any model of computation; preprocessing time is not even defined in earlier models such as semigroup arithmetic and pointer machines. In some instances, our bounds allow us to improve the space-time tradeoff bounds established in section 4.

THEOREM 7.1. *Any partition graph that supports line emptiness queries in time t after preprocessing time p satisfies the inequality $pt^2 = \Omega(n^2)$ in the worst case.*

Proof. Suppose $p < n^2$, since otherwise there is nothing to prove. Let $k = cp^{3/2}/n$, where c is a constant to be specified later. Note that $k = O(n^2)$, and since $p = \Omega(n\log n)$ by Theorem 3.5, we also have $k = \Omega(n^{1/2}\log^{1/2} n)$. Thus, there is a set of n points and k lines such that for any partition graph, the total time required to preprocess the n points and correctly answer the k line queries is at least $\alpha n^{2/3}k^{2/3} = \alpha c^{2/3}p$ for some positive constant α [31]. If we choose $c = (2/\alpha)^{3/2}$, the total query time is at least p. Thus, at least one query requires time at least $p/k = \Omega(n/p^{1/2})$. ◻

This lower bound almost matches the best known upper bound $pt^2 = O(n^2\log^{\varepsilon} n)$, due to Matoušek [42].

The following higher-dimensional bound follows from Corollary 6.2 using precisely the same argument.

THEOREM 7.2. *Any polyhedral partition graph that supports d-dimensional hyperplane emptiness queries in time t after preprocessing time p satisfies the inequalities $pt^{(d+2)(d-1)/2} = \Omega(n^d)$ and $pt^{2/(d-1)} = \Omega(n^{(d+2)/d})$ in the worst case. When $d \leq 3$, these bounds apply to arbitrary partition graphs.*

Although in general these bounds are far from optimal, there are two interesting special cases that match known upper bounds [17, 38, 42] up to polylogarithmic factors.

COROLLARY 7.3. *Any polyhedral partition graph that supports hyperplane emptiness queries after $O(n\,\mathrm{polylog}\,n)$ preprocessing time requires query time $\Omega(n^{(d-1)/d}/$

polylog n) *in the worst case. When $d \leq 3$, this bound applies to arbitrary partition graphs.*

Corollary 7.4. *Any polyhedral partition graph that supports hyperplane emptiness queries in $O(\operatorname{polylog} n)$ time requires preprocessing time $\Omega(n^d/\operatorname{polylog} n)$ in the worst case. When $d \leq 3$, this bound applies to arbitrary partition graphs.*

In any realistic model of computation, the size of a data structure is a lower bound on its preprocessing time. However, partition graphs can have large subgraphs that are never visited during the preprocessing phase or that cannot be visited by any query. In principle, since we do not charge for the actual construction of a partition graph, its size can be arbitrarily larger than its preprocessing time.

We say that a partition graph is *trim* if every edge that does not point to a leaf is traversed both while preprocessing some point and while answering some query. Given any partition graph, we can easily make it trim (trim it?) without increasing any of its resource bounds. Since $s \geq \Delta \cdot p$ for any trim partition graph, any asymptotic lower bound on the preprocessing time for a trim partition graph is also a lower bound on its size.

Corollary 7.5. *Any trim partition graph of size s that supports line emptiness queries in time t satisfies the inequality $st^2 = \Omega(n^2)$ in the worst case.*

This lower bound is optimal, up to constant factors. Chazelle [17] and Matoušek [42] describe a family of line query data structures satisfying the matching upper bound $st^2 = O(n^2)$.

Corollary 7.6. *Any trim polyhedral partition graph of size s that supports d-dimensional hyperplane emptiness queries in time t satisfies the inequality $st^{(d+2)(d-1)/2} = \Omega(n^d)$ in the worst case. In particular, if $t = O(\operatorname{polylog} n)$, then $s = \Omega(n^d/\operatorname{polylog} n)$. When $d \leq 3$, these bounds apply to arbitrary trim partition graphs.*

Corollary 7.7. *Any trim polyhedral partition graph of size s that supports d-dimensional hyperplane emptiness queries in time t satisfies the inequality $st^{2/(d-1)} = \Omega(n^{(d+2)/d})$ in the worst case. In particular, if $s = O(n \operatorname{polylog} n)$, then $t = \Omega(n^{(d-1)/d}/\operatorname{polylog} n)$. When $d \leq 3$, these bounds apply to arbitrary trim partition graphs.*

Corollary 7.6 is an improvement over Theorem 4.5 for all $s = \Omega(n^{d-1})$ or $t = O(n^{2(d-1)/d^3})$; and Corollary 7.7 is an improvement whenever $s = O(n^{1+2/(d^2+d)})$ or $t = \Omega(n^{1-2/d})$. (These bounds are conservative; the actual breakpoints are much messier.) The lower bounds for near-linear space and polylogarithmic query time are optimal up to polylogarithmic factors.

All of these lower bounds apply to hyperplane and halfspace counting and reporting queries as well, by Lemma 3.4. In fact, the results in [31] imply that for counting and reporting queries, the preprocessing-query tradeoffs apply to *arbitrary* partition graphs, and the space-time tradeoffs to arbitrary trim partition graphs, in all dimensions. Corollary 7.6 is always an improvement (although a small one) over the lower bound in Corollary 4.3.

8. Halfspace emptiness queries. The space and time bounds for the best hyperplane (or simplex) emptiness query data structures are only a polylogarithmic factor smaller than the bounds for hyperplane (or simplex) counting queries. The situation is entirely different for halfspace queries. The best halfspace counting data structure known requires roughly $\Omega(n^d)$ space to achieve logarithmic query time [17, 42]; whereas, the same query time can be achieved with $o(n^{\lfloor d/2 \rfloor})$ space if we only want to know whether the halfspace is empty [44].

Table 2
Best known upper bounds for halfspace emptiness queries.

	Space	Preprocessing	Query Time	Source
$d \le 3$	$O(n)$	$O(n \log n)$	$O(\log n)$	[21, 3, 26]
$d \ge 4$	$O(n^{\lfloor d/2 \rfloor} / \log^{\lfloor d/2 \rfloor} n)$	$O(n^{\lfloor d/2 \rfloor} / \log^{\lfloor d/2 \rfloor - \varepsilon} n)$	$O(\log n)$	[44]
	$O(n)$	$O(n^{1+\varepsilon})$	$O(n^{1-1/\lfloor d/2 \rfloor} 2^{O(\log^* n)})$	[39]
	$O(n)$	$O(n \log n)$	$O(n^{1-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$	[44]
	$n \le s \le n^{\lfloor d/2 \rfloor}$	$O(s \operatorname{polylog} n)$	$O((n \operatorname{polylog} n)/s^{1/\lfloor d/2 \rfloor})$	[44]

Table 2 lists the resource bounds for the best known online halfspace emptiness data structures. The fastest offline algorithm, given n points and k halfspaces, requires

$$O\left(n \log k + (nk)^{\lfloor d/2 \rfloor/(\lfloor d/2 \rfloor + 1)} \operatorname{polylog}(n+k) + k \log n\right)$$

time to decide if any point lies in any halfspace [44]. In contrast, the only lower bounds previously known for halfspace emptiness queries are trivial. Linear space and logarithmic query time are required to answer online queries. A simple reduction from the set intersection problem shows that $\Omega(n \log k + k \log n)$ time is required for the offline problem in the algebraic decision tree and algebraic computation tree models [8, 50].

In this section, we derive the first nontrivial lower bounds on the complexity of halfspace emptiness queries. To prove our results, we use a simple reduction argument to transform hyperplane queries into halfspace queries in a higher-dimensional space [31, 29]. A similar transformation is described by Dwyer and Eddy [27].

Define the function $\sigma_d : \mathbb{R}^{d+1} \to \mathbb{R}^{\binom{d+2}{2}}$ as follows:

$$\sigma_d(x_0, x_1, \ldots, x_d) = \left(x_0^2, x_1^2, \ldots, x_d^2, \sqrt{2}\, x_0 x_1, \sqrt{2}\, x_0 x_2, \ldots, \sqrt{2}\, x_{d-1} x_d\right).$$

This map has the property that $\langle \sigma_d(p), \sigma_d(h) \rangle = \langle p, h \rangle^2$ for any vectors $p, h \in \mathbb{R}^{d+1}$, where $\langle \cdot, \cdot \rangle$ denotes the usual inner product. In a more geometric setting, σ_d maps points and hyperplanes in $\mathbb{R}^d$, represented as homogeneous coordinate vectors, to points and hyperplane in $\mathbb{R}^{d(d+3)/2}$, also represented in homogeneous coordinates. For any point p and hyperplane h in $\mathbb{R}^d$, the point $\sigma_d(p)$ is contained in the hyperplane $\sigma_d(h)$ if and only if p is contained in h; otherwise, $\sigma_d(p)$ is strictly above $\sigma_d(h)$. Thus, a hyperplane h intersects a point set P if and only if the closed lower halfspace $\sigma_d(h)^-$ intersects the lifted point set $\sigma_d(P)$. In other words, any (lower) halfspace emptiness data structure for $\sigma_d(P)$ is also a hyperplane emptiness data structure for P.

Unfortunately, this is not quite enough to give us our lower bounds, since the reduction does not preserve the model of computation. Specifically, the query regions in a partition graph used to answer d-dimensional queries must be subsets of $\mathbb{R}^d$. To complete the reduction, we need to show that the $d(d+3)/2$-dimensional partition graph can be "pulled back" to a d-dimensional partition graph.

In order for such a transformation to be possible, we need to restrict the query regions allowed in our partition graphs. A *Tarski cell* is a semialgebraic set defined by a constant number of polynomial equalities and inequalities, each of constant degree. Every Tarski cell has a constant number of connected components, and the intersection of any two Tarski cells is another Tarski cell (with larger constants). A *semialgebraic* partition graph is a partition graph whose query regions are all Tarski cells.

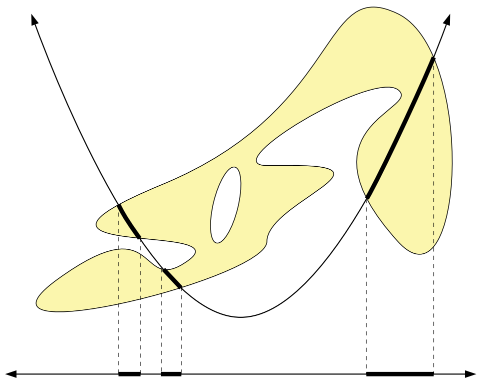


FIG. 5. *Each Tarski cell induces a constant number of lower-dimensional query regions.*

THEOREM 8.1. *Let P be a set of points in $\mathbb{R}^d$, and let $\hat{P} = \sigma_d(P)$. Given a semialgebraic partition graph $\hat{G}$ that supports $d(d+3)/2$-dimensional halfspace emptiness queries over $\hat{P}$, we can construct a semialgebraic partition graph G that supports d-dimensional hyperplane emptiness queries over P, with the same asymptotic space, preprocessing time, and query time bounds.*

Proof. We actually prove a stronger theorem, by assuming only that $\hat{G}$ supports emptiness queries for hyperplanes of the form $\sigma_d(h)$. Since no point in $\hat{P}$ is ever below such a hyperplane, any partition graph that supports lower halfspace emptiness queries also supports our restricted class of hyperplane emptiness queries. As we noted earlier, these queries are equivalent to hyperplane emptiness queries over the original point set P.

Given $\hat{G}$, we construct G as follows. G has the same set of nodes as $\hat{G}$, but with different query regions. Since each query region $\hat{R}$ in the original partition graph $\hat{G}$ is a Tarski cell, it intersects the algebraic surface $\sigma_d(\mathbb{R}^d)$ in a constant number of connected components $\hat{R}_1, \hat{R}_2, \ldots, \hat{R}_\delta$, where the constant δ depends on the number and degree of the inequalities that define $\hat{R}$. The query regions in G are the preimages $R_i = \sigma_d^{-1}(\hat{R}_i)$ of these components. See Figure 5.

The edge associated with each d-dimensional query region R_i has the same endpoints as the edge associated with the original query region $\hat{R}$. Thus, there may be several edges in G with the same source and target. (Recall that partition graphs are directed acyclic *multi*graphs.) If a query region $\hat{R}$ does not intersect $\sigma_d(\mathbb{R}^d)$, then the corresponding edge in $\hat{G}$ is not represented in G at all, so G may not be a connected graph. Nodes in G that are not connected to the root can be safely discarded. The size, preprocessing time, and query time for G are clearly at most a constant factor more than the corresponding resources for $\hat{G}$.

The leaf subsets P_ℓ in G cannot be larger than the corresponding subsets $\hat{P}_\ell$ in $\hat{G}$. (They might be smaller, but that only helps us.) Similarly, a hyperplane query cannot reach more leaves in G than the corresponding query reaches in $\hat{G}$. It follows that G supports hyperplane emptiness queries: For any hyperplane h, if $\hat{G}$ reports that $\sigma_d(h)$ is empty, G (correctly) reports that h is empty. □

We emphasize that some restriction on the query regions is necessary to prove any nontrivial lower bounds for halfspace emptiness queries. There is a partition graph of constant size, requiring only linear preprocessing, that supports halfspace emptiness queries in constant time. The graph consists of a single primal node with two query regions—the convex hull of the points and its complement—and two leaves. On the other hand, our restriction to Tarski cells is stronger than necessary. It suffices that every query region intersects (some projective transformation of) the surface $\sigma_d(\mathbb{R}^d)$ in a constant number of connected components.

The following corollaries are now immediate consequences of our earlier results.

Corollary 8.2. *For any $d \geq 2$, any semialgebraic partition graph that supports d-dimensional halfspace emptiness queries has size $\Omega(n)$, preprocessing time $\Omega(n \log n)$, and worst-case query time $\Omega(\log n)$.*

Corollary 8.3. *Any semialgebraic partition graph of size s that supports $d(d+3)/2$-dimensional halfspace emptiness queries in time t satisfies the inequality $st^d = \Omega((n/\log n)^{d-(d-1)/(d+1)})$ in the worst case.*

Corollary 8.4. *The worst-case running time of any semialgebraic partitioning algorithm which, given n points and k halfspaces in $\mathbb{R}^d$, decides if any halfspace contains a point, is $\Omega(n \log k + k \log n)$ for all $2 \leq d \leq 4$, $\Omega(n \log k + n^{2/3}k^{2/3} + k \log n)$ for all $5 \leq d \leq 8$, and $\Omega(n \log k + n^{5/6}k^{1/2} + n^{1/2}k^{5/6} + k \log n)$ for all $d \geq 9$.*

Corollary 8.5. *Any semialgebraic partition graph that supports 5-dimensional halfspace emptiness queries in time t after preprocessing time p satisfies the inequality $pt^2 = \Omega(n^2)$ in the worst case. Any trim semialgebraic partition graph of size s that supports 5-dimensional halfspace emptiness queries in time t satisfies the inequality $st^2 = \Omega(n^2)$ in the worst case.*

Corollary 8.6. *Any semialgebraic partition graph that supports 9-dimensional halfspace emptiness queries in time t after preprocessing time p satisfies the inequalities $pt^5 = \Omega(n^3)$ and $pt = \Omega(n^{5/3})$ in the worst case. Any trim semialgebraic partition graph of size s that supports 9-dimensional halfspace emptiness queries in time t satisfies the inequalities $st^5 = \Omega(n^3)$ and $st = \Omega(n^{5/3})$ in the worst case.*

Corollaries 8.2 and 8.4 are optimal when $d \leq 3$; Corollary 8.4 is also optimal up to polylogarithmic factors when $d = 5$; and Corollary 8.5 is optimal up to polylogarithmic factors.

Theorem 8.1 does not imply better offline lower bounds or preprocessing/query tradeoffs for halfspace emptiness queries in dimensions higher than 9, since the corresponding hyperplane results require polyhedral query regions. Marginally better lower bounds can be obtained directly in dimensions 14 and higher(!) in the *polyhedral* partition graph model by generalizing the arguments in sections 5 and 6 (as in [30]). However, since these lower bounds are far from optimal, we omit further details.

9. Conclusions.

We have presented the first nontrivial lower bounds on the complexity of hyperplane and halfspace emptiness queries. Our lower bounds apply to a broad class of range query data structures based on recursive decomposition of primal and/or dual space.

The lower bounds we developed for counting and reporting queries actually apply to any type of query where the points in the query range are required as the union of several subsets. For example, simplex range searching data structures are typically constructed by composing several levels of halfspace "counting" data structures [42]. To answer a query for the intersection of k halfspaces, the points in the first halfspace are (implicitly) extracted as the disjoint union of several subsets, and a $(k-1)$-halfspace query is recursively performed on each subset.

With a few notable exceptions, our lower bounds are far from the best known upper bounds, and a natural open problem is to close the gap. In particular, we have only "trivial" lower bounds for 4-dimensional halfspace emptiness queries. We conjecture that the correct space-time tradeoffs are $st^d = \Theta(n^d)$ for hyperplanes and $st^{\lfloor d/2 \rfloor} = \Theta(n^{\lfloor d/2 \rfloor})$ for halfspaces. Since these bounds are achieved by current algorithms—exactly for hyperplanes [42], within polylogarithmic factors for halfspaces [44]—the only way to prove our conjecture is to improve the lower bounds.

Our space-time tradeoffs derive from lower bounds for halfspace queries in the semigroup arithmetic model [10], and our preprocessing-query tradeoffs follow from lower bounds on the combinatorial complexity of polyhedral covers. Any improvements to these lower bounds would improve our results as well. Both of these results ultimately reduce to bounds on the minimum size of a decomposition of the (weighted) incidence graph of a set of points and a set of halfspaces into complete bipartite subgraphs.

The best known data structures for d-dimensional hyperplane emptiness queries and $2d$- or $(2d+1)$-dimensional halfspace emptiness queries have the same resource bounds. We conjecture that this is also true of *optimal* data structures for these problems. Is there a reduction from hyperplane queries to halfspace queries that only increases the dimension by a constant factor (preferably two)?

Finally, can our techniques be applied to other closely related problems, such as nearest neighbor queries [2], linear programming queries [40, 11] and ray shooting queries [2, 20, 41, 44]?

Acknowledgments. I thank Pankaj Agarwal for suggesting studying the complexity of online emptiness problems.

REFERENCES

[1] P. K. Agarwal and J. Erickson, *Geometric range searching and its relatives*, in Advances in Discrete and Computational Geometry, B. Chazelle, J. E. Goodman, and R. Pollack, eds., Contemp. Math. 223, Amer. Math. Soc., Providence, RI, 1999, pp. 1–56.

[2] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.

[3] A. Aggarwal, M. Hansen, and T. Leighton, *Solving query-retrieval problems by compacting Voronoi diagrams*, in Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing, Baltimore, MD, 1990, pp. 331–340.

[4] A. Andersson, *Sublogarithmic searching without multiplications*, in Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, Milwaukee, WI, 1995, pp. 655–663.

[5] A. Andersson, *Faster deterministic sorting and searching in linear space*, in Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 135–141.

[6] A. Andersson and K. Swanson, *On the difficulty of range searching*, Comput. Geom., 8 (1997), pp. 115–122.

[7] S. Arya and D. M. Mount, *Approximate range searching*, in Proceedings of the 11th Annual ACM Symposium on Computational Geometry, Vancouver, British Columbia, Canada, 1995, pp. 172–181.

[8] M. Ben-Or, *Lower bounds for algebraic computation trees*, in Proceedings of the 15th Annual ACM Symposium on the Theory of Computing, Boston, MA, 1983, pp. 80–86.

[9] A. Borodin, R. Ostrovsky, and Y. Rabani, *Lower bounds for high dimensional nearest neighbor search and related problems*, in Proceedings of the 31st Annual ACM Symposium on the Theory of Computing, Atlanta, GA, 1999, pp. 312–321.

[10] H. Brönnimann, B. Chazelle, and J. Pach, *How hard is halfspace range searching?*, Discrete Comput. Geom., 10 (1993), pp. 143–155.

[11] T. M. Chan, *Fixed-dimensional linear programming queries made easy*, in Proceedings of the 12th Annual ACM Symposium on Computational Geometry, Philadelphia, PA, 1996, pp. 284–290.

[12] T. M. Chan, *Output-sensitive results on convex hulls, extreme points, and related problems*, Discrete Comput. Geom., 16 (1996), pp. 369–387.

[13] B. Chazelle, *A functional approach to data structures and its use in multidimensional searching*, SIAM J. Comput., 17 (1988), pp. 427–462.

[14] B. Chazelle, *Lower bounds on the complexity of polytope range searching*, J. Amer. Math. Soc., 2 (1989), pp. 637–666.

[15] B. Chazelle, *Lower bounds for orthogonal range searching,* I: *The reporting case*, J. ACM, 37 (1990), pp. 200–212.

[16] B. Chazelle, *Lower bounds for orthogonal range searching,* II: *The arithmetic model*, J. ACM, 37 (1990), pp. 439–463.

[17] B. Chazelle, *Cutting hyperplanes for divide-and-conquer*, Discrete Comput. Geom., 9 (1993), pp. 145–158.

[18] B. Chazelle, *Lower bounds for off-line range searching*, Discrete Comput. Geom., 17 (1997), pp. 53–66.

[19] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, *Algorithms for bichromatic line segment problems and polyhedral terrains*, Algorithmica, 11 (1994), pp. 116–132.

[20] B. Chazelle and J. Friedman, *Point location among hyperplanes and unidirectional ray-shooting*, Comput. Geom., 4 (1994), pp. 53–62.

[21] B. Chazelle, L. J. Guibas, and D. T. Lee, *The power of geometric duality*, BIT, 25 (1985), pp. 76–90.

[22] B. Chazelle and B. Rosenberg, *Simplex range reporting on a pointer machine*, Comput. Geom. Theory Appl., 5 (1996), pp. 237–247.

[23] K. Clarkson, H. Edelsbrunner, L. J. Guibas, M. Sharir, and E. Welzl, *Combinatorial complexity bounds for arrangements of curves and spheres*, Discrete Comput. Geom., 5 (1990), pp. 99–160.

[24] K. L. Clarkson, *An algorithm for approximate closest-point queries*, in Proceedings of the 10th Annual ACM Symposium on Computational Geometry, Stony Brook, NY, 1994, pp. 160–164.

[25] M. de Berg, M. Overmars, and O. Schwarzkopf, *Computing and verifying depth orders*, SIAM J. Comput., 23 (1994), pp. 437–446.

[26] D. P. Dobkin and D. G. Kirkpatrick, *Determining the separation of preprocessed polyhedra – a unified approach*, in Proceedings of the 17th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 443, Springer-Verlag, New York, 1990, pp. 400–413.

[27] R. Dwyer and W. Eddy, *Maximal empty ellipsoids*, Internat. J. Comput. Geom. Appl., 6 (1996), pp. 169–186.

[28] H. Edelsbrunner and M. H. Overmars, *Batched dynamic solutions to decomposable searching problems*, J. Algorithms, 6 (1985), pp. 515–542.

[29] J. Erickson, *On the relative complexities of some geometric problems*, in Proceedings of the Seventh Canadian Conference on Computational Geometry, Quebec City, Quebec, Canada, 1995, pp. 85–90.

[30] J. Erickson, *New lower bounds for halfspace emptiness*, in Proceedings of the 37th Annual IEEE Symposium on the Foundations of Computer Science, Burlington, VT, 1996, pp. 472–481.

[31] J. Erickson, *New lower bounds for Hopcroft's problem*, Discrete Comput. Geom., 16 (1996), pp. 389–418.

[32] J. Erickson, *Space-time tradeoffs for emptiness queries*, in Proceedings of the 13th Annual ACM Symposium on Computational Geometry, Nice, France, 1997, pp. 304–313.

[33] M. L. Fredman, *Lower bounds on the complexity of some optimal data structures*, SIAM J. Comput., 10 (1981), pp. 1–10.

[34] J. E. Goodman and R. Pollack, *Allowable sequences and order types in discrete and computational geometry*, in New Trends in Discrete and Computational Geometry, J. Pach, ed., Algorithms Combin. 10, Springer-Verlag, Berlin, 1993, pp. 103–134.

[35] L. Guibas, M. Overmars, and J.-M. Robert, *The exact fitting problem in higher dimensions*, Comput. Geom., 6 (1996), pp. 215–230.

[36] M. Henk, J. Richter-Gebert, and G. M. Ziegler, *Basic properties of convex polytopes*, in Handbook of Discrete and Computational Geometry, J. E. Goodman and J. O'Rourke, eds., CRC Press LLC, Boca Raton, FL, 1997, pp. 243–270.

[37] J. Kleinberg, *Two algorithms for nearest-neighbor search in high dimension*, in Proceedings of the 29th Annual ACM Symposium on the Theory of Computing, El Paso, TX, 1997, pp. 599–608.

[38] J. Matoušek, *Efficient partition trees*, Discrete Comput. Geom., 8 (1992), pp. 315–334.

[39] J. Matoušek, *Reporting points in halfspaces*, Comput. Geom., 2 (1992), pp. 169–186.

[40] J. Matoušek, *Linear optimization queries*, J. Algorithms, 14 (1993), pp. 432–448.

[41] J. Matoušek, *On vertical ray shooting in arrangements*, Comput. Geom., 2 (1993), pp. 279–285.

[42] J. Matoušek, *Range searching with efficient hierarchical cuttings*, Discrete Comput. Geom., 10 (1993), pp. 157–182.

[43] J. Matoušek, *Geometric range searching*, ACM Comput. Surv., 26 (1994), pp. 421–461.

[44] J. Matoušek and O. Schwarzkopf, *On ray shooting in convex polytopes*, Discrete Comput. Geom., 10 (1993), pp. 215–232.

[45] P. B. Milterson, *Lower bounds for Union-Split-Find related problems on random access machines*, in Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 625–634.

[46] P. B. Milterson, N. Nisan, S. Safra, and A. Widgerson, *On data structures and asymmetric communication complexity*, J. Comput. System Sci., 57 (1998), pp. 37–49.

[47] M. H. Overmars, *Efficient data structures for range searching on a grid*, J. Algorithms, 9 (1988), pp. 254–275.

[48] J. Pach and P. K. Agarwal, *Combinatorial Geometry*, John Wiley & Sons, New York, 1995.

[49] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[50] J. M. Steele and A. C. Yao, *Lower bounds for algebraic decision trees*, J. Algorithms, 3 (1982), pp. 1–8.

[51] J. Stolfi, *Oriented Projective Geometry: A Framework for Geometric Computations*, Academic Press, New York, 1991.

[52] L. Székely, *Crossing numbers and hard Erdős problems in discrete geometry*, Combin., Probab., Comput., 6 (1997), pp. 353–358.

[53] E. Szemerédi and W. Trotter, Jr., *Extremal problems in discrete geometry*, Combinatorica, 3 (1983), pp. 381–392.

[54] R. E. Tarjan, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[55] D. E. Willard, *Log-logarithmic worst case range queries are possible in space* $\Theta(n)$, Inform. Process. Lett., 17 (1983), pp. 81–89.

[56] A. C. Yao, *On the complexity of maintaining partial sums*, SIAM J. Comput., 14 (1985), pp. 277–288.

[57] A. C.-C. Yao, *Should tables be sorted?*, J. ACM, 28 (1981), pp. 615–628.

[58] G. M. Ziegler, *Lectures on Polytopes*, Grad. Texts in Math. 152, Springer-Verlag, Heidelberg, 1994.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1997–2015

PARALLEL SORTING WITH LIMITED BANDWIDTH[*]

MICAH ADLER[†], JOHN W. BYERS[‡], AND RICHARD M. KARP[§]

Abstract. We study the problem of sorting on a parallel computer with limited communication bandwidth. By using the PRAM(m) model, where p processors communicate through a globally shared memory which can service m requests per unit time, we focus on the trade-off between the amount of local computation and the amount of interprocessor communication required for parallel sorting algorithms. Our main result is a lower bound of $\Omega(\frac{n \log m}{m \log n})$ on the time required to sort n numbers on the exclusive-read and queued-read variants of the PRAM(m). We also show that Leighton's Columnsort can be used to give an asymptotically matching upper bound in the case where m grows as a fractional power of n. The bounds are of a surprising form in that they have little dependence on the parameter p. This implies that attempting to distribute the workload across more processors while holding the problem size and the size of the shared memory fixed will not improve the optimal running time of sorting in this model. We also show that both the lower and the upper bounds can be adapted to bridging models that address the issue of limited communication bandwidth: the LogP model and the bulk-synchronous parallel (BSP) model. The lower bounds provide further convincing evidence that efficient parallel algorithms for sorting rely strongly on high communication bandwidth.

Key words. parallel sorting, limited bandwidth, PRAM, LogP, BSP

AMS subject classifications. 68W10, 68Q17

PII. S0097539797315884

1. Introduction.

A large body of theoretical research has concentrated on algorithms designed in the parallel random access machine (PRAM) model of computation. The PRAM allows processors to communicate with each other in unit time through a large globally shared memory, which leads to algorithms that have a high degree of parallelism but perform a great deal of interprocessor communication, an inexpensive operation in the PRAM model. This leaves unresolved the question of how to design algorithms for machines which have limited interprocessor communication bandwidth.

Addressing this limitation has motivated the development of other models of parallel computation, representative of which are the BSP model [33], the LogP model [15], and the PRAM(m) model [35]. Provably efficient algorithms in the PRAM model are not necessarily the most efficient algorithms for these models, so a host of problems must be reevaluated in this framework. In this paper, we examine the problem of sorting in the context of parallel machines with limited communication bandwidth. We formalize the sorting problem as follows.

DEFINITION 1.1. **The Sorting Problem.**
Input: *n distinct keys $k_1 \dots k_n$, with total order $k_{(1)} < k_{(2)} < \cdots < k_{(n)}$.*

[*]Received by the editors January 31, 1997; accepted for publication (in revised form) June 24, 1999; published electronically April 11, 2000. This research was supported in part by the National Science Foundation operating grant CCR-9005448. A large portion of this work was done while the authors were affiliated with the University of California, Berkeley, and the International Computer Science Institute in Berkeley, CA.

http://www.siam.org/journals/29-6/sicomp/31588.html

[†]Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (micah@cs.umass.edu).

[‡]Department of Computer Science, Boston University, Boston, MA 02215 (byers@cs.bu.edu).

[§]Department of Computer Science, University of California, Berkeley, CA, and International Computer Science Institute, Berkeley, CA 94720 (karp@cs.berkeley.edu).

Output at processor i: *A sorted list of keys:* $k_{(\frac{in}{p}+1)} \cdots k_{(\frac{in}{p}+\frac{n}{p})}$.

We concentrate on the complexity of sorting in the PRAM(m) model. In this variant of the classical PRAM model, p processors communicate through a globally shared memory consisting of m memory cells and the entire input, assumed to be of size n, is provided to each processor in a globally shared read-only memory (ROM). This model allows us to focus on the trade-off between the amount of information derived from local computation and the amount of information derived from interprocessor communication.

Each of the m shared memory cells consists of $\log n$ bits. As in traditional PRAM models, the resolution of contention for these shared memory cells can be defined in a variety of ways. In this paper, we focus on the exclusive-read and queued-read variants of the PRAM(m) model (ER PRAM(m) and QR PRAM(m), respectively); these are defined in section 2. The main result of this paper is the proof of a lower bound that holds for both the ER PRAM(m) and the QR PRAM(m) on the time required to sort n distinct keys of

$$\Omega\left(\frac{n \log m}{m \log n}\right).$$

The bound holds when $n > p^2$, which is the case of primary interest, since typical parallel applications involve problems where the input size is much larger than the number of processors. This lower bound does not rely on any restriction on the local computation of a processor. This is in contrast to sorting results which prove lower bounds on comparison-based algorithms. The proof extends to both Monte Carlo and Las Vegas randomized algorithms and to algorithms which allow for the m shared memory cells to employ a concurrent write contention resolution rule.

In order to prove the lower bound, we introduce the *oracle model of computation*, a model that allows us to quantify a trade-off between local computation and information received from other processors. In this model, which is defined more fully in section 2.1, processors are not required to transmit any information. Rather, all interprocessor communication is simulated by an oracle that is assumed to know the entire input before computation begins. The oracle is of unlimited computational power, and thus can precompute any function of the inputs before computation begins. We prove that even in this setting, local computation is of such limited utility that the oracle must provide a large amount of information in order to enable the processors to solve the sorting problem efficiently.

When $m = O(n^\beta)$, for some $\beta < 1$, we show that a version of Columnsort [26] has a running time that is bounded by

$$O\left(\frac{n}{m}(1-\beta)^{-3.42}\right).$$

For $n \gg m$, the case of greatest interest, the final factor becomes a small constant and so in this setting the ratio between the upper and lower bounds is $\Theta(\frac{\log n}{\log m})$. When sorting k-bit keys, the algorithm used in the upper bound runs with a slowdown of a factor of $O(\frac{k}{\log n})$ on a machine model in which the input is distributed among all processors rather than stored in a globally shared ROM. This gives the algorithm more credibility from a practical standpoint.

We also show that our results can be generalized to two other models that incorporate limited communication throughput, the LogP model and the BSP model. In both of these models, the processors communicate using point-to-point messages, and a parameter g represents the minimum number of cycles between transmission of successive messages from a processor. To prove a lower bound for the ER PRAM(m) model, we show a lower bound on the number of bits which must be transmitted through the network in order to solve the sorting problem efficiently. Coupling this bound with model-dependent lower bounds on the amount of time required to transmit a fixed number of bits in the BSP and LogP models results in lower bounds for sorting in these two models. We defer definitions of the models and the exact form of the bounds to the section where those results are discussed.

Our results show that fast parallel algorithms to solve the sorting problem must rely on large amounts of communication. Furthermore, we have the surprising result that both our upper and lower bounds are unaffected by attempting to distribute the work across an unlimited number of processors, while holding fixed the problem size, the size of the shared memory, and the number of processors that actually output the result. Therefore, to increase the speed of parallel sorting on a machine with limited communication bandwidth, increasing bandwidth is more likely to improve the running time than is increasing the number or computational power of processors.

The remainder of the paper is organized as follows. In the rest of section 1, we briefly compare our results with previous work in parallel sorting. In section 2, we provide a complete description of both the PRAM(m) model and our lower bound tool, the oracle model of communication. Sections 3 and 4 provide proofs of our lower bound for deterministic and randomized algorithms for sorting in the ER PRAM(m) and QR PRAM(m) models of computation. Section 5 provides the matching upper bound, and section 6 briefly describes extensions of those proofs to the LogP and BSP models.

1.1. Previous work. From the large body of research in the realm of parallel sorting algorithms, we discuss several results which also focus on interprocessor communication requirements. Using Thompson's VLSI model [32], Leighton, in [26], proves a lower bound of $\mathrm{AT}^2 = \Omega(n^2 \log^2 n)$ for sorting n keys of size $\Theta(\log n)$, where A is the area of a VLSI chip and T is the running time of the chip. His methods can be used to show bounds of the form $\Omega\left(\frac{n}{m}\right)$ in a PRAM model with a globally shared memory of size m, but in which the input is evenly distributed across the p processors, rather than stored in a globally shared ROM. Indeed, an interesting question would be to determine whether we could apply our lower bound technique to a nonstandard VLSI model in which the chip could receive each input in more than one location and at more than one time.

Other related work on parallel sorting includes [12], where Borodin and Cook prove that sorting requires $\mathrm{TIME} \cdot \mathrm{SPACE} = \Omega(\frac{n^2}{\log n})$. Aggarwal, Chandra, and Snir show in [3] that any parallel comparison-based algorithm that sorts n words requires $\Omega(\frac{n \log n}{p \log(\frac{n}{p})})$ communication steps. Also, the same authors show in [5] that sorting requires time $\Omega(\frac{n \log n}{p} + l \log p)$ in a model where reading or writing a block of size b from memory takes time $l + b$.

The PRAM(m) model was introduced in [35] and has been studied subsequently in [28], [19], [18], [9], [30], [1], and [10]. The case where $n \gg p$ was first examined in [30], where Mansour, Nisan, and Vishkin prove a lower bound of $\Omega(\frac{n}{\sqrt{mp}})$ for several problems, including sorting, in a concurrent read version of the PRAM(m), which implies the same bound in the ER PRAM(m) and the QR PRAM(m).

An easy upper bound on the time required for sorting can be obtained by using a variant of Cole's parallel merge sort [13] for the PRAM. Cole's algorithm uses

n processors to sort n keys in time O($\log n$) time. This algorithm requires the use of a total of $O(n \log n)$ shared memory cells per time step, but by letting each of the m words of shared memory simulate $O(\frac{n \log n}{m})$ cells of the PRAM memory, we can run Cole's algorithm in time O($\frac{n \log^2 n}{m}$) on the ER PRAM(m). Related work on upper bounds includes [16], in which Cypher and Sanz discuss a recursive version of Columnsort and introduce Cubesort, which can be used to obtain a running time of $O(\frac{n}{m}(1-\beta)^{-2})25^{\log^* n - \log^*(n/m)}$ for sorting on the ER PRAM(m), where $m = O(n^{\beta})$. However, this algorithm has substantial overhead and is considerably more involved then the one presented in this paper. A recursive version of Columnsort is also used by Aggarwal and Huang in [6] to obtain an algorithm for sorting in fixed connection networks.

Subsequent to a preliminary version of this paper in [2], Adler [1] provides an algorithm for sorting in the concurrent-read PRAM(m) (CR PRAM(m)) that is considerably faster than the lower bound for the ER PRAM(m) presented in this paper. Thus, that result together with the lower bound presented here imply that the CR PRAM(m) is strictly more powerful than the ER PRAM(m). [1] also slightly improves the ER PRAM(m) upper bound to $O(\frac{n \log p}{m \log n})$.

Finally, with respect to BSP algorithms for sorting, Gerbessiotis and Valiant [21] introduce a randomized algorithm for parallel sorting in the BSP model. Also, subsequent to an earlier version of this paper, the upper bound for sorting in the BSP model has been improved by both Goodrich [24] and by Gerbessiotis and Siniolakis [20]. The form of these bounds is deferred to section 5.2, where they are described in the context of our description of the BSP model.

2. The PRAM(m) model.

In this section, we define the PRAM(m) model, and then describe a theoretical tool derived from the PRAM(m) model, the oracle model of communication. The primary goal of these models is to examine the effectiveness of parallel computation given a sharp limitation on interprocessor communication.

In a classical PRAM, p processors communicate by writing to and reading from a large globally shared memory in unit time. However, in practice, the available per-processor bandwidth to shared memory can be quite small. Access to shared memories is slowed by such factors as long message send overheads [15], contention at memory banks, the fact that memory banks are much slower than processors [11], and bandwidth limitations of the network connecting processors to memory banks. Similar difficulties exist in distributed memory parallel machines. The parameter m of the PRAM(m) model focuses attention on this bottleneck, by enforcing the condition that the shared memory can service only m requests per unit time, where $m < p$. This is modeled as a PRAM consisting of m shared memory cells, each of size $\log n$ bits, as shown in Figure 2.1. We note that all the results in this paper can easily be extended to a model in which each memory cell can hold a word of w bits, independent of the input size.

The input of size n is provided to the PRAM(m) in a read-only shared memory (ROM) concurrently available to all of the processors. Conceptually, this is equivalent to having each processor begin with an identical copy of the input in its local memory. This capability serves to concentrate our lower bound efforts on the amount of communication required for actual computation, rather than on the amount required to distribute the input. Since such a ROM may be unrealistic from a practical standpoint, upper bounds achieved in this model that rely on use of the ROM are only applicable to problems in which the entire input is initially known by all the proces-

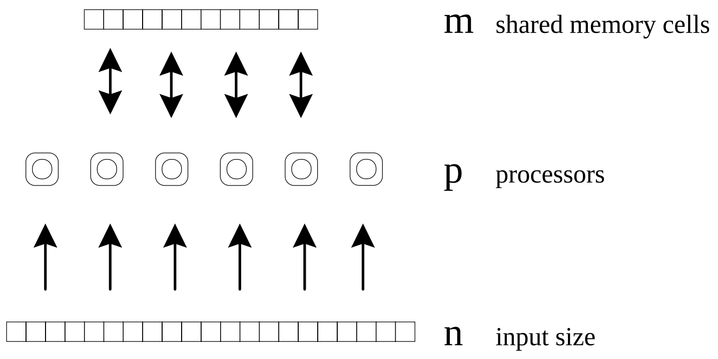


FIG. 2.1. *The PRAM(m) model.*

sors. During each synchronized round of computation, every processor can perform one of four actions: it can read the contents of a ROM location, read the contents of a globally shared memory location, write to a globally shared memory location, or perform local computation.

As defined in [35], the PRAM(m) model allows processors concurrent read, concurrent write access to the globally shared memory. In this paper, we consider exclusive-read and queued-read variants of the PRAM(m) model. In the ER PRAM(m), two distinct processors are forbidden from reading the same memory cell at the same time step. We also define the QR PRAM(m), where read contention at a memory cell is resolved as follows: each step of an algorithm completes in k time steps, where k is the maximum number of processors reading the same memory location during that step of the algorithm. Finally, we define the asynchronous QR PRAM(m), where every memory cell services one request if any requests to that cell are pending, and all other requests are stored in a FIFO queue. We note that both queued contention resolution strategies are analogous to those devised in [22] and [23] for the standard PRAM. The contention resolution strategy for write access to the shared memory can be either concurrent write, queued write, or exclusive write. Our upper and lower bounds are not affected by this choice, and thus, we leave this component of the model unspecified.

2.1. The oracle model of communication. It is often the case in parallel computing that the amount of computation required by a processor is greatly reduced by receiving results of computations performed by other processors. In order to quantify a trade-off between local computation and information received from other processors, we define the oracle model of computation. This lower bound tool uses the principle that the combined information a processor receives from all other processors is no more useful than the information it can receive from a single processor with unlimited computational resources and access to all the information the processors have.

In the oracle model, shown in Figure 2.2, processors do not transmit any information. Rather, each processor only receives information from an oracle of unlimited computational power, and a read-only memory (ROM) that contains the input. The oracle transmits information to the processors through p *oracle memories* consisting of cells of size $\log n$ bits. Each of these memory cells is referred to as an *oracle word*. Processor i has read-only access to the ith oracle memory but is not able to access any of the other oracle memories. We subject the oracle to the restriction that it compute

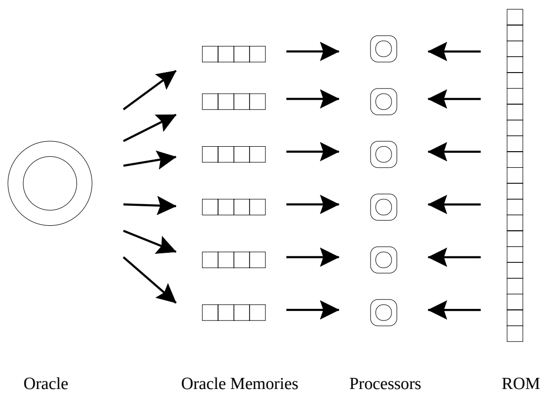


FIG. 2.2. *The flow of information in the oracle model.*

and set all the values of the oracle memory before the processors begin computation. This restriction does not alter the power of the model; it only serves to simplify the analysis.

The processors access the input using a concurrently readable ROM, which is identical to the ROM of the PRAM(m) model. During computation, at each synchronous time step every processor is allowed to perform one of three actions: it can read the contents of a cell from its oracle memory, read an input from the ROM, or perform local computation. The oracle knows the entire input and the programs executed by each of the processors. We are interested in the trade-off between the maximum number of time steps required by any processor and the total number of cells read from the oracle memory. Lower bounds on the number of cells that must be read from the oracle memory by all processors combined with the limited throughput of the memory give corresponding lower bounds on the execution time.

More formally, consider algorithms A_e and A_o designed for the ER PRAM(m) and the oracle model, respectively. Let $r(A_e, i, p)$ denote the aggregate number of reads the p processors perform from the shared memory and the ROM on input i. Likewise, let $t(A_e, i, p)$ denote the number of time steps A_e runs on i with p processors. Define $r(A_o, i, p)$ and $t(A_o, i, p)$ similarly for algorithm A_o.

DEFINITION 2.1. *An oracle algorithm A_o* exactly simulates *a PRAM(m) algorithm A_e if for all values of p and on all inputs i, $r(A_o, i, p) = r(A_e, i, p)$, $t(A_o, i, p) = t(A_e, i, p)$, and A_o computes the same output as A_e.*

LEMMA 2.2. *Given any ER PRAM(m) algorithm A_e, there is an oracle algorithm A_o such that A_o exactly simulates A_e.*

Proof. Consider the execution of the ER PRAM(m) algorithm A_e on input i. Let $w(u, v)$ denote the contents of the cell processor u would read at time v. The oracle can compute $w(u, v)$ for all u, v instantaneously in advance of the simulation by using its unlimited resources. To perform the simulation, the oracle simply furnishes $w(u, v)$ in oracle memory u at time v for all u, v. The processors then execute their ER PRAM(m) algorithms, ignoring all write operations and reading from their oracle memory in place of reading from shared memory locations. The simulation has zero slowdown and the processors read the same total number of cells in both executions. $\square$

Note that the oracle model can also exactly simulate any QR PRAM(m) algorithm, as well as any asynchronous QR PRAM(m) model. Furthermore, a similar lemma shows that an oracle model in which processors have concurrent read access to a single, m cell oracle memory can exactly simulate any CR PRAM(m) algorithm, but proving strong lower bounds for sorting in this model remains open. The oracle model can also be used to prove lower bounds for randomized algorithms. To allow the oracle to simulate the programs of processors in such a setting, we give the oracle access to the random bits used by each processor prior to the execution of the algorithm.

3. Lower bounds.

In this section, we prove lower bounds for sorting algorithms in the oracle model by showing that even when all processors know the range of keys that need to be output by each processor, the task of locating those keys within the input is difficult. If we wish to sort n keys, and if all processors know the range of key values that will be output by each processor, but not the value of these keys, nor their location within the ROM, then the work remaining can be formalized as the *permutation routing problem.*

DEFINITION 3.1.

The permutation routing problem.

Input: *n memory locations, each containing a processor ID, such that each processor ID appears exactly $\frac{n}{p}$ times.*

Output at processor i: *a list of the locations where i appears.*

LEMMA 3.2. *Any algorithm for the ER PRAM(m) that sorts n distinct keys in time T can solve any instance of the permutation routing problem of size n in time T. The same is true for the QR PRAM(m).*

Proof. We derive from each location of the permutation routing problem a key to be sorted, where the key is the concatenation of the processor ID stored at that location and the location index within the ROM. Sorting these keys is sufficient to inform each processor i of the locations where i appears. $\square$

Thus, any lower bound for the permutation routing problem implies an identical lower bound for sorting. In order to prove our lower bounds for this problem in different scenarios, we first prove a lower bound on a simpler problem in the oracle model. This problem is called the *processor d permutation routing problem*, and is defined as follows, with d any processor ID between 1 and p. The input is chosen uniformly at random from the set of all possible n element inputs to the permutation routing problem. Processor d is required to determine the list of locations in the ROM where d appears, but the remaining processors are not required to do anything. We are interested in the average, over all possible n element inputs to the permutation routing problem, of the number of oracle words processor d reads, given a limitation on how many ROM locations processor d reads.

LEMMA 3.3. *In any deterministic algorithm for the oracle model that solves the processor d permutation routing problem when $n > p^2$, if the average number of oracle words read by processor d is at most $\frac{n \log m}{8p \log n}$, and processor d never reads more than $\frac{n \log m}{m \log n}$ ROM locations, then on an input chosen uniformly at random, processor d produces an incorrect result with probability at least $\frac{1}{4}$.*

The proof of this lemma is the main technical portion of our lower bound, and is deferred to section 3.1. We first discuss its implications.

THEOREM 3.4. *For any deterministic ER PRAM(m) algorithm that solves the sorting problem for any set of n distinct keys, where $n > p^2$, the average over all permutations of the input keys of the time required by the algorithm is at least $\frac{n \log m}{8m \log n}$. The same is true for any deterministic algorithm for the QR PRAM(m).*

Proof. We assume there is a sorting algorithm A for the ER PRAM(m), where the average over all permutations of the input keys of the time to perform A is less than $\frac{n \log m}{8m \log n}$, and we reach a contradiction. Let $w(A, C)$ be the total number of words read from the shared memory when algorithm A is executed on input C. Let $r(A, C, i)$ be the number of ROM locations read by processor i when A is executed on input C. We shall use $w(A, C)$ and $r(A, C, i)$ to represent these quantities for both the PRAM(m) as well as the oracle model.

Since at most m words can be read from the shared memory at any time step, when C ranges over all permutations of the input keys, the average of $w(A, C)$ is less than $\frac{n \log m}{8 \log n}$. Also, the average over all such C of $\max_i r(A, C, i)$ is less than $\frac{n \log m}{8m \log n}$. By Lemmas 2.2 and 3.2, this implies the existence of an oracle algorithm A' for the permutation routing problem, where the average over all input permutations C of $w(A', C)$ is less than $\frac{n \log m}{8 \log n}$, and the average over all input permutations of $\max_i r(A, C, i)$ is less than $\frac{n \log m}{8m \log n}$.

For any such A', there is some processor d such that the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, and the average over all inputs of $r(A', C, d)$ is at most $\frac{n \log m}{8m \log n}$. By Markov's inequality, in A', the fraction of inputs C where $r(A', C, d) \geq \frac{n \log m}{m \log n}$ is at most $\frac{1}{8}$. Thus, we can use A' to construct algorithm A'' for the oracle model. In A'', processor d behaves the same as in A', except that it only performs at most $\frac{n \log m}{m \log n}$ ROM queries. If in A' processor d requires more ROM queries on a given input, on that input in A'', processor d returns an arbitrarily chosen permutation. Since algorithm A' responds correctly on all inputs, A'' responds correctly on at least $\frac{7}{8}$ of all possible inputs. This contradicts Lemma 3.3, and thus there does not exist such an algorithm A for the ER PRAM(m). The proof for the QR PRAM(m) is identical. ∎

For Las Vegas algorithms, or randomized strategies that are guaranteed to provide a correct solution with a bound only on the expected running time, we have a lower bound which follows from a direct application of Yao's lemma [36].

THEOREM 3.5. *For any Las Vegas algorithm A_v for the ER PRAM(m) where $n > p^2$, there is some permutation of the inputs* I *for the sorting problem such that A_v requires expected time at least $\frac{n \log m}{8m \log n}$ to solve* I. *Also, the expected running time of any Las Vegas algorithm on an input chosen uniformly at random from the set of all inputs is at least $\frac{n \log m}{8m \log n}$. The same is true for the QR PRAM(m).*

Proof. Yao's lemma [36] states that if there is a distribution over the inputs such that every deterministic algorithm requires time at least L for that distribution, then for any randomized algorithm there exists an input for which the expected running time is at least L. This combined with Theorem 3.4 directly implies the first claim of Theorem 3.5. The second claim follows from Theorem 3.4 and the fact that any Las Vegas algorithm is actually a distribution over deterministic algorithms and thus cannot fare better than the best deterministic algorithm. ∎

For Monte Carlo algorithms, or randomized strategies with bounded running time which provide a correct solution with probability greater than $\frac{3}{4}$, we have the following lower bound.

THEOREM 3.6. *For any Monte Carlo algorithm A_m for the ER PRAM(m) where $n > p^2$, there is some input* I *for the sorting problem such that A_m requires time at least $\frac{n \log m}{8m \log n}$ to solve* I. *Also, for the uniform distribution over all possible inputs, the running time of any Monte Carlo algorithm is at least $\frac{n \log m}{8m \log n}$. The same is true for* *the QR PRAM(m).*

We prove this theorem using an alternate formulation of Yao's lemma, provided in [29].

LEMMA 3.7. *Let P_1 be the success probability of a T step randomized algorithm solving problem B, where the success probability is taken over the random choices made by the algorithm and minimized over all possible inputs. Let P_2 be the success probability over a distribution $\mathcal{D}$ of inputs, maximized over all possible T step deterministic algorithms to solve B. Then, $P_1 \leq P_2$.*

Theorem 3.6 follows from a direct application of Lemma 3.7 to the following lemma.

LEMMA 3.8. *For any deterministic ER PRAM(m) algorithm A that solves the sorting problem for any set of n distinct keys, where $n > p^2$, if A always uses fewer than $\frac{n \log m}{8m \log n}$ time steps, then when the input is chosen uniformly at random from the set of all possible permutations of the inputs, the probability that every processor successfully produces the correct output is $\leq \frac{3}{4}$. The same is true for any deterministic algorithm for the QR PRAM(m).*

Proof. We assume that there is such a deterministic algorithm A_d which produces the correct output with probability greater than $\frac{3}{4}$, and we reach a contradiction. In such an algorithm, for every input C, $w(A_d, C) < \frac{n \log m}{8 \log n}$ and $\max_i r(A_d, C, i) < \frac{n \log m}{8m \log n}$. By Lemmas 2.2 and 3.2, this implies the existence of an oracle algorithm A'_d for the permutation routing problem, where the total number of oracle words read by all of the processors is less than $\frac{n \log m}{8 \log n}$, and $\max_i r(A'_d, C, i) < \frac{n \log m}{8m \log n}$. For any such A', there is some processor d such that the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, and $r(A'_d, C, d) \leq \frac{n \log m}{8m \log n}$. However, this implies the existence of an algorithm for the processor d permutation routing problem, where the average over all inputs of the number of oracle words read from the oracle memory for processor d is less than $\frac{n \log m}{8p \log n}$, $r(A'_d, C, d) < \frac{n \log m}{8m \log n}$, and yet processor d responds correctly with probability $> \frac{3}{4}$. This contradicts Lemma 3.3, and thus there does not exist such an algorithm A_d. ◻

3.1. The processor d permutation routing problem. In this subsection, we prove Lemma 3.3, restated here for convenience.

LEMMA 3.9. *In any deterministic algorithm for the oracle model that solves the processor d permutation routing problem when $n > p^2$, if the average number of oracle words read by processor d is at most $\frac{n \log m}{8p \log n}$, and processor d never reads more than $\frac{n \log m}{m \log n}$ ROM locations, then on an input chosen uniformly at random, processor d produces an incorrect result with probability at least $\frac{1}{4}$.*

For concreteness and simplicity, we represent an input to the permutation routing problem as a bit matrix B with n rows and p columns. If processor ID j appears in location i in the permutation routing problem instance, then $B_{ij} = 1$; otherwise $B_{ij} = 0$. Rows of B correspond to locations, and columns of B correspond to processors, so each column of B has exactly $\frac{n}{p}$ ones, and there is exactly one 1 in each row. A ROM query to location k reveals row k of this matrix. In order to solve the processor d permutation routing problem correctly, processor d must be able to specify column d of B exactly.

The proof of this lower bound employs the "little birdie" principle: giving a processor additional information never increases the complexity of the problem that the processor must solve. The side information that the "little birdie" reveals a priori

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

FIG. 3.1. *Permutation routing problem input:* $n = 6$, $p = 3$.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

FIG. 3.2. *A hidden matrix for processor* 1 *consistent with input in Figure* 3.1

to processor d is a perturbed representation of the input specification B called a *hidden matrix*. Providing this matrix to processor d allows us to prove lower bounds on the amount of information subsequent ROM queries provide to processor d.

The hidden matrix is chosen as follows. A permutation routing problem instance B is chosen uniformly at random and revealed to the oracle. Then, based on the choice of B, a hidden matrix H is chosen and revealed to processor d and the oracle. To construct the hidden matrix H, we first choose an $n \times p$ matrix G uniformly at random from binary matrices whose dth column is identical to the dth column of B, and the remaining columns each have either $\lfloor \frac{n}{p(p-1)} \rfloor$ 1's or $\lceil \frac{n}{p(p-1)} \rceil$ 1's, such that every row with a 1 in column d has exactly one 1 in some other column, and all other rows have no 1's. The hidden matrix H is defined to be $H = B \oplus G$, where $\oplus$ denotes the bitwise XOR of the two matrices. This mapping evenly redistributes the 1's from column d of B across the other columns while leaving all other rows unchanged. A pictorial representation of an input matrix and a possible hidden matrix constructed from it are given in Figures 3.1 and 3.2. Based on B and H, the oracle places some number of words in processor d's oracle memory. Processor d then executes its deterministic algorithm and produces its output.

We say that an input C is *consistent* with a hidden matrix H if there is a matrix G perturbing C as defined above such that $C \oplus G = H$. Let $\mathcal{C}(H)$ denote the set of inputs that are consistent with hidden matrix H. For deterministic algorithms, the pair $\langle C, H \rangle$, where C is an input consistent with hidden matrix H uniquely determines S, the setting of oracle memory d. We say that S is consistent with hidden matrix H if there is $C \in \mathcal{C}(H)$ such that H and C determine S. Let $\mathcal{S}(H)$ be the set of all settings of the oracle memory S that are consistent with H. Also, we say that input C is consistent with both H and S if H and C determine S. Let $\mathcal{C}(H, S)$ be the set of all inputs C that are consistent with H and S. Finally, for a hidden matrix H and an input $C \in \mathcal{C}(H)$, let the indicator variable $R(H, C) = 1$ if processor d produces the correct output on C when given H and 0 otherwise.

We first demonstrate that for any given hidden matrix, setting of the oracle memory, and algorithm, there cannot be too many inputs for which the algorithm produces the correct result.

CLAIM 3.9. *In any algorithm for the processor d permutation routing problem, where processor d is provided with any hidden matrix H and any setting of the oracle memory S, if processor d performs no more than $\frac{n \log m}{m \log n}$ ROM queries, then*

$$\sum_{C \in \mathcal{C}(H,S)} R(H,C) \leq Z, \text{ where } Z = \sum_{r=1}^{\frac{n}{p}} \binom{\frac{n \log m}{m \log n}}{r}.$$

Proof. After the setting of the oracle memory and the hidden matrix have been fixed, we can model processor i's actions by a decision tree, in which each node of the decision tree corresponds to a ROM query, and each leaf of the tree corresponds to a processor state achievable after performing at most $\frac{n \log m}{m \log n}$ ROM queries. The number of distinct results that processor d can produce is at most the number of leaves in this tree. We show that for any nonredundant algorithm, i.e., an algorithm that only examines each ROM location once, the tree has at most Z leaves. Any redundant algorithm can be simulated by a nonredundant algorithm, and thus the number of distinct results processor d can produce is at most Z for all algorithms.

Suppose processor d reads the value of row i of C from the ROM. Then, either (a) row i of H is identical to row i of C, or (b) C has a 1 in column d of row i whereas H has a 1 in some other column. Since processor d knows H at the start of the algorithm, the decision tree has branching factor two. Since processor ID d only appears in $\frac{n}{p}$ elements, at most $\frac{n}{p}$ of these ROM queries can discover elements where processor ID d appears. Mapping successful discoveries to left branches and unsuccessful queries to right branches ensures that any algorithm which is nonredundant has a decision tree where any path from root to leaf can have at most $\frac{n}{p}$ left branches. The number of distinct leaves of the decision tree that can be reached by a path from root to leaf with k left branches is at most $\binom{\frac{n \log m}{m \log n}}{k}$. Thus, the possible number of leaves in the decision tree is at most $\binom{\frac{n \log m}{m \log n}}{0} + \binom{\frac{n \log m}{m \log n}}{1} + \cdots + \binom{\frac{n \log m}{m \log n}}{n/p}$. ∎

For any algorithm for the processor d permutation routing problem, let $a(H)$ be the average number of oracle words provided to processor d, where the average is taken over all inputs in $\mathcal{C}(H)$.

CLAIM 3.10. *Consider any algorithm for the processor d permutation routing problem where $n > p^2$, where the little birdie provides processor d with a hidden matrix, and where processor d performs at most $\frac{n \log m}{m \log n}$ ROM queries. For any H, if $a(H) < \frac{n \log m}{4p \log n}$, then on an input chosen uniformly at random from the set $\mathcal{C}(H)$, the probability that processor d produces the correct output is $< \frac{1}{2}$.*

Proof. The total number of inputs in $\mathcal{C}(H)$ on which processor d responds correctly is at most

$$\sum_{C \in \mathcal{C}(H)} R(H,C) = \sum_{S \in \mathcal{S}(H)} \sum_{C \in \mathcal{C}(H,S)} R(H,C),$$

and so the probability of a successful response is at most

$$\frac{1}{|\mathcal{C}(H)|} \sum_{S \in \mathcal{S}(H)} \sum_{C \in \mathcal{C}(H,S)} R(H,C).$$

But, by Claim 3.9, this is at most Y, where

$$Y = \sum_{S \in \mathcal{S}(H)} \min\left(\frac{Z}{|\mathcal{C}(H)|}, \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}\right).$$

We assume that $Y \geq \frac{1}{2}$, and we reach a contradiction by showing that this implies that $a(H) \geq \frac{n \log m}{4p \log n}$. First, note that $Y \geq \frac{1}{2}$ implies that $|\mathcal{S}(H)| \geq \frac{|\mathcal{C}(H)|}{2Z}$. Let $|S|$ denote the number of words in oracle memory setting S. Because there are at most $2^{r \log n}$ settings of the oracle memory with $\leq r$ words, there are at least $\frac{3|\mathcal{C}(H)|}{8Z}$ oracle memory settings $S \in \mathcal{S}(H)$ such that

$$|S| \geq \frac{\log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{\log n}.$$

We call such oracle memory settings *large settings*. We use this to minimize $a(H)$ subject to $Y \geq \frac{1}{2}$. Note that

$$a(H) = \sum_{S \in \mathcal{S}(H)} |S| \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}.$$

By counting the total contribution to $a(H)$ of the large settings, we see that

$$a(H) \geq \frac{3|\mathcal{C}(H)|}{8Z} \cdot \frac{\log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{\log n} \cdot \frac{|\mathcal{C}(H,S)|}{|\mathcal{C}(H)|}.$$

We can assume that for any $S \in \mathcal{S}(H)$, $|\mathcal{C}(H,S)| \geq Z$, since given any valid solution, any sets $\mathcal{C}(H,S)$ of smaller cardinality can be combined into sets of cardinality Z without changing the value of Y and without increasing $a(H)$. Thus,

$$a(H) \geq \frac{3 \log\left(\frac{|\mathcal{C}(H)|}{8Z}\right)}{8 \log n}.$$

Using the fact that $n > p^2$, we have that for any H,

$$|\mathcal{C}(H)| \geq \binom{\lfloor \frac{n}{p-1} \rfloor}{\lfloor \frac{n}{p(p-1)} \rfloor}^{p-1}.$$

Using the inequality $\left(\frac{a}{b}\right)^b \leq \binom{a}{b} \leq \left(\frac{ae}{b}\right)^b$, and the fact that $m \leq p$ implies that the sum expressed by Z is dominated by the final term, this gives us

$$a(H) \geq \frac{3n \log m}{8p \log n} - o\left(\frac{n \log m}{p \log n}\right),$$

which is a contradiction. Thus, the probability of a correct response from processor d is less than $\frac{1}{2}$. $\square$

Proof of Lemma 3.3. The number of hidden matrices consistent with a given input is invariant over the choice of input and the number of inputs consistent with a given hidden matrix is invariant over the choice of matrix. Thus, if the average, over all inputs, of the number of oracle words read by processor d is at most $\frac{n \log m}{8p \log n}$, then the average of $a(H)$ over all H is at most $\frac{n \log m}{8p \log n}$. By Markov's inequality, this implies that $a(H) \leq \frac{n \log m}{4p \log n}$ for at least $\frac{1}{2}$ of the hidden matrices H. The input to the permutation routing problem is chosen uniformly at random from the set of all inputs. One method for producing this uniform distribution is as follows: first a hidden matrix H_i is chosen uniformly at random from the set of all hidden matrices that can be given to d. Then, an input C_i is chosen uniformly at random from $\mathcal{C}(H_i)$. With probability at least $\frac{1}{2}$, for the resulting choice of H_i, $a(H_i) \leq \frac{n \log m}{4p \log n}$. By Claim 3.10, if $a(H_i) \leq \frac{n \log m}{4p \log n}$, then the probability that processor d responds correctly is at most $\frac{1}{2}$. Therefore, the probability that processor d responds correctly on an input chosen uniformly at random, when given a hidden matrix, is at most $\frac{3}{4}$. By the little birdie principle, the probability that processor d responds correctly on an input chosen uniformly at random, when not given a hidden matrix, also is at most $\frac{3}{4}$. $\square$

4. The upper bound.

We show that a version of Leighton's Columnsort [26] performs well in both the ER PRAM(m) and the QR PRAM(m). Moreover, this algorithm runs in a model where there is no globally shared ROM for the input (which may not always be realistic in practice), but instead the input is distributed across the processor's local memories.

THEOREM 4.1. *There is an ER PRAM(m) algorithm for sorting n keys which runs in time*

$$O\left(\frac{n}{m}(1-\beta)^{-3.42)}\right),$$

provided $p \geq m \log n$ and $m = O(n^{\beta})$, for some $\beta < 1$. This algorithm has the same running time on the QR PRAM(m).

Proof. It is sufficient to provide a sorting algorithm for the ER PRAM(m). To do so, we use a recursive version of Columnsort, which we describe below. In Columnsort, the n keys are thought of as elements in a matrix M. There is a requirement on the aspect ratio of M: if M is an $s \times r$ matrix, then s must be larger than r^2. The elements are sorted using seven phases, where each phase is one of three types: phases that sort the columns of the matrix, phases that perform an odd-even transposition sort along the rows of the matrix, and phases that route a fixed permutation of the matrix elements, where each column routes an equal number of elements to every other column. The following simple description of Columnsort is provided in [27]. In phases 1, 3, and 7, the columns are sorted into increasing order. In phase 5, odd columns are sorted into increasing order and even columns are sorted into decreasing order. In phase 2, the matrix is "transposed": the items are picked up in column-major order and set down in row-major order (preserving the shape of the matrix). Phase 4 applies the reverse of the permutation applied in phase 2, and phase 6 performs two steps of odd-even transposition sort to each row.

We specify a call to recursive Columnsort in the ER PRAM(m) by two parameters: k, the number of keys to be sorted, and a, the number of memory cells dedicated to this function call. We develop the following recurrence relation for the running time of recursive Columnsort, where the keys are contained in a ROM of size n:

$$RC(k,a) = \text{ O } \left(\frac{k}{a}\right) \text{ if } k \geq a^3 \log n,$$

$$RC(k,a) = 4\ RC\left(k^{2/3},\ ak^{-1/3}\right) + \text{ O}\left(\frac{k}{a}\right) \qquad \text{otherwise.}$$

In the version of Columnsort we use to obtain this recurrence, each of the m memory cells is assigned a set of $\log n$ processors. These $m \log n$ processors sort the n keys, and inform each of the $p - m\log n$ remaining processors of the range of keys that they need to output. Note that since the algorithm makes effective use of only $m \log n$ processors, this algorithm is consistent with the observation that increasing communication throughput, as opposed to adding processors, is required for faster parallel sorting.

The base case for the recurrence, where $k \geq a^3 \log n$, works as follows. The k keys are thought of as being entries in a matrix M of keys of size $\frac{k}{a \log n} \times a \log n$; this matrix satisfies the aspect ratio requirement of Columnsort. We have $a \log n$ available processors, and each of these is assigned to one column of M. Thus, a memory cell serving a set of $\log n$ processors handles data transfer for $\log n$ columns. Recall that each memory cell is assigned to the set of $\log n$ columns to which its processors are assigned. We now show that we can implement each of the three types of phases of Columnsort on the PRAM(m) in time $\mathrm{O}(\frac{k}{a})$. Sorting the columns can be performed by each of the a processors locally in time $O(\frac{k \log k}{a \log n}) = O(\frac{k}{a})$ by any of a variety of known serial algorithms.

Routing the fixed permutation on the matrix elements requires each processor to send an identical number of keys to every other processor, and thus can be done with a single pass through all the entries. A single element is routed by the source and destination processors using the shared memory location that is assigned to the destination processor. The source processor writes to this memory cell the key's address in the ROM, and then the destination processor reads this address. Since each memory cell handles $\log n$ columns, and each column contains $\frac{k}{a \log n}$ keys, the total number of addresses written to each memory cell is $O(\frac{k}{a})$. Thus, the entire permutation can be routed in time $O(\frac{k}{a})$. One phase of odd-even transposition sort can be performed by each processor routing all the keys currently in its column to the processor that is assigned to the neighboring column. This can also be done in time $O(\frac{k}{a})$.

For the case where $k < a^3 \log n$, we sort the keys as follows. The keys are thought of as the elements in a matrix M of size $k^{2/3} \times k^{1/3}$. This matrix satisfies the aspect ratio of Columnsort, and thus we use Columnsort to sort this matrix as well. We can still route the permutations of the matrix M and perform the phases of odd-even transposition sort in time $\mathrm{O}(\frac{k}{a})$. This follows from the fact that for each permutation only k keys need to be routed through the shared memory and this operation can be performed without conflict while making use of each cell during each of the $\mathrm{O}(\frac{k}{a})$ time steps. For the phases which sort the columns of the matrix, we employ parallel calls to recursive Columnsort, one call per column. Each column consists of $k^{2/3}$ keys, and we evenly distribute the a available memory cells (with their associated processors) across the columns. Thus, each of the 4 sorting phases takes time RC $\left(k^{2/3}, ak^{-1/3}\right)$. An example which graphically describes one level of this recursive procedure is given in Figure 4.1.

The algorithm starts with a call to recursive Columnsort with n keys and m memory cells (with each assigned $\log n$ processors). After the n keys are sorted, the $m \log n$ processors can inform each of the remaining $p - m \log n$ processors of the correct sorted list of $\frac{n}{p}$ keys to output in the same amount of time as is required to route one permutation. To analyze the running time of sorting n keys on the ER PRAM(m), we evaluate the recurrence RC (n, m). The running time of the algorithm is dominated by the time for sorting at the bottom of the recursion at level j: $\mathrm{O}(\frac{n}{m}4^j)$. When m is a fixed function of n where $m = \mathrm{O}(n^\beta)$ for $\beta < 1$, this j is the smallest

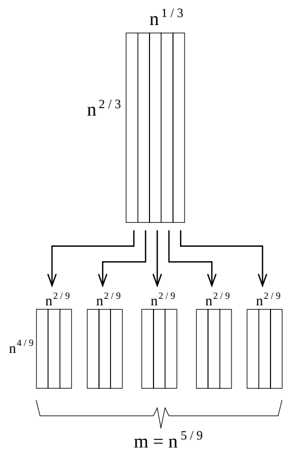


FIG. 4.1. *The recursive algorithm when $m = n^{5/9}$*

integer that satisfies $n^{((2/3)^j)} \leq n^{1-\beta}$, and the following bound on 4^j follows directly:

$$4^j \leq (1-\beta)^{\frac{2}{\log \frac{2}{3}}} \approx (1-\beta)^{-3.42}.$$

Substituting into the formula above gives an upper bound on the running time of $O(\frac{n}{m}(1-\beta)^{-3.42})$. □

5. Other limited bandwidth models. We can use the techniques discussed for the PRAM(m) to derive bounds in other parallel models which address the issue of limited communication throughput. We give a brief discussion of translating the ER PRAM(m) lower bound for sorting into the LogP model [15] and the BSP model [33] and state the best known upper bounds for sorting in these models.

5.1. The LogP model. In the LogP model, limited communication throughput in a parallel machine is enforced by requiring that each processor must wait for a gap of at least g cycles between the transmission of consecutive point-to-point messages. The three other LogP parameters are P, the number of processors, L, the latency of a message in the network, and o, the overhead (in cycles) to place a fixed-size message onto the network. Note that this model uses point-to-point messages for communication, as opposed to using the global shared memory used in the PRAM(m) model. We make the additional assumption that the point-to-point messages, or packets, have a maximum size w, measured in bits.

In this model, only P packets can be issued into the network each g time steps, and thus the throughput of the network is

$$m_L = \left\lceil \frac{wP}{g \log n} \right\rceil$$

$\log n$-bit words per machine cycle. We denote this expression for throughput by m_L to make plain its correspondence with m in the PRAM(m) model. In order to prove a lower bound for sorting in the LogP model, we first show that any LogP model algorithm can be simulated in the oracle model.

LEMMA 5.1. *If $w > \log g$, then given any LogP algorithm A_l that completes in time T, there is an oracle algorithm A_o that computes the same function as A_l, also in time T, and A_o writes at most $2m_LT$ words to each of the oracle memories.*

Proof. We partition the time steps of the LogP algorithm A_l into *epochs*, where each epoch consists of g consecutive time steps. Note that each processor receives at most one message during any epoch. We number the bits of each oracle memory, ignoring word boundaries. We can simulate A_l with an oracle algorithm A_o, where epoch i in A_l is represented in A_o by the oracle utilizing bits $(i-1)(\lfloor\log g\rfloor+1+w)+1$ through $i(\lfloor\log g\rfloor+1+w)+1$ in each oracle memory. The first $\lfloor\log g\rfloor+1$ of the bits for each epoch are used to inform processor j of whether or not a message arrives during that epoch, and in the case of an arrival, the exact time step of the arrival during the epoch. In the case of an arrival, the remaining w bits contain the message contents. In A_o, the processors execute their algorithms for A_l, ignoring steps where messages are sent, and reading from the oracle memory at the start of every epoch. The total number of bits read is $P(\lfloor\log g\rfloor+1+w)\lceil\frac{T}{g}\rceil$. When $w > \log g$, this is $O(m_L T\log n)$, and thus at most $O(m_L T)$ words are required in each oracle memory. □

We briefly point out why we require that the oracle give each processor the timing information provided by the $\lfloor\log g\rfloor+1$ additional bits used for each epoch. The LogP model is an asynchronous model, and thus processors cannot use the timing information to ensure the correctness of the algorithm. However, for the purpose of running time analysis, it is assumed that each processor behaves synchronously. Thus, in the optimal algorithm, it is possible that a processor is able to use the timing information to achieve a better running time than an algorithm that does not make inferences based on this information. Note that any algorithm that does not use this timing information can be simulated in the oracle model using at most $O(m_L T)$ words, even in the case where $w \leq g$.

As in the ER PRAM(m) model, when m_L grows as a fractional power of n, the time required to sort n keys is asymptotically no less than the time required to route all n keys through the network, even in the case where every processor knows all the keys in advance. Let $T_s(n)$ be the optimal sequential time required to sort the n keys.

Theorem 5.2. *In the LogP model, sorting n distinct keys requires expected time*

$$\Omega\left(\frac{T_s(n)}{P}+\frac{n\log m_L}{m_L\log n}+L+o+g\right),$$

provided that $n > P^2$, $w > \log g$, and $T_s(n) \geq L$. This bound holds even in the case that every processor has access to every key at the start of the algorithm.

Proof. We first assume there exists an algorithm A_l for the LogP model, where the average over all inputs of the time to perform A_l is at most $\frac{n\log m_L}{16 m_L\log n}$, and we reach a contradiction. By Lemma 5.1, such an algorithm A_l implies the existence of an oracle model algorithm where the average number of oracle words used is at most $\frac{n\log m_L}{8\log n}$, and the average of the maximum number of ROM queries by any processor is at most $\frac{n\log m_L}{8 m_L\log n}$. However, as we saw in Theorem 3.4, this leads to a contradiction of Lemma 3.3, and thus there does not exist such an algorithm A_l for the ER PRAM(m). The lower bound then follows from the fact that at least one transmission, which requires time at least $\max(L, o, g)$, is required in any algorithm that sorts in time faster than time $T_s(n)$, and that the time to sort on P processors is no faster than the optimal time to sort on one processor divided by P. □

A recursive implementation of Columnsort similar to that presented in section 4 can be tuned to deliver the following asymptotic performance.

Theorem 5.3. *In the LogP model, sorting n keys known to all processors can be* *completed in time*

$$O\left(T_s\left(\frac{n}{P}\right)+\frac{n}{m_L}+L+o+g\right),$$

provided that $P=O(n^\beta)$ for some constant $\beta<1$ and that $w\le\log n$.

The case where the input is distributed across the processors requires long keys to be sent in their entirety, rather than sending just the original index of the key. Otherwise, the algorithm, as well as the resulting bounds, are the same.

5.2. The BSP model. We now briefly describe analogous bounds for sorting in the BSP model proposed by Valiant [33], [34]. The model consists of a set of processors capable of transmitting point-to-point messages through a communication network and facilities for performing barrier synchronization across any subset of the processors. The three parameters of the model are P, the number of processors, L, the minimum number of local computation steps between successive synchronization operations, and g, the ratio between the throughput of local computation to the throughput at which a processor may inject point-to-point messages into the network. As in the LogP model, g enforces a limit on the communication throughput available to each processor. We assume that each transmitted packet is at most w bits in size.

Computation in the BSP model proceeds in *supersteps*, wherein each processor in parallel executes a task consisting of some number of local computation steps, message transmissions, and message receipts, subject to the constraints imposed by the parameter g. The superstep lasts for kL time steps, where k is the minimum integer such that all processors have completed their tasks before time kL. As in the LogP model, the total communication throughput in the BSP model is $m_B=\lceil\frac{wP}{g\log n}\rceil$ words per step of local computation, where each word consists of $\log n$ bits.

THEOREM 5.4. *In the BSP model, sorting n distinct keys requires time*

$$\Omega\left(\frac{T_s(n)}{P}+\frac{n\log m_B}{m_B\log n}+L+o+g\right)$$

when $n>P^2$ and $T_s(n)\ge L$.

Proof. Using the technique from Lemma 5.1, we see that the oracle model is also capable of simulating any BSP algorithm. Thus, the existence of any algorithm that sorts faster than $\frac{n\log m_B}{16m_B\log n}$ again implies the existence of an algorithm that contradicts Lemma 3.3. ◻

Using recursive Columnsort, it is straightforward to show that in the BSP model, sorting any n keys can be completed in time $\mathrm{O}(T_S(\frac{n}{P})+\frac{n}{m_B}+L+g+o)$, provided that $P=\mathrm{O}(n^\beta)$ for some constant $\beta<1$ and that $w\le\log n$. This result for sorting in the BSP model compares with the previous best randomized methods of Gerbessiotis and Valiant [21] for the BSP model with the assumption that each packet that is transmitted consists of exactly one key. Their algorithms run in time $\mathrm{O}(\frac{n\log n}{P}+gp^\epsilon+\frac{gn}{P}+L)$, with high probability, for any positive constant $\epsilon<1$, and for $P\le n^{1-\delta}$, where δ is a small constant depending on ϵ. After the preliminary version of this paper appeared in [2], work on this problem by Goodrich [24] tightened the bounds for sorting on the BSP, giving deterministic algorithms which run in time $\mathrm{O}(\frac{n\log n}{P}+(L+\frac{gn}{P})(\log n/\log(n/P)))$ for all values of P, coupled with a matching lower bound. Other recent work by Gerbessiotis and Siniolakis [20] gives a deterministic algorithm for sorting on the BSP which runs in time $(1+\mathrm{o}(1))(\frac{n\log n}{P}+L)+\mathrm{O}(\frac{gn}{P})$ for $P=n^{1-\epsilon}$, $0<\epsilon<1$, and uses 1-optimal local computation.

6. Conclusion. We have examined the problem of sorting on parallel models with limited communication bandwidth. Our main results include upper and lower bounds for sorting on exclusive- and queued-read variants of the PRAM(m) model which are asymptotically optimal for many practical settings of the parameters and are otherwise asymptotically tight to within at most a logarithmic factor. The form of our bound is noteworthy in that it demonstrates that all efficient parallel algorithms for sorting in this limited bandwidth model depend on large amounts of interprocessor communication. The techniques used to develop the bounds also apply to the LogP model and the BSP model, bridging models which consider the effect of limited bandwidth on parallel computation. For all three models of computation considered, when $m = \Omega(n^{\beta})$, the time to sort and the time to transmit all the keys through the shared memory (or the network) are asymptotically equivalent, even in the case where the entire input is known to each of the processors. Furthermore, as long as $n > p^2$, the bounds do not depend on the parameter p, so that when attempting to improve the performance of parallel sorting on machines with limited communication bandwidth, increasing communication bandwidth is more likely to be beneficial than increasing the number of processors. The lower bound, however, does not apply to the concurrent read version of the PRAM(m) originally introduced by Mansour, Nisan, and Vishkin in [30], and thus the asymptotic complexity of sorting in this model remains an open question.

Acknowledgments. We would like to thank Ralph Werchner for his useful comments and suggestions on an earlier version of this paper and for the valuable and insightful suggestions of the anonymous SICOMP referees.

REFERENCES

[1] M. Adler, *New coding techniques for improved bandwidth utilization*, in Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 173–182.

[2] M. Adler, J. Byers, and R. Karp, *Parallel sorting with limited bandwidth*, in Proceedings of the Seventh ACM Symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, 1995, pp. 129–136.

[3] A. Aggarwal, A. Chandra, and M. Snir, *Communication complexity of PRAMs*, Theoret. Comput. Sci., 71 (1990), pp. 3–28.

[4] A. Aggarwal, A. Chandra, and M. Snir, *Hierarchical memory with block transfer*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 204–216.

[5] A. Aggarwal, A. Chandra, and M. Snir, *On communication latency in PRAM computations*, in Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, 1989.

[6] A. Aggarwal and M.-D. A. Huang, *Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks*, in Proceedings of the Third Aegean Workshop on Computing, Lecture Notes in Comput. Sci. 319, 1988, pp. 339–350.

[7] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[8] M. Ajtai, J. Komlós, and E. Szemerédi, *An $O(n \log n)$ sorting network*, Combinatorica, 3 (1983), pp. 1–19.

[9] Y. Azar, *Lower bounds for threshold and symmetric functions in parallel computation*, SIAM J. Comput., 21 (1992), pp. 329–338.

[10] P. Beame, F. Fich, and R. Sinha, *Separating the power of CREW and EREW PRAMs with small communication width*, Inform. and Comput., 138 (1997), pp. 89–99.

[11] G. Blelloch, P. Gibbons, Y. Matias, and M. Zagha, *Accounting for memory bank contention and delay in high-bandwidth multiprocessors*, in Proceedings of the Seventh ACM Symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, 1995, pp. 84–94.

[12] A. Borodin and S. Cook, *A time-space tradeoff for sorting on a general sequential model of computation*, SIAM J. Comput., 11 (1982), pp. 287–297.

[13] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[14] S. Cook, C. Dwork, and R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[15] D. Culler, R. M. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: A practical model of parallel computation*, Commun. ACM, 39 (1996), pp. 78–85.

[16] R. Cypher and J. Sanz, *Cubesort: A parallel algorithm for sorting N data items with S-sorters*, J. Algorithms, 13 (1992), pp. 211–234.

[17] A. Dusseau, D. Culler, K. Schauser, and R. Martin, *Fast parallel sorting under LogP: Experience with the CM-*5, IEEE Trans. Parallel and Distributed Systems, 7 (1996), pp. 791–805.

[18] F. Fich, M. Li, P. Ragde, and Y. Yesha, *Lower bounds for parallel random access machines with read only memory*, Inform. and Comput., 83 (1989), pp. 234–244.

[19] F. Fich, P. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[20] A. Gerbessiotis and C. Siniolakis, *Deterministic sorting and randomized median finding on the BSP model*, in Proceedings of the Eighth ACM Symposium on Parallel Algorithms and Architectures, Padua, Italy, 1996, pp. 223–232.

[21] A. Gerbessiotis and L. Valiant, *Direct bulk-synchronous algorithms*, J. Parallel Distributed Comput., 22 (1994), pp. 251–267.

[22] P. Gibbons, Y. Matias, and V. Ramachandran, *The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms*, SIAM J. Comput., 28 (1999), pp. 734–770.

[23] P. Gibbons, Y. Matias, and V. Ramachandran, *Efficient low-contention parallel algorithms*, in Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures, Cape May, NJ, 1994, pp. 236–247.

[24] M. Goodrich, *Communication-efficient parallel sorting*, in Proceedings of the 28th Annual ACM Symposium on Theory of Computing, Philadelphia, PA, 1996, pp. 247–256.

[25] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier, Amsterdam, The Netherlands, 1990, pp. 869–941.

[26] T. Leighton, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput., c-34 (1985), pp. 344–354.

[27] T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan-Kaufmann, San Francisco, San Mateo, CA, 1992.

[28] M. Li and Y. Yesha, *Separation and lower bounds for ROM and non-deterministic models of parallel computation*, in Proceedings of the 18th ACM Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 177–187.

[29] P. Mackenzie, *Lower bounds for randomized exclusive write PRAMs*, in Proceedings of the Seventh ACM Symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, 1995, pp. 254–263.

[30] Y. Mansour, N. Nisan, and U. Vishkin, *Trade-offs between communication throughput and parallel time*, in Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, 1994, pp. 372–381.

[31] K. Mehlhorn and U. Vishkin, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories*, Acta Inform., 21 (1984) pp. 339–374.

[32] C. Thompson, *A Complexity Theory for VLSI*, Ph.D. thesis., Carnegie-Mellon University, Pittsburgh, PA, 1980.

[33] L. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33(8) (1990), pp. 103–111.

[34] L. Valiant, *General purpose parallel architectures*, Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier, Amsterdam, The Netherlands, 1990, pp. 943–971.

[35] U. Vishkin and A. Wigderson, *Trade-offs between depth and width in parallel computation*, SIAM J. Comput., 14 (1985), pp. 303–314.

[36] A. Yao, *Probabilistic computations: Toward a unified measure of complexity*, in Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, Providence, RI, 1977, pp. 222–227.

SIAM J. Comput.
Vol. 29, No. 6, pp. 2016–2039

CONSTRUCTING PLANAR CUTTINGS IN THEORY AND PRACTICE[*]

SARIEL HAR-PELED[†]

Abstract. We present several variants of a new randomized incremental algorithm for computing a cutting in an arrangement of n lines in the plane. The algorithms produce cuttings whose expected size is $O(r^2)$, and the expected running time of the algorithms is $O(nr)$. Both bounds are asymptotically optimal for nondegenerate arrangements. The algorithms are also simple to implement, and we present empirical results showing that they perform well in practice. We also present another efficient algorithm (with slightly worse time bound) that generates small cuttings whose size is guaranteed to be close to the best known upper bound of J. Matoušek [*Discrete Comput. Geom.*, 20 (1998), pp. 427–448].

Key words. cuttings, range-searching, computational geometry

AMS subject classifications. 65Y25, 68U05

PII. S0097539799350232

1. Introduction.

A natural approach for solving various problems in computational geometry is the divide-and-conquer paradigm. A typical application of this paradigm to problems involving a set $\hat{S}$ of n lines in the plane is to fix a parameter $r > 0$ and to partition the plane into regions $R_1, \ldots, R_m$ (those regions are usually vertical trapezoids, or triangles, but we will consider here also convex polygons with more edges) such that the number of lines of $\hat{S}$ that intersect the interior of R_i is at most n/r for any $i = 1, \ldots, m$ (see Figure 1). This allows us to split the problem at hand into subproblems, each involving the subset of lines intersecting a region R_i. Such a partition is called a $(1/r)$-*cutting* of the plane. See [Aga91] for a survey of algorithms that use cuttings. For further work related to cuttings, see [AM95].

The first (though not optimal) construction of cuttings is due to Clarkson [Cla87]. Chazelle and Friedman [CF90] showed the existence of $(1/r)$-cuttings with $m = O(r^2)$ (a bound that is worst-case tight). They also showed that such cuttings, consisting of vertical trapezoids, can be computed in $O(nr)$ time. An optimal deterministic algorithm for generating cuttings was given by [Cha93]. Although those constructions are asymptotically optimal, they do not seem to produce a practically small number of regions. Coming up with a really small number of regions (i.e., reducing the constant of proportionality) is important for the efficiency of (recursive) data structures and algorithms that use cuttings. Currently, the best lower bound on the number of vertical trapezoids in a $(1/r)$-cutting in an arrangement of lines is $2.54(1 - o(1))r^2$, and the optimal cutting has at most $8r^2 + 6r + 4$ trapezoids; see [Mat98]. Improving the upper and lower bounds on the size of cuttings is still open, indicating that our understanding of cuttings is still far from being fully satisfactory. In section 3, we outline Matoušek's construction for achieving the upper bound and show a slightly improved construction (see below for details).

[*]Received by the editors January 13, 1999; accepted for publication August 27, 1999; published electronically April 18, 2000. A preliminary version of this paper appeared in the 14th ACM Symposium of Computational Geometry, Minneapolis, MN, 1998. This work was supported by a grant from the U.S.–Israeli Binational Science Foundation. This work was part of the author's Ph.D. thesis, prepared at Tel-Aviv University under the supervision of Prof. Micha Sharir.

http://www.siam.org/journals/sicomp/29-6/35023.html

[†]School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel (sariel@math.tau.ac.il).

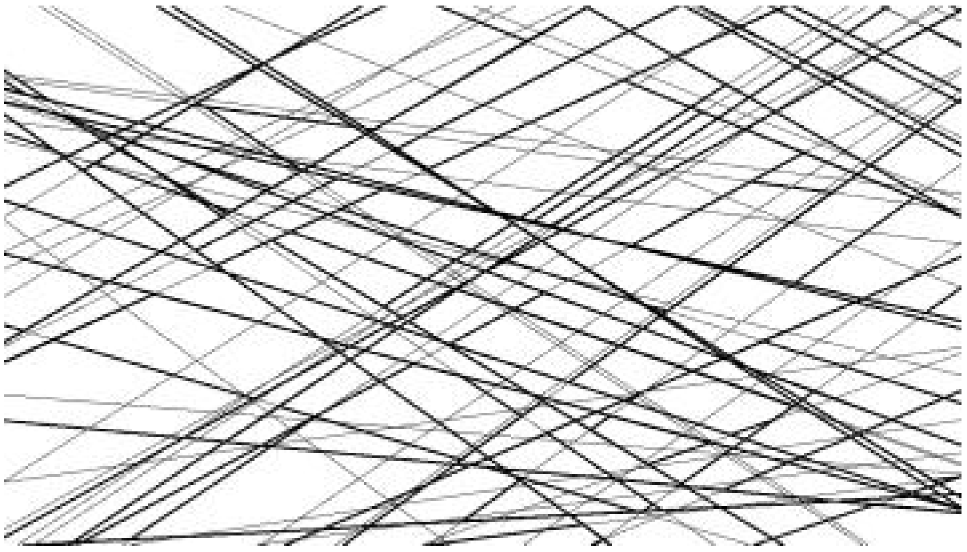


FIG. 1. $(1/20)$-*cutting of* 100 *lines as computed by our demo program* [HP98]*, using the* `PolyTree` *algorithm* (*see section* 4.2). *The boundaries of the cutting regions are marked by thick lines, and each such region intersects at most* 5 *lines in its interior.*

In this paper we propose several variants of a new and simple randomized incremental algorithm `CutRandomInc` for constructing cuttings and prove the expected worst-case tight performance bounds, as stated in the abstract, for `CutRandomInc` and for some of its variants. We also present empirical results on several algorithms/heuristics for computing cuttings that we have implemented. They are mostly variants of our new algorithm, and they all perform well in practice.[1] As already stated, $O(r^2)$ bounds on the expected size of the cuttings for some of those variants can be proved. For the other improved algorithms, no formal proof of performance is currently available, and we leave this as an open question for further research.

Matoušek [Mat98] gave an alternative construction for cuttings, showing that there exists a $(1/r)$-cutting with at most (roughly) $8r^2$ vertical trapezoids. Unfortunately, this construction relies on computing the whole arrangement, and its computation thus takes $O(n^2)$ time. We present a new randomized algorithm that is based on Matoušek's construction; it generates a $(1/r)$-cutting of size $\leq (1+\varepsilon)8r^2$, in $O\left(\frac{nr}{\varepsilon}\log^2 n\right)$ expected time, where $0 < \varepsilon \leq 1$ is any prescribed constant.

In section 2, we present the main two variants of the new algorithm `CutRandomInc` and analyze their expected running time and the expected number of trapezoids that they produce. Specifically, the expected running time is $O(nr)$ and the expected size of the output cutting is $O(r^2)$. We also analyze, in section 2.1, another variant `CRIVPolygon` of the algorithm that also has similar performance bounds. In section 3, we present and analyze our variant of Matoušek's construction. In section 4, we present our empirical results, comparing the new algorithms with several other algorithms/heuristics for constructing cuttings. These algorithms are mostly also variants of `CutRandomInc` (except that we still do not have a formal analysis of their performance bound), but they also include a variant of the older algorithm of Chazelle

[1]In spite of the theoretical importance of cuttings (in the plane and in higher dimensions), we are not aware of any (other) implementation of efficient algorithms for constructing cuttings.

and Friedman. The first batch of the implemented algorithms generate cuttings that consist of vertical trapezoids. Our empirical results show that the cuttings generated by the new algorithm `CutRandomInc` and its variants have between $10r^2$ and $14r^2$ vertical trapezoids. (The algorithms generate smaller cuttings when r is small. For example, for $r = 2$ the constant is about 9.) In contrast, the Chazelle–Friedman algorithm generates cuttings of size roughly $70r^2$. Some variants of our algorithm are based on cuttings by convex polygons with a small number of edges rather than by vertical trapezoids. These perform even better in practice, and we have a proof of optimality for one of the methods `CRIVPolygon`, which can be interpreted as an extension of `CutRandomInc` (see section 2.1). We conclude in section 5 by mentioning a few open problems. A program with a graphical user interface (GUI) demonstrating the algorithms and heuristics presented in the paper is available on the web in source form [HP98].

2. Incremental randomized construction of cuttings.

Given a set $\hat{S}$ of n lines in the plane, let $\mathcal{A}(\hat{S})$ denote the arrangement of $\hat{S}$, namely, the partition of the plane into faces, edges, and vertices as induced by the lines of $\hat{S}$ [Ede87]. Let $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ denote the partition of the plane into vertical trapezoids (i.e., the vertical decomposition of $\mathcal{A}(\hat{S})$), obtained by erecting two vertical segments up and down from each vertex of $\mathcal{A}(\hat{S})$ and extending each of them until it either reaches a line of $\hat{S}$, or otherwise all the way to infinity.

Computing the decomposed arrangement $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ can be done as follows. Pick a random permutation $S = \langle s_1, \dots, s_n \rangle$ of $\hat{S}$. Put $S_i = \langle s_1, \dots, s_i \rangle$ for $i = 1, \dots, n$. We compute incrementally the decomposed arrangements $\mathcal{A}_{\mathcal{VD}}(S_i)$ for $i = 1, \dots, n$ by inserting the ith line s_i of S into $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$. To do so, we compute the *zone* Z_i of s_i in $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$, which is the set of all trapezoids in $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$ that intersect s_i. We split each trapezoid of Z_i into at most four trapezoids, such that no trapezoid intersects s_i in its interior, as in [SA95]. Finally, we perform a pass over all the newly created trapezoids, merging vertical trapezoids that are adjacent and have identical top and bottom lines. The merging step guarantees that the resulting decomposition is $\mathcal{A}_{\mathcal{VD}}(S_i)$, independently of the insertion order of elements in S_i; see [dBvKOS97].

However, if we decide to skip the merging step, the resulting structure, denoted as $\mathcal{A}^{|}(S_i)$, depends on the order in which the lines are inserted into the arrangement. In fact, $\mathcal{A}^{|}(S_i)$ is $\mathcal{A}_{\mathcal{VD}}(S_i)$ with additional superfluous vertical walls. Each such vertical wall is a fragment of a vertical wall that was created at an earlier stage and got split during a later insertion step.

DEFINITION 2.1. *Let $\hat{S}$ be a set of n lines in the plane, and let $0 < c < 1$ be a constant. A* c-cutting *of $\hat{S}$ is a partition of the plane into regions $R_1, \dots, R_m$ such that, for each $i = 1, \dots, m$, the number of lines of $\hat{S}$ that intersect the interior of R_i is at most cn.*

A region C in the plane is c-active *if the number of lines of $\hat{S}$ that intersect the interior of C is larger than cn.*

A $(1/r)$-*cutting* is thus a partition of the plane into m regions such that none of them is $(1/r)$-active. Chazelle and Friedman [CF90] showed that one can compute, in $O(nr)$ time, a $(1/r)$-cutting that consists of $O(r^2)$ vertical trapezoids. Both bounds are asymptotically tight in the worst case.

We propose a new algorithm for computing a cutting that works by incrementally computing the arrangements $\mathcal{A}^{|}(S_i)$, using a random insertion order S of the lines. The new idea in the algorithm is that any "light" trapezoid (i.e., a trapezoid that is not $(1/r)$-active) constructed by the algorithm is immediately added to the final

```
ALGORITHM  CutRandomInc($\hat{S}$, $r$, $merge\text{–}flag$)
    Input:  A set $\hat{S}$ of $n$ lines, a positive integer $r$, and
            a flag $merge\text{–}flag$ that indicates whether merging is used or not
    Output: A $(1/r)$-cutting of $\hat{S}$ by vertical trapezoids
begin
    Choose a random permutation $S = \langle s_1, s_2, \ldots, s_n \rangle$ of $\hat{S}$.
    $\mathcal{C}_0 \leftarrow \{\mathbb{R}^2\}$.
    $i \leftarrow 0$.
    while there are $(1/r)$-active trapezoids in $\mathcal{C}_i$ do
        $i \leftarrow i + 1$
        $Zone_i \leftarrow$ The set of $(1/r)$-active trapezoids in $\mathcal{C}_{i-1}$ that intersect $s_i$.
        $Zone'_i \leftarrow \cup_{\Delta \in Zone_i} split(\Delta, s_i)$,
            where $split(\Delta, s)$ is the operation of splitting a vertical trapezoid $\Delta$
            crossed by a line $s$ into at most four vertical trapezoids, as in
            [dBvKOS97], such that the new trapezoids cover $\Delta$, and they do not
            intersect $s$ in their interior.
        if $merge\text{–}flag$ then
            Merge adjacent trapezoids in $Zone'_i$ that have the same top
                and bottom lines (one of which is $s_i$).
        end if
        $\mathcal{C}_i \leftarrow (\mathcal{C}_{i-1} \setminus Zone_i) \cup Zone'_i$.
    end while

    return $\mathcal{C}_i$
end CutRandomInc
```

FIG. 2. *Algorithm for constructing a $(1/r)$-cutting of an arrangement of lines.*

cutting, and the algorithm does not maintain the arrangement inside such a trapezoid from this point on. In this sense, one can think of the algorithm as being greedy; that is, it adds a trapezoid to the cutting as soon as one is constructed and proceeds in this manner until the whole plane is covered. The algorithm, called `CutRandomInc`, is depicted in Figure 2.

The algorithm has two variants. One does not merge adjacent trapezoids (as in the construction of $\mathcal{A}^{|}(S)$), while the other performs such mergings (as in the construction of $\mathcal{A}_{\mathcal{VD}}(S)$).

If `CutRandomInc` outputs $\mathcal{C}_k$ for some $k < n$, then $\mathcal{C}_k$ has no $(1/r)$-active trapezoids, and it is thus a $(1/r)$-cutting. After the ith line is inserted, it is guaranteed that no active trapezoid of $\mathcal{C}_j$ intersects s_i in its interior for $j \geq i$. This remains true even if merging is done by `CutRandomInc`. In particular, $\mathcal{C}_n$ has no active region, and it is a cutting.

To appreciate the following proof of correctness and optimality of `CutRandomInc`, one has to observe that the covering $\mathcal{C}_i$ of the plane maintained by `CutRandomInc` depends heavily on the order in which the lines are inserted into the arrangement. Indeed, the set of active trapezoids maintained by `CutRandomInc` falls outside the classical frameworks of Clarkson and Shor [CS89], lazy randomized incremental construction [dBDS95], and epsilon nets [HW87]. See Figures 3 and 4 for situations that illustrate the difference between these frameworks and ours. In order to analyze our

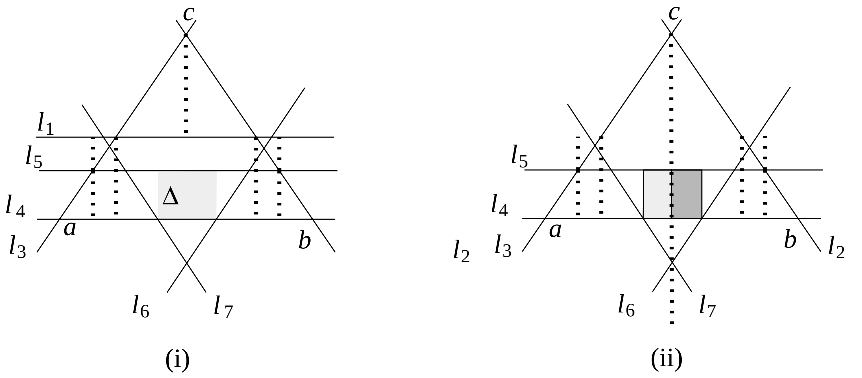


FIG. 3. *If merging is* not *used by* `CutRandomInc`*, an active trapezoid $\Delta \in \mathcal{C}_i$ might disappear if we skip an insertion of a line which does not belong to the defining or crossing sets of Δ. Indeed, if* `CutRandomInc` *inserts the lines in the order $l_1, l_2, \ldots, l_7$, then the trapezoid Δ is created; see* (i)*. However, if we skip the insertion of the line l_1, then the trapezoid Δ is not created, because the ray emanating downward from $l_2 \cap l_3$ intersects it. This implies that there is no "locality" in the determination of which trapezoids arise in the execution of* `CutRandomInc`*, so the standard techniques of* [CS89, dBDS95, HW87] *cannot be applied directly in analyzing* `CutRandomInc`.

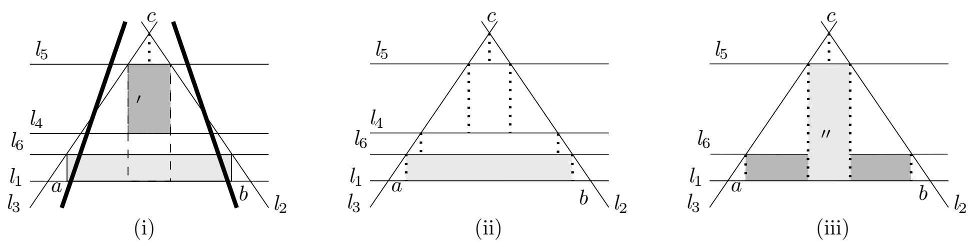


FIG. 4. *Even if merging is used by* `CutRandomInc`*, an active trapezoid $\Delta \in \mathcal{C}_i$ might disappear if we skip an insertion of a line which is completely unrelated to Δ. The thick lines represent two sets of* 100 *parallel lines, and we want to compute a* (1/10)*-cutting. We execute* `CutRandomInc` *with the first 6 lines $l_1, \ldots, l_6$ in this order. Note that any trapezoid that intersects a thick line is active. The first trapezoid Δ' inside $\triangle abc$ that becomes inactive is created when the line l_5 is being inserted; see parts* (i) *and* (ii)*. However, if we skip the insertion of the line l_4 (as in part* (iii)*), the corresponding inactive trapezoid Δ'' will extend downward and intersect Δ. Since Δ'' is inactive, the decomposition of the plane inside Δ'' is no longer maintained. In particular, this implies that the trapezoid Δ will not be created, since it is being blocked by Δ'', and no merging involving areas inside Δ'' will take place. Here too l_4 belongs neither to the defining nor to the crossing sets of Δ.*

algorithms, new techniques need to be developed.

In the following, we denote by R a *selection* of $\hat{S}$ of length $r \le n$, i.e., an ordered sequence of r distinct elements of $\hat{S}$. By a slight abuse of notation, we also denote by R the unordered set of its elements. We define the *weight* of a trapezoid to be the number of lines that cross its interior.

DEFINITION 2.2. Let $\mathcal{T} = \mathcal{T}(\hat{S})$ denote the set of all vertical trapezoids whose top and bottom edges are contained in lines of $\hat{S}$, and whose vertical sides are contained in lines that pass through vertices of $\mathcal{A}(\hat{S})$.

For a selection R of $\hat{S}$, let $\mathcal{CT}_{\mathcal{VD}}(R)$ denote the set of trapezoids of $\mathcal{A}_{\mathcal{VD}}(R)$. A trapezoid in $\mathcal{CT}_{\mathcal{VD}}(R)$ is defined by at most four lines. For an integer $0 \le k \le n$, let $\mathcal{CT}_{\mathcal{VD}}(R, k)$ denote the trapezoids of $\mathcal{CT}_{\mathcal{VD}}(R)$ having weight at least k.

DEFINITION 2.3. A vertical segment that serves as a left or right side of a trapezoid in $\mathcal{T}(\hat{S})$ is called a *splitter*. The *weight* of a splitter is the number of lines of $\hat{S}$

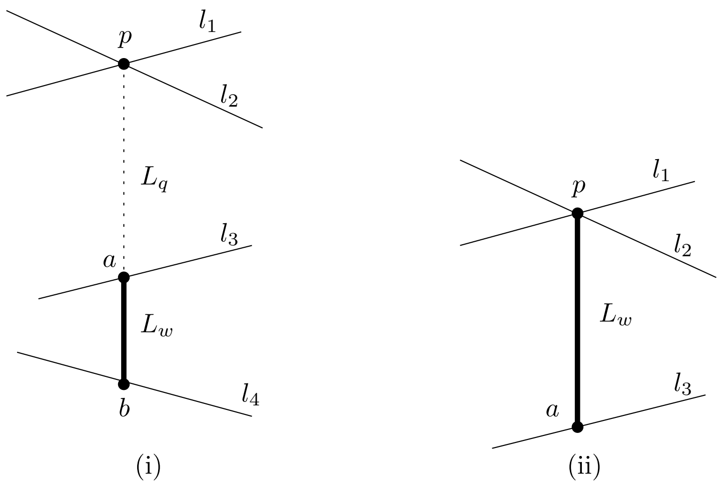


FIG. 5. *Splitters created by* `CutRandomInc`.

that cross the relative interior of the splitter. For a selection R of $\hat{S}$, let $\mathcal{CT}_{\mathcal{SP}}(R)$ denote the set of splitters of the trapezoids of $\mathcal{A}^{|}(R)$, and let $\mathcal{CT}_{\mathcal{SP}}(R,k)$ denote the set of splitters in $\mathcal{CT}_{\mathcal{SP}}(R)$ of weight at least k, where $0 \leq k \leq n$. In general, a splitter in $\mathcal{CT}_{\mathcal{SP}}(R)$ is uniquely defined by four lines: two define the vertex of the arrangement through which the vertical line containing the splitter passes, and two define (pass through) its top and bottom endpoints; see Figure 5(i). There are also splitters that are adjacent to the vertex that induces them (see Figure 5(ii)); these are defined uniquely by three lines.

In the following, S denotes the random permutation of $\hat{S}$ used by `CutRandomInc`.

LEMMA 2.4. *Let s be a splitter induced by $\{l_1, l_2, l_3, l_4\} \subseteq \hat{S}$, so that l_1 and l_2 intersect at a vertex p; s is contained in the vertical line l passing through p; and the endpoints of s are $a = l_3 \cap l$, $b = l_4 \cap l$, with a nearer to p than b. Let L_w be the set of lines of $\hat{S}$ that intersect the relative interior of s, and let L_q be the set of lines of $\hat{S}$ that intersect the relative interior of ap; see Figure* 5(i)*. Then the probability of s to be created by* `CutRandomInc` *is bounded by*

$$\frac{8}{(w+1)^2(q+w+3)^2},$$

where $w = |L_w|, q = |L_q|$.

If one of the endpoints of the splitter is p, then the probability of s to be created is $\leq 6/(w+1)^3$, where $w = |L_w|$ and L_w is the set of lines crossing the relative interior of s; see Figure 5(ii).

Proof. Let A denote the event that l_1, l_2 appear in S before all the lines of $L_q \cup L_w \cup \{l_3\}$, and let B denote the event that l_3, l_4 appear in S before all the lines of L_w.

The event A is a necessary and sufficient condition for pb (or a longer segment) to be created when merging is not used. The event B, conditioned on A, is a necessary and sufficient condition for the vertices delimiting s to be created before s is being "killed." Hence s is created by `CutRandomInc` (without merging) if and only if $A \cap B$ occurs for S. (If merging is used, only one implication holds: If s is created, then $A \cap B$ occurs for S.)

To compute $P(A \cap B)$ it suffices to consider permutations of only the $q+w+4$ lines in $L_q \cup L_w \cup \{l_1, l_2, l_3, l_4\}$. We distinguish between two cases.

(i) The first three lines in such a permutation are the lines l_1, l_2, l_4, in any order in which l_4 is not the third line. This ensures the occurrence of A. We now choose the $w+1$ locations of the lines in $L_w \cup \{l_3\}$ in the permutation and place l_3 at the first of the these locations, thus ensuring B. The number of such permutations is $(6-2)\binom{q+w+1}{w+1} w!q! = 4(q+w+1)!/(w+1)$.

(ii) The first two lines in the permutation are l_1, l_2 (in any order). Again, A is ensured. To ensure B, we choose the $w+2$ locations of the lines in $L_w \cup \{l_3, l_4\}$ and place l_3, l_4 as the first two of them (in any order). The number of such permutations is

$$2!\binom{q+w+2}{w+2} 2!w!q! = \frac{4(q+w+2)!}{(w+1)(w+2)}.$$

It is easily verified that these two cases exhaust all possibilities of $A \cap B$ to arise. Hence,

$$\begin{aligned} P(A \cap B) &= \frac{4(q+w+1)!}{(w+1)(q+w+4)!} + \frac{4(q+w+2)!}{(w+1)(w+2)(q+w+4)!} \\ &\le \frac{4}{(w+1)(q+w+2)(q+w+3)^2} + \frac{4}{(w+1)^2(q+w+3)^2} \\ &\le \frac{8}{(w+1)^2(q+w+3)^2}. \end{aligned}$$

The proof of the second part of the lemma follows by observing that ap is created if and only if l_1, l_2, l_3 appear in S before all the lines of L_w. The probability for this to happen is $3!w!/(w+3)! \le 6/(w+1)^3$. ☐

DEFINITION 2.5. Let $\mathcal{T}_{\mathcal{SP}}(S)$ denote the set of splitters in $\bigcup_{i=1}^{n} \mathcal{CT}_{\mathcal{SP}}(S_i)$, and let $\mathcal{T}_{\mathcal{VD}}(S)$ denote the set of trapezoids in $\bigcup_{i=1}^{n} \mathcal{CT}_{\mathcal{VD}}(S_i)$. Let $\mathcal{T}_{\mathcal{SP}}^{\mathcal{A}}(S) = \bigcup_{i=1}^{n} \mathcal{CT}_{\mathcal{SP}}(S_i, n/(2r))$, and let $\mathcal{T}_{\mathcal{VD}}^{\mathcal{A}}(S) = \bigcup_{i=1}^{n} \mathcal{CT}_{\mathcal{VD}}(S_i, n/r)$.

LEMMA 2.6. *Let S be a random permutation of $\hat{S}$. Then*

$$E\left[\sum_{s \in \mathcal{T}_{\mathcal{SP}}^{\mathcal{A}}(S)} (w(s))^c\right] = O\left(n^c r^{2-c}\right)$$

for $c = 0$ or $c = 1$.

Proof. Let p be a vertex of $\mathcal{A}(\hat{S})$. The expected contribution of all the splitters that lie on the vertical line passing through p to the above sum is at most

$$\begin{aligned} &O\left(\sum_{w=n/2r}^{\infty} \sum_{q=0}^{\infty} \frac{w^c}{(w+1)^2(q+w+3)^2} + \sum_{w=n/2r}^{\infty} \frac{w^c}{(w+1)^3}\right) \\ &= O\left(\sum_{w=n/2r}^{\infty} \sum_{q=0}^{\infty} \frac{w^{c-2}}{(q+w+3)^2} + \left(\frac{n}{r}\right)^{c-2}\right) = O\left(\sum_{w=n/2r}^{\infty} w^{c-3} + \left(\frac{n}{r}\right)^{c-2}\right) \\ &= O\left(\left(\frac{n}{r}\right)^{c-2}\right) \end{aligned}$$

for $c = 0, 1$.

Since there are $O(n^2)$ vertices in $\mathcal{A}(\hat{S})$, it follows that

$$E\left[\sum_{s\in\mathcal{T}^{\mathcal{A}}_{\mathcal{SP}}(S)} (w(s))^c\right] = O\left(n^c r^{2-c}\right). \qquad \square$$

Lemma 2.6 implies that the number of "heavy" splitters generated by `CutRandomInc`, with or without merging, is $O(r^2)$, and their total weight is $O(nr)$.

LEMMA 2.7. *Let S be a random permutation of $\hat{S}$. Then*

$$W = E\left[\sum_{\Delta\in\mathcal{T}^{\mathcal{A}}_{\mathcal{VD}}(S)} (w(\Delta))^c\right] = O\left(n^c r^{2-c}\right)$$

for $c = 0, 1$.

Proof. The probability of a trapezoid Δ of weight w to be created during the computation of $\mathcal{CT}_{\mathcal{VD}}(S)$, if it is defined by $d \le 4$ lines, is proportional to $1/w^d$. Let f^d_w denote the number of trapezoids of $\mathcal{T}_{\mathcal{VD}}(S)$ that are defined by d lines of S and have weight w. Let $F^d_{\le w} = \sum_{q=0}^{w} f^d_q$. By the Clarkson–Shor probabilistic technique [CS89], we have $F^d_{\le w} = O((n/w)^2 w^d) = O(n^2 w^{d-2})$. Let W_d denote the contribution to W made by trapezoids defined by d lines. Then

$$\begin{aligned}
W_d &= \sum_{w=n/r}^{n-d} \frac{f^d_w w^c}{w^d} = \sum_{w=n/r}^{n-d} f^d_w w^{c-d} \le \sum_{w=n/r}^{n-d} (F^d_{\le w} - F^d_{\le w-1}) w^{c-d} \\
&\le F^d_{\le n} n^{c-d} + \sum_{w=n/r}^{n-d-1} F^d_{\le w}(w^{c-d} - (w+1)^{c-d}) = O\left(n^c + \sum_{w=n/r}^{n-d-1} F^d_{\le w} w^{c-d-1}\right) \\
&= O\left(n^c + \sum_{w=n/r}^{n-d-1} n^2 w^{d-2} w^{c-d-1}\right) = O\left(n^c + n^2 \sum_{w=n/r}^{\infty} w^{c-3}\right) \\
&= O\left(n^c + n^2 (n/r)^{c-2}\right) = O\left(n^c r^{2-c}\right).
\end{aligned}$$

Overall, $W = \sum_{d=1}^{4} W_i = O(n^c r^{2-c})$. $\square$

By Lemma 2.7, the expected number of trapezoids in $\mathcal{T}^{\mathcal{A}}_{\mathcal{VD}}(S)$ is $O(r^2)$, and their expected total weight is $O(nr)$.

Remark 2.8. Lemmas 2.6 and 2.7 hold for any $0 \le c < 2$, but we only need the results for $c = 0, 1$.

Let $\nabla\mathcal{VD}_i = \mathcal{CT}_{\mathcal{VD}}(S_i, n/r) \setminus \mathcal{CT}_{\mathcal{VD}}(S_{i-1}, n/r)$, let $\nabla\mathcal{SP}_i = \mathcal{CT}_{\mathcal{SP}}(S_i, n/2r) \setminus \mathcal{CT}_{\mathcal{SP}}(S_{i-1}, n/2r)$, and let $\nabla\mathcal{C}_i$ be the set of *active* trapezoids in $\mathcal{C}_i \setminus \mathcal{C}_{i-1}$, namely, the new active trapezoids created in the ith iteration of the algorithm.

LEMMA 2.9. (a) *Each new trapezoid $\tau \in \nabla\mathcal{C}_i$ must be contained in a new trapezoid $\Delta \in \nabla\mathcal{VD}_i$.*

(b) *Let Δ be a $(1/r)$-active trapezoid in $\nabla\mathcal{VD}_i$, and let $\nabla\mathcal{C}_i(\Delta)$ (resp., $\nabla\mathcal{SP}_i(\Delta)$) denote the set of trapezoids in $\nabla\mathcal{C}_i$ (resp., splitters in $\nabla\mathcal{SP}_i$) that are contained in Δ. Then*

$$\sum_{\tau\in\nabla\mathcal{C}_i(\Delta)} w(\tau) = O\left(w(\Delta) + \sum_{s\in\nabla\mathcal{SP}_i(\Delta)} w(s)\right)$$

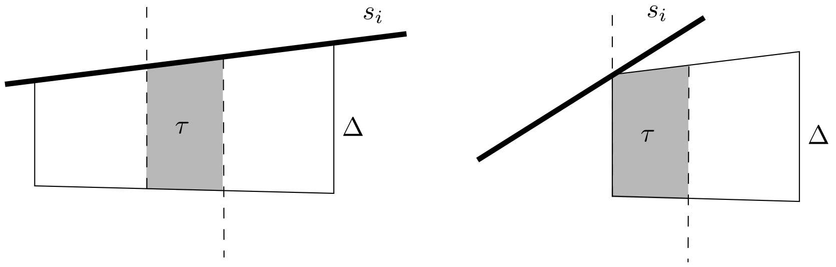


FIG. 6. *A new trapezoid $\tau \in \nabla\mathcal{C}_i$ must lie inside a new trapezoid $\Delta \in \nabla\mathcal{VD}_i$.*

and

$$\left|\nabla\mathcal{C}_i(\Delta)\right| = O\left(\frac{r}{n}w(\Delta) + \left|\nabla\mathcal{SP}_i(\Delta)\right|\right).$$

The lemma holds regardless of whether or not `CutRandomInc` *performs merging.*

Proof. (a) Let $\tau \in \nabla\mathcal{C}_i$ be an active trapezoid created in the ith iteration of `CutRandomInc`, and let Δ be the trapezoid of $\mathcal{CT}_{\mathcal{VD}}(S_i)$ that contains Δ. If the line s_i is the top or bottom line of τ, then clearly Δ is also newly created. Otherwise s_i must delimit one of the vertical sides of τ. This side is also a side of Δ, so Δ is newly created. Hence $\Delta \in \nabla\mathcal{VD}_i$. See Figure 6.

(b) Let τ be a trapezoid in $\nabla\mathcal{C}_i(\Delta)$. We charge the weight of τ either to (a portion of) the weight of the (active) trapezoid of Δ, or to a new "heavy" splitter that bounds τ.

As noted in (a), at least one of the splitters of τ must be new (i.e., created in the ith iteration), and this remains true even if τ was created by merging a few active trapezoids of $\mathcal{C}_{i-1}$.

Consider the lines $l \in \hat{S}$ that cross τ. If at least half of these lines intersect the ceiling and/or floor of Δ, we charge $w(\tau)$ to those intersection points whose number is at least $w(\tau)/2$. Since there are at most $2w(\Delta)$ such intersections on the boundary of Δ, it follows that the sum of the weights $w(\tau)$ of such trapezoids is at most $4w(\Delta)$.

So one can assume that at least half the lines that cross τ do not intersect either the floor or the ceiling of τ. This implies that the new splitter must intersect at least $w(\tau)/2 > n/(2r)$ of these lines, which implies that $s \in \nabla\mathcal{SP}_i$. Thus, we charge $w(\tau)$ to $w(s)$. Since $w(\tau) \leq 2w(s)$, and each such splitter can be charged at most twice, the first inequality of (b) follows.

As for the second inequality, if a $(1/r)$-active trapezoid $\tau \in \nabla\mathcal{C}_i(\Delta)$ does not have a splitter of $\nabla\mathcal{SP}_i$ as one of its sides, then there at least n/r intersections between the lines crossing τ and the bottom and top edges of τ. The number of such trapezoids within Δ is at most $2w(\Delta)/(n/r) = O((r/n)w(\Delta))$. This is easily seen to imply the second inequality. ∎

THEOREM 2.10. *The expected size of the cutting generated by* `CutRandomInc` *(with or without merging) is $O(r^2)$ and the expected running time is $O(nr)$.*

Proof. Since the direct work involved in creating the children of a trapezoid is proportional to the number of lines that cross it, we can bound the overall work performed by `CutRandomInc` by the total weight of the active trapezoids that it generates.

Of course, if we perform merging, there is also additional work associated with merging trapezoids. However, the merging stage can be performed in linear time in the total weight of the split trapezoids. This requires a somewhat careful but routine implementation, so that the running time remains linear even when merging the conflict lists of a potentially large number of trapezoids, during the creation of a single new (merged) trapezoid.

Thus, using Lemma 2.9(a), the expected running time is proportional to

$$E\left[\sum_{i=1}^{n}\sum_{\tau\in\nabla\mathcal{C}_i} w(\tau)\right] = E\left[\sum_{i=1}^{n}\sum_{\Delta\in\nabla\mathcal{VD}_i}\sum_{\tau\in\nabla\mathcal{C}_i(\Delta)} w(\tau)\right].$$

By Lemma 2.9(b), we have

$$\begin{aligned} E\left[\sum_{i=1}^{n}\sum_{\tau\in\nabla\mathcal{C}_i} w(\tau)\right] &= E\left[\sum_{i=1}^{n}\sum_{\Delta\in\nabla\mathcal{VD}_i} O\left(w(\Delta)+\sum_{s\in\nabla\mathcal{SP}_i(\Delta)} w(s)\right)\right] \\ &= O\left(E\left[\sum_{\Delta\in\mathcal{T}^{\mathcal{A}}_{\mathcal{VD}}(S)} w(\Delta)+\sum_{s\in\mathcal{T}^{\mathcal{A}}_{\mathcal{SP}}(S)} w(s)\right]\right) = O(nr) \end{aligned}$$

by Lemmas 2.6, 2.7.

As for the expected size of the cutting,

$$\begin{aligned} E\left[\sum_{i=1}^{n}|\nabla\mathcal{C}_i|\right] &= E\left[\sum_{i=1}^{n}\sum_{\Delta\in\nabla\mathcal{VD}_i}|\nabla\mathcal{C}_i(\Delta)|\right] = O\left(E\left[\sum_{\Delta\in\mathcal{T}^{\mathcal{A}}_{\mathcal{VD}}(S)}\frac{r}{n}w(\Delta)+|\mathcal{T}^{\mathcal{A}}_{\mathcal{SP}}(S)|\right]\right) \\ &= O(r^2), \end{aligned}$$

by Lemma 2.9(b). □

The algorithm `CutRandomInc` works also for planar arrangements of segments and x-monotone curves (such that the number of intersections of any pair of curves is bounded by a constant). This follows by a straightforward adaption of the proof to those cases, which we omit, and it is summarized in the following proposition.

PROPOSITION 2.11. *Let $\hat{\Gamma}$ be a set of x-monotone curves such that each pair intersects in at most a constant number of points. Then the expected size of the $(1/r)$-cutting generated by* `CutRandomInc` *for Γ is $O(r^2)$, and the expected running time is $O(nr)$ for any integer $1 \le r \le n$ in an appropriate model of computation.*[2]

However, the arrangement of a set of n segments or curves might have subquadratic complexity (since the number of intersection points might be subquadratic). This raises the question of whether `CutRandomInc` generates smaller cuttings for such sparse arrangements.

Indeed, an algorithm of [dBS95] generates cuttings of size $O(r+\frac{r^2}{n^2}\kappa)$ in expected time $O\left(n\log r+\frac{r}{n}\kappa\right)$, where κ is the overall complexity of the arrangement. Using `CutRandomInc` for this case, we obtain the following slightly weaker bounds.

PROPOSITION 2.12. *Let $\hat{\Gamma}$ be a set of n curves, such that each pair of curves of $\hat{\Gamma}$ intersect in at most a constant number of points. Then the expectedsize of the*

[2]Namely, the real-RAM model—the intersection of two curves, and the value of a curve at a certain x-coordinate can be computed in $O(1)$ time.

$(1/r)$-cutting generated by `CutRandomInc`*, when applied to $\hat{\Gamma}$, is*

$$O\left(r \log r + \frac{r^2}{n^2}\kappa\right),$$

and the expected running time is $O(n \log^2 r + r\kappa/n)$ for any integer $1 \le r \le n$, where κ is the complexity of $\mathcal{A}(\hat{\Gamma})$.

Since `CutRandomInc` generates superficial splitters, the results for sparse arrangements are slightly worse (by a factor of $\log r$ if κ is relatively small) than those of [dBS95]. We omit any further details. The algorithm of [dBS95] is similar to the algorithm of Chazelle and Friedman [CF90], and we therefore believe that in practice `CutRandomInc` (with merging) will generate much smaller cuttings for $\mathcal{A}(\hat{\Gamma})$, as the results of section 4 might suggest.

Remark 2.13. An interesting question is whether `CutRandomInc` can be extended to higher dimensions. If we execute `CutRandomInc` in higher dimensions, we need to use a more complicated technique in decomposing each of our "vertical trapezoids" whenever it intersects a newly inserted hyperplane. Chazelle and Friedman's algorithm uses bottom vertex triangulation for this decomposition. However, in our case, it is easy to verify that `CutRandomInc` might generate simplices so that the size of their defining set need not be bounded by a constant, if we use bottom vertex triangulation. This implies that the current analysis cannot be extended to this case. We leave the problem of extending `CutRandomInc` to higher (say, three) dimensions as an open problem for further research.

2.1. Cuttings by vertical polygons. In this section, we present another variant of `CutRandomInc` that uses "vertical polygons" instead of vertical trapezoids and establish similar optimal performance bounds for this variant.

Definition 2.14. Let $\hat{S}$ be a set of n lines. A convex polygon P is a *μ-vertical polygon* of $\hat{S}$ if the boundary of P, except for the two vertical sides of P, if any, is contained in $\bigcup \hat{S}$, and the number of nonvertical edges of P is at most μ, where μ is a small positive constant (the case $\mu = 2$ is the case of vertical trapezoids). We denote the set of all such polygons by $\mathcal{VP}_\mu(\hat{S})$. A μ-vertical polygon P is a *μ-corridor* if its two splitters (i.e., vertical sides) are defined by vertices of $\mathcal{A}(\hat{S})$ lying on the boundary of P. Note that the size of the defining set of a μ-corridor is $\le \mu + 2$.

In the following, we consider μ to be a prescribed small constant.

We can use μ-vertical polygons instead of vertical trapezoids in `CutRandomInc`; namely, each region maintained by `CutRandomInc` is a μ-vertical polygon. Whenever a new region is being created, it is split into two subregions if it has more than μ nonvertical edges. This is done by erecting a splitter from an appropriate vertex of the region. Let `CRIVPolygon` denote this variant of `CutRandomInc`. Here too we have the option of performing *merging*. That is, we merge an as-long-as-possible sequence of adjacent μ-vertical polygons within the same face of $\mathcal{A}(S_i)$ into a single vertical polygon, as long as the number of its nonvertical edges does not exceed μ.

The gain in using μ-vertical polygons instead of vertical trapezoids is that the number of splitters generated is much smaller, yielding smaller cuttings. See section 4. The disadvantage is that the regions output by the algorithm are more complex to handle in any subsequent application of the cutting.

Lemma 2.15. *Let $\hat{S}$ be a set of n lines in the plane. The number of μ-corridors of $\mathcal{A}(\hat{S})$ is $O(n^2)$.*

Proof. A vertex p of $\mathcal{A}(\hat{S})$ is the left-bottom vertex of onlya constant number ($O(\mu)$) of μ-corridors. The lemma follows since the number of such vertices is $O(n^2)$. ∎

Note that any μ-vertical polygon is contained in only a constant number of μ-corridors. Let S be a random permutation of the lines of $\hat{S}$, and let $\mathcal{CO}_i$ denote the set of active μ-corridors in $\mathcal{A}(S_i)$ for $i = 1, \dots, n$. Let $\mathcal{T}^{\mathcal{A}}_{\mathcal{CO}} = \bigcup_{i=1}^{n} \mathcal{CO}_i$. Note that those corridors are not necessarily disjoint. Moreover, each active region maintained by `CRIVPolygon` is contained inside at least one active corridor, and the set of splitters generated by `CRIVPolygon` is a subset of the set of splitters generated by `CutRandomInc`. Thus, Lemma 2.6 holds for the active splitters generated by `CRIVPolygon`. As for the corridors, we have the following lemma.

LEMMA 2.16. *Let S be a random permutation of $\hat{S}$; then*

$$W = E\left[\sum_{\tau \in \mathcal{T}^{\mathcal{A}}_{\mathcal{CO}}(S)} (w(\tau))^c\right] = O\left(n^c r^{2-c}\right)$$

for $c = 0, 1$.

Proof. We follow the proof of Lemma 2.7, using Lemma 2.15 to bound the number of "heavy" μ-corridors. This follows by observing that we can apply the Clarkson–Shor technique [CS89] to bound the number of μ-corridors with weight at most k. Indeed, arguing as in the proof of Lemma 2.7, the expected contribution to total weight of μ-vertical corridors having weight at least $n/(2r)$, created by the algorithm and defined by d lines, is $O(n^c r^{2-c})$ for $d = 1, \dots, \mu + 4$. ∎

THEOREM 2.17. *The expected size of the cutting generated by* `CRIVPolygon` *(with or without merging) is $O(r^2)$, and the expected running time is $O(nr)$.*

Proof. We only sketch the proof, since it is similar to the analysis of `CutRandomInc`. Note that the proof of Lemma 2.9 can be adapted to handle active corridors and the active regions maintained by `CRIVPolygon`. Indeed, after the ith iteration of the algorithm, each newly created active μ-vertical polygon is contained in (at least one) newly created active μ-corridor. We assign each such μ-vertical polygon to one of those newly active μ-corridors in an arbitrary manner.

Now, to bound the work associated with such a μ-corridor X, we apply the same charging scheme used in Lemma 2.9, charging the weight of all the active μ-vertical polygons that were assigned to X (all of them are contained inside X) to the weight of X and to the weight of the newly created "heavy" splitters inside X.

Since each splitter is contained in a constant number of μ-corridors, this implies that the overall charge made to a newly created splitter s is $O(w(s))$.

Overall, this implies that the overall expected running time of the algorithm is bounded by the total weight of the active μ-corridors and the heavy splitters that it creates.

By Lemmas 2.6 and 2.16, the expected total weight of the μ-vertical polygons generated by `CRIVPolygon` is $O(nr)$, which implies immediately that the expected running time is $O(nr)$, and the expected size of the cuttings is $O(r^2)$, by following the proof of Theorem 2.10. We omit the details. ∎

Remark 2.18. We can further modify the algorithm `CRIVPolygon` so that it tries to remove inactive regions from the left and right sides of any newly created active region. We call this variant `PolyVertical`. It is easy to verify that the same proof of correctness, with slight modifications, works also for `PolyVertical`.

3. Generating small cuttings. In this section, we present an efficient algorithm that generates cuttings of guaranteed small size. The algorithm is based on

Matoušek's construction of small cuttings [Mat98]. We first review this construction and then show how to modify it for building small cuttings efficiently.

Definition 3.1 (see [Mat98]). Let $\hat{S}$ be a set of n lines in the plane in general position, i.e., every pair of lines intersects in exactly one point, no three have a common point, no line is vertical or horizontal, and the x-coordinates of all intersections are distinct. The *level* of a point in the plane is the number of lines of $\hat{S}$ lying strictly below it. Consider the set E_k of all edges of the arrangement of $\hat{S}$ having level k (where $0 \le k < n$). These edges form an x-monotone connected polygonal line, which is called the *level* k of the arrangement of $\hat{S}$.

Definition 3.2 (see [Mat98]). Let E_k be the level k in the arrangement $\mathcal{A}(\hat{S})$ with edges $e_0, e_1, \dots, e_t$ (from left to right), and let p_i be a point in the interior of the edge e_i, for $i = 0, \dots, t$. The *q-simplification* of the level k for an integer parameter $1 \le q \le t$ is defined as the x-monotone polygonal line containing the part of e_0 to the left of the point p_0, the segments $p_0 p_q$, $p_q p_{2q}, \dots, p_{\lfloor (t-1)/q \rfloor q} p_t$, and the part of e_t to the right of p_t. Let $\mathrm{simp}_q(E_k)$ denote this polygonal line.

Let $\hat{S}$ be a set of n lines in general position, and let $\mathcal{E}_{i,q}$ denote the union of the levels $E_i, E_{i+q}, \dots, E_{n+i-q}$ for $i = 0, \dots, q-1$. Let $\mathrm{simp}_q(\mathcal{E}_{i,q})$ denote the set of edges of the q-simplifications of the levels of $\mathcal{E}_{i,q}$.

Matoušek showed that the vertical decomposition of the plane induced by $\mathrm{simp}_q(\mathcal{E}_{i,q})$, where $q = n/(2r)$ (we assume that n is divisible by $2r$), is a $(1/r)$-cutting of the plane for any $i = 0, \dots, q-1$. Moreover, the following holds.

Theorem 3.3 (see [Mat98]). *Let $\hat{S}$ be a set of n lines in general position, let r be a positive integer, and let $q = n/(2r)$. Then the subdivision of the plane defined by the vertical decomposition of $\mathrm{simp}_q(\mathcal{E}_{m,q})$ is a $(1/r)$-cutting of $\mathcal{A}(\hat{S})$, where $0 \le m < q$ is the index $i \in \{0, \dots, q-1\}$ for which $|\mathcal{E}_{i,q}|$ is minimized. Moreover, the cutting generated has at most $8r^2 + 6r + 4$ trapezoids.*

Remark 3.4. (i) Matoušek's construction can be slightly improved, by noting that the leftmost and rightmost points in a q-simplification of a level can be placed at "infinity"; that is, we replace the first and second edges in the q-simplification by a ray emanating from p_q which is parallel to e_0. We perform a similar shortcut for the two last edges of the simplified level. We denote this improved simplification by simp'_q. It is easy to prove that using this improved simplification also results in a $(1/r)$-cutting of $\mathcal{A}(\hat{S})$ with at most $8r^2 + 6r + 4 - 4 \cdot 2r = 8r^2 - 2r + 4$ vertical trapezoids.

(ii) Inspecting Matoušek's construction, we see that if we can only find an i such that $|\mathcal{E}_{i,q}| \le cn^2/q$, where $c > 1$ is a prescribed constant, then the vertical decomposition induced by $\mathrm{simp}'_q(\mathcal{E}_{i,q})$ is a $(1/r)$-cutting having $\le c(8r^2 - 2r + 4)$ trapezoids.

Let $n_i = |\mathcal{E}_{i,q}|$ for $i = 0, \dots, q-1$. Matoušek's construction is carried out by computing the numbers $n_0, \dots, n_q$ and picking the minimal number n_i, which is guaranteed to be no larger than the average n^2/q. Unfortunately, implementing this scheme explicitly requires computing the whole arrangement $\mathcal{A}(\hat{S})$, so the resulting running time is $O(n^2)$. Let us assume for the moment that one can compute any of the numbers n_i quickly. Then, as the following lemma shows, one can compute a number n_i which is $\le (1+\varepsilon)n^2/q$, without computing all the n_i's.

Lemma 3.5. *Let $n_0, \dots, n_{q-1}$ be q positive integers, whose sum $m = \sum_{i=0}^{q-1} n_i$ is known in advance, and let $\varepsilon > 0$ be a prescribed constant. One can compute an index $0 \le k < q$, such that $n_k \le \lceil (1+\varepsilon)m/q \rceil$, by repeatedly picking uniformly and independently a random index $0 \le i < q$ and by checking whether $n_i \le \lceil (1+\varepsilon)m/q \rceil$. The expected number of iterations required is $\le 1 + 1/\varepsilon$.*

Proof. Let Y_i be the random variable which is the value of n_i picked in the ith iteration. Using Markov's inequality[3], one obtains

$$Pr\left[Y_i \geq (1+\varepsilon)\frac{m}{q}\right] \leq \frac{E\left[Y_i\right]}{(1+\varepsilon)\frac{m}{q}}.$$

Since $E[Y_i] = m/q$, we have that the probability for failure in the ith iteration is

$$Pr\left[Y_i \geq (1+\varepsilon)\frac{m}{q}\right] \leq \frac{1}{1+\varepsilon}.$$

Let X denote the number of iterations required by the algorithm. Then $E[X]$ is bounded by the expected number of trials to the first success in a geometric distribution with probability $p \geq 1 - \frac{1}{1+\varepsilon}$. Thus, the expected number of iterations is bounded by

$$E[X] \leq \frac{1}{p} \leq \frac{1}{1-\frac{1}{1+\varepsilon}} = 1 + \frac{1}{\varepsilon}. \qquad \square$$

To apply Lemma 3.5 in our setting, we need to supply an efficient algorithm for computing the level of an arrangement of lines in the plane.

LEMMA 3.6. *Let $\hat{S}$ be a set of n lines in the plane. Then one can compute, in $O((n+h)\log^2 n)$ time, the level k of $\mathcal{A}(\hat{S})$, where $h = |E_k|$ is the complexity of the level.*

Proof. The technique presented here is well known (see [BDH99] for a recent example); we include it for the sake of completeness of exposition. Let $e_0, \ldots, e_t$ be the edges of the level k from left to right (where e_0, e_t are rays).

Let e be an edge of the level k. Let f be the face of $\mathcal{A}(\hat{S})$ having e on its boundary and lying above e. In particular, all the edges on the bottom part of ∂f belong to the level k.

Let $f_1, \ldots, f_r$ be the faces of $\mathcal{A}(\hat{S})$ having the level k as their "floor," from left to right. The ray e_0 can be computed in $O(n)$ time since it lies on line l_k of $\hat{S}$, with the kth largest slope. Moreover, by intersecting l_k with the other lines of $\hat{S}$, one can compute e_0 in linear time.

Any face of $\mathcal{A}(\hat{S})$ is uniquely defined as an intersection of half-planes induced by the lines of $\hat{S}$. For the face f_1, we can compute the half-planes and their intersection that represents f_1, in $O(n \log n)$ time; see [dBvKOS97]. To carry out the computation of the bottom parts of $f_2, \ldots, f_r$, one can dynamically maintain the intersection representing f_i as we traverse the level k from left to right. To do so, we will use the data structure of Overmars and Van Leeuwen [OvL81] that maintains such an intersection, with $O(\log^2 n)$ time for each update operation. As we move from f_i to f_{i+1} through a vertex v, we have to "flip" the two half-planes associated with the two lines passing through v, at a cost of $O(\log^2 n)$ time per vertex. Similarly, if we are given an edge e on the boundary of f_i we can compute the next edge in $O(\log^2 n)$ time.

Thus, we can compute the level k of $\mathcal{A}(\hat{S})$ in $O((n+h)\log^2 n)$ time. $\square$

Combining Lemmas 3.5 and 3.6, we have the following theorem.

THEOREM 3.7. *Let $\hat{S}$ be a set of n lines in the plane, and let $0 < \varepsilon \leq 1$ be a prescribed constant. Then one can compute a $(1/r)$-cutting of $\mathcal{A}(\hat{S})$, having at*

[3]The inequality asserts that $Pr[Y \geq t] \leq \frac{E[Y]}{t}$ for a random variable Y that assumes only nonnegative values.

most $(1+\varepsilon)(8r^2 - 2r + 4)$ *trapezoids. The expected running time of the algorithm is* $O\left(\left(1+\frac{1}{\varepsilon}\right) nr \log^2 n\right)$.

Proof. By the above discussion, it is enough to find an index $0 \le i \le q-1$, such that $|\mathcal{E}_{i,q}| \le M = (1+\varepsilon)\frac{n^2}{q} \le 2(1+\varepsilon)nr$, where $q = \lceil n/(2r) \rceil$. By Remark 3.4(ii), the vertical decomposition of $\mathrm{simp'}_q(\mathcal{E}_{i,q})$ is a $(1/r)$-cutting of the required size.

Picking i randomly, we have to check whether $|\mathcal{E}_{i,q}| \le M$. We can compute $\mathcal{E}_{i,q}$ by computing the levels $E_i, E_{i+q}, \dots, E_{i+\lfloor (n-i-1)/q \rfloor q}$ in an output-sensitive manner, using Lemma 3.6. Note that if $|\mathcal{E}_{i,q}| > M$, we can abort as soon as the number of edges we computed exceeds M. Thus, we can check if $|\mathcal{E}_{i,q}| \le M$ takes $O(nr \log^2 n)$ time. By Lemma 3.6, the expected number of iterations the algorithm performs until the inequality $|\mathcal{E}_{i,q}| \le M$ will be satisfied is $\le 1 + 1/\varepsilon$. Thus, the expected running time of the algorithm is

$$O\left(\left(1+\frac{1}{\varepsilon}\right) nr \log^2 n\right),$$

since the vertical decomposition of $\mathrm{simp'}_q(\mathcal{E}_{i,q})$ (which is the resulting cutting) can be computed in additional $O(nr)$ time. In fact, one can also compute, in $O(nr)$ time, for each trapezoid in the cutting, the lines of $\hat{S}$ that intersect it. $\square$

4. Empirical results.

In this section, we present the empirical results we got for computing cuttings in the plane using `CutRandomInc` and various related heuristics that we have implemented and experimented with. A program with a GUI demonstrating the algorithms and heuristics presented in the paper is available on the web in source form [HP98].

4.1. The implemented algorithms—using vertical trapezoids.

We have implemented the algorithm `CutRandomInc` presented in section 2 as well as several other algorithms for constructing cuttings. In this section, we report on the experimental results that we obtained.

Most of the algorithms we have implemented are variants of `CutRandomInc`. The algorithms implemented are the following (we denote by $K(\Delta)$ the set of lines that cross a vertical trapezoid Δ).

Classical: This is a variant of the algorithm of Chazelle and Friedman [CF90] for constructing a cutting. We pick a sample $R \subseteq \hat{S}$ of r lines and compute its arrangement $A = \mathcal{A}_{\mathcal{VD}}(R)$. For each active trapezoid $\Delta \in A$, we pick a random sample $R_\Delta \subseteq K(\Delta)$ of size $6k \log k$, where $k = \lceil r|K(\Delta)|/n \rceil$, and compute the arrangement of $\mathcal{A}_{\mathcal{VD}}(R_\Delta)$ inside Δ. If $\mathcal{A}_{\mathcal{VD}}(R_\Delta)$ is not a $(1/r)$-cutting, then the classical algorithm performs resampling inside Δ until it reaches a cutting. Our implementation is more naive, and it simply continues recursively into the active subtrapezoids of $\mathcal{A}_{\mathcal{VD}}(R_\Delta)$.

Cut randomized incremental: This is `CutRandomInc` without merging, as described in Figure 2.

Randomized incremental: This is `CutRandomInc` with merging.

The following four heuristics, for which we currently do not have a proof of any concrete bound on the expected size of the cutting that they generate, also perform well in practice.

Parallel incremental: Let $\mathcal{C}_i$ be the covering generated in the ith iteration of the algorithm. For each active trapezoid $\Delta \in \mathcal{C}_i$, pick a random line from $K(\Delta)$ and insert it into Δ (i.e., split Δ accordingly). Continue until there are no active trapezoids. Note that unlike `CutRandomInc` the insertion operations are performed locally inside each trapezoid, and the line chosen for insertion in each trapezoid is independent of the lines chosen for other trapezoids.

Greedy trapezoid: This is a variant of `CutRandomInc`, where we try to be "smarter" about the line inserted into the partition in each iteration. Let V_i be the set of trapezoids of $\mathcal{C}_i$ with maximal weight. We pick randomly a trapezoid Δ out of the trapezoids of V_i and pick randomly a line s from $K(\Delta)$. We then insert s into $\mathcal{C}_i$.

Greedy line: Similar to *greedy trapezoid*, but here we compute the set $\mathcal{U}$ of lines of $\hat{S}$, for which $w'(s)$ is maximal, where $w'(s)$ is the number of active trapezoids in $\mathcal{C}_i$ that intersect the line s. We pick randomly a line from $\mathcal{U}$ and insert it into the current partition of the plane.

Greedy weighted line: Similar to *greedy line*, but our weight function is

$$w'(s) = \sum_{\Delta \in \mathcal{C}_i, s \cap \Delta \neq \emptyset, w(\Delta) > n/r} \left\lfloor \frac{w(\Delta)}{\left\lfloor \frac{n}{3r} \right\rfloor} \right\rfloor ;$$

namely, we give a higher priority to lines that intersect heavier $(1/r)$-active trapezoids.

4.2. Polygonal cuttings. In judging the quality of cuttings, the size of the cutting is of major concern. However, other factors might also be important. For example we want the regions defining the cutting to be as simple as possible. Furthermore, there are applications where we are not interested directly in the size of the cutting, but rather in the overall number of vertices defining the cutting regions. This is useful when applying cuttings in the dual plane and transforming the vertices of the cutting back to the primal plane, as done in the computation of partition trees [Mat92]. A natural question is the following: Can one compute better cuttings if one is willing to use cutting regions which are different from vertical trapezoids?

For example, if one is willing to cut using nonconvex regions having a nonconstant description complexity, the size of the cutting can be improved to $4r^2+2r+2$ [Mat98]. On the other hand, if one wishes to cut a collection of lines by triangles instead of trapezoids, the situation becomes somewhat disappointing, because the smallest cuttings currently known for this case are generated by taking the cutting of Remark 3.4 and by splitting each trapezoid into two triangles. This results in cuttings having (roughly) $16r^2$ triangles.

In this section, we present a slightly different approach for computing cuttings, suggested to us by Matoušek, that works extremely well in practice. The new approach, a variant of which has already been presented in section 2.1, is based on cuttings using polygonal convex regions with a small number of sides, instead of vertical trapezoids. Namely, we apply `CutRandomInc`, where each region is a convex polygon (of constant complexity). Whenever we insert a new line into an active region, we split the polygon into two new polygons. Of course, it might be that the number of vertices of a new polygon is too large. If so, we split each such polygon into two subpolygons ensuring that the number of vertices of the new polygons is below our threshold.

Intuitively, the benefit in this approach is that the number of superfluous entities (i.e., vertical walls in the case of vertical trapezoids) participating in the definition of the cutting regions is much smaller. Moreover, since the cutting regions are less restrictive, the algorithm can be more flexible in its maintenance of the active regions.

Here are the different methods we tried.

PolyTree: We use `CutRandomInc` where each region is a convex polygon having at most k-sides. When inserting a new line, we first split each of the active regions

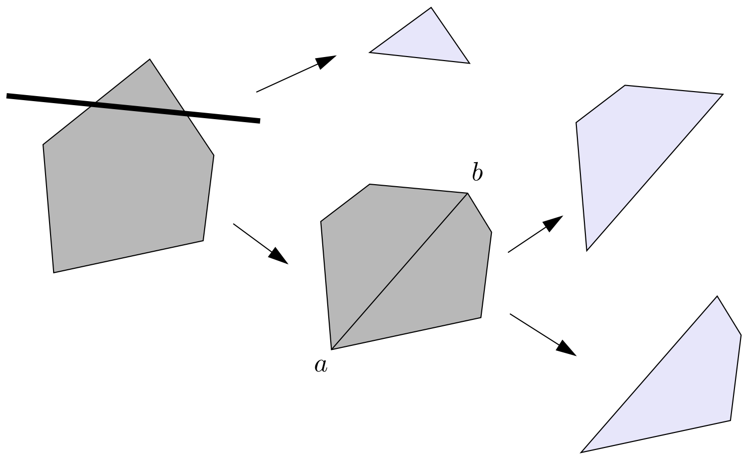


FIG. 7. *In the* `PolyTree` *algorithm, each time a polygon is being split by a line, we might have to further split it because a split region might have too many vertices.*

that intersect it into two subpolygons. If a split region R has more than k sides, we further split it using the diagonal of R that achieves the best balanced partition of R; namely, it is the pair of vertices a, b realizing the following minimum:

$$\min_{a,b \in V(R)} \max\left(w(R \cap H_{ab}^{+}), w(R \cap H_{ab}^{-})\right),$$

where $V(R)$ is the set of vertices of R, $w(R)$ is the number of lines intersecting R, and H_{ab}^{+} (resp., H_{ab}^{-}) is the closed half-plane lying to the right (resp., left) of ab. See Figure 7.

PolyTriangle: Modified `PolyTree` for generating cuttings by triangles. In each stage, we check whether a newly created region R can be triangulated into a set of inactive triangles. To do so, compute an arbitrary triangulation of the region R and check if all the triangles generated in our (arbitrary) triangulation of R are inactive. If so, replace R in our cuttings by its triangulation.[4]

PolyDeadLeaf: Modified `PolyTree` for generating cuttings by triangles. Whenever a region is being created we check whether it has a leaf triangle (a triangle defined by three consecutive vertices of the region) that is inactive. If we find such an inactive triangle, we add it immediately to the final cutting. We repeat this process until the region cannot be further shrunk.

PolyVertical: (This is the variant presented in section 2.1.) We use `PolyTree`, but instead of splitting along a diagonal, we split along a vertical ray emanating from one of the vertices of the region. The algorithm also tries to remove dead regions from the left and right side of the region. Intuitively, each region is now an extended vertical trapezoid having a convex ceiling and floor, with at most two additional vertical walls.

Remark 4.1. Note that for all the polygonal cutting methods, except `CRIVPolygon` (this is `PolyVertical` without the removal of dead regions; see section 2.1) and `PolyVertical`, it is not even clear that the number of regions they maintain in the ith iteration is $O(i^2)$. Thus, the proof of Theorem 2.10 does not work for those methods.

[4]Computing the "best" triangulation (i.e., the weight of the heaviest triangle is minimized) is relatively complicated and requires dynamic programming. It is not clear that it is going to perform better than `PolyDeadLeaf`, described below.

4.3. Implementation details. As an underlying data structure for our testing, we implemented the history-graph data structure [Sei91]. Our random arrangements were constructed by choosing n points uniformly and independently on the left side of the unit square, and similarly on the right side of the unit square. We sorted the points and connected them by lines in a transposed manner. This yields a random arrangement with all the $\binom{n}{2}$ intersections inside the unit square.

We had implemented our algorithm in `C++`. We had encountered problems with floating point robustness at an early stage of the implementation and decided to use exact arithmetic instead, using `LEDA` rational numbers [MN95]. While this solved the robustness problems, we had to deal with a couple of other issues:

- Speed: Using exact arithmetic instead of floating point arithmetic resulted in a slowdown by a factor 20–40. The time to perform an operation in exact arithmetic is proportional to the bit-sizes of the numbers involved. To minimize the size of the numbers used in the computations, we normalized the line equations so that the coefficients are integer numbers (in reduced form). Using more advanced techniques one can get close to floating-point speed with the "security" provided by exact arithmetic. See [AHH+99].
- Memory consumption: A `LEDA` rational is represented by a block of memory dynamically allocated on the heap. In order to save, both in the memory consumed and the time used by the dynamic memory allocator, we observe that in a representation of vertical decomposition the same number appears in several places (i.e., an x-coordinate of an intersection point appears in 6 different vertical trapezoids). We reduce memory consumption, by storing such a number only once. To do so, we use a repository of rational numbers generated so far by the algorithm. Whenever we compute a new x-coordinate, we search it in the repository, and if it does not exist, then we insert it. In particular, each vertical trapezoid is represented by two pointers to its x_{left} and x_{right} coordinates and pointers to its top and bottom lines.
  The repository is implemented using Treaps [SA96].

4.4. Handling degeneracies. Geometric degeneracy is one of the main obstacles when implementing geometric algorithms. To overcome this problem, we used exact arithmetic. While this ensured that the underlining geometric primitives work correctly, it does not tackle the problem of geometric degeneracies directly. Fortunately, in our case the handling of geometric degeneracies is straightforward.

Indeed, the various algorithms we presented for computing cuttings use only two geometric primitives. The first is *split*, which split a region (i.e., vertical trapezoid/convex polygon) Δ by a line l that crosses it (the line might be an input line, or a line connecting two vertices of the region) into a constant number of regions $R_1, \ldots, R_k$. Here, to overcome degeneracy the algorithm throws away regions having an empty interior. If the regions are polygons, an additional step is added to remove superfluous collinear vertices (i.e., a vertex lying on the line induced by its two neighbors) from the newly created regions.

The second primitive checks whether a line l crosses the interior of a region Δ. This is done by checking for each vertex of Δ, on which side of l it lies. If two vertices of Δ lie on opposite sides of l, then l crosses the interior of Δ. Since the program uses exact arithmetic this primitive gives the correct results.

4.5. Results—vertical trapezoids. The empirical results we got for the algorithms/heuristics of subsection 4.1 are depicted in Tables 1–5.

Table 1
Results for 1/2-cuttings. Each entry is the size of the minimal cutting computed, divided by $2^2 = 4$. *The corresponding value in Matoušek's construction* [Mat98] *is smaller than* 12.00.

Lines	Parallel Inc	Classical	Randomized Inc	Greedy trapezoid	Greedy line	Greedy weighted line
4	1.50	1.50	1.50	1.50	1.50	1.50
8	2.00	12.50	2.75	2.25	2.25	2.50
16	4.75	25.75	4.25	4.75	4.00	4.50
32	4.75	24.75	7.00	5.50	6.00	6.25
64	6.75	28.25	6.50	7.50	7.50	7.25
128	8.75	26.75	7.75	7.25	8.25	8.50
256	8.75	30.75	6.50	8.00	6.75	7.25
512	6.00	36.50	8.25	9.75	8.50	8.00
1024	9.25	26.00	10.00	7.75	7.75	9.00
2048	7.50	28.50	9.00	7.25	8.75	9.75
4096	7.50	35.25	8.50	8.25	7.75	7.50
8192	8.25	36.75	7.00	7.75	6.25	7.50
16384	9.00	30.25	8.75	8.00	8.50	8.75
32768	10.25	33.00	7.75	9.00	6.25	7.75
65536	10.00	31.25	6.50	6.75	6.75	8.50

Table 2
Results for 1/4-cuttings. Each entry is the size of the minimal cutting computed, divided by $4^2 = 16$. *The corresponding value in Matoušek's construction* [Mat98] *is smaller than* 9.75.

Lines	Parallel Inc	Classical	Randomized Inc	Greedy trapezoid	Greedy line	Greedy weighted line
8	1.88	2.69	1.56	2.00	1.62	1.62
16	3.69	20.31	3.44	3.62	3.50	3.62
32	6.12	31.12	6.56	5.50	5.75	5.50
64	8.25	35.94	8.06	7.69	7.12	8.06
128	9.19	37.12	9.75	9.94	8.62	7.88
256	11.00	44.06	9.62	9.00	10.25	8.19
512	11.75	48.75	9.81	10.31	9.75	10.12
1024	12.75	46.88	12.12	11.25	10.25	10.62
2048	11.19	37.00	11.50	11.00	10.81	10.50
4096	11.81	44.38	10.94	11.19	10.62	10.50
8192	12.19	52.00	10.19	11.06	10.25	10.00
16384	12.31	43.88	10.94	11.25	10.31	10.88
32768	11.50	42.69	10.69	11.81	11.31	10.25

Table 3
Results for 1/8-cuttings. Each entry is the size of the minimal cutting computed, divided by $8^2 = 64$. *The corresponding value in Matoušek's construction* [Mat98] *is smaller than* 8.81.

Lines	Parallel Inc	Classical	Randomized Inc	Greedy trapezoid	Greedy line	Greedy weighted line
16	2.23	4.08	2.06	1.89	1.86	1.83
32	4.80	23.17	4.19	3.92	3.91	3.53
64	7.67	37.25	6.64	6.27	6.31	6.12
128	10.44	45.50	8.84	8.83	8.80	8.31
256	11.56	44.12	9.91	10.53	9.91	9.94
512	12.77	50.14	11.36	11.11	10.86	11.20
1024	13.31	47.58	11.61	11.33	10.95	11.31
2048	13.42	54.84	12.36	11.17	12.38	11.17
4096	15.00	53.17	12.08	12.22	12.08	11.98
8192	13.98	51.75	11.73	12.19	12.67	12.27

TABLE 4

Results for 1/16*-cuttings. Each entry is the size of the minimal cutting computed, divided by* $16^2 = 256$*. The corresponding value in Matoušek's construction* [Mat98] *is smaller than* 8.39

Lines	Parallel Inc	Classical	Randomized Inc	Greedy trapezoid	Greedy line	Greedy weighted line
32	2.70	5.54	2.18	2.09	2.04	2.11
64	5.38	24.27	4.52	4.21	4.36	4.20
128	8.16	44.00	7.16	7.00	6.66	6.55
256	10.88	56.30	9.34	9.25	9.16	8.48
512	12.61	66.60	11.26	10.85	10.37	10.18
1024	14.02	66.64	12.30	11.40	11.23	11.10
2048	14.24	67.25	12.51	11.84	11.98	11.78

TABLE 5

Results for 1/32*-cuttings. Each entry is the size of the minimal cutting computed, divided by* $32^2 = 1024$*. The corresponding value in Matoušek's construction* [Mat98] *is smaller than* 8.19*.*

Lines	Parallel Inc	Classical	Randomized Inc	Greedy trapezoid	Greedy line	Greedy weighted line
64	2.84	5.45	2.26	2.19	2.14	2.14
128	5.48	24.73	4.54	4.44	4.33	4.22
256	8.89	52.62	7.52	7.05	6.68	6.61
512	11.26	67.72	9.48	9.54	8.98	8.87
1024	13.25	74.47	11.63	10.86	10.23	10.34

For each value of r and each value of n, we computed a random arrangement of lines inside the unit square, as described above. For each such arrangement, we performed 10 tests for each algorithm/heuristic. The tables present the size of the minimal cutting computed in those tests. Each entry is the size of the output cutting divided by r^2. In addition, each table caption presents a range containing the size of the cutting that can be obtained by Matoušek's algorithm [Mat98].

It is an interesting question whether using merging by `CutRandomInc` results in practice in smaller cuttings. We tested this empirically, and the results are presented in Table 6. As can be seen in Table 6, using merging does generate smaller cuttings, but the improvement in the cutting size is rather small. The difference in the size of the generated cuttings seems to be less than $2r^2$.

4.6. Implementing Matoušek's construction. In Table 7, we present the empirical results for Matoušek's construction, comparing it with the slight improvement described in Remark 3.4. For small values of r, the improved version yields considerably smaller cuttings than Matoušek's construction, making it the best method we are aware of for constructing small cuttings.

We had implemented Matoušek's algorithm naively, using quadratic space and time. Currently, this implementation cannot be used for larger inputs because it runs out of memory. Implementing the more efficient algorithm described in Theorem 3.7 is nontrivial since it requires the implementation of the rather complex data structure of Overmars and van Leeuwen [OvL81]. However, if it is critical to reduce the size of a cutting for large inputs, the algorithm of Theorem 3.7 seems to be the best available option.

Overall, Matoušek's algorithm gives the best results for cuttings by vertical trapezoids. However, for polygonal cuttings, the results we got are even better, as described below.

Table 6
Comparing the size of cuttings computed by `CutRandomInc`*, with or without using merging. Each entry is the size of the minimal cutting computed, divided by* r^2*.*

Number of lines	Value of r									
	2		4		8		16		32	
		Merge		Merge		Merge		Merge		Merge
4	1.50	1.50	—	—	—	—	—	—	—	—
8	2.25	2.75	1.88	1.50	—	—	—	—	—	—
16	4.75	4.00	3.44	3.56	2.27	2.03	—	—	—	—
32	5.00	5.75	6.44	6.25	4.86	4.31	2.56	2.14	—	—
64	7.25	7.75	8.25	9.00	7.36	6.45	5.30	4.54	2.78	2.29
128	8.00	7.50	8.94	9.38	9.56	9.00	8.48	7.19	5.44	4.52
256	8.00	8.00	11.94	9.88	11.86	10.81	10.84	9.73	8.26	7.29
512	5.75	9.00	10.25	10.62	13.05	11.48	12.36	11.29	11.17	9.55
1024	8.25	6.75	12.62	10.12	12.69	11.83	13.80	11.96	13.07	11.24
2048	7.50	8.50	11.31	10.31	13.06	12.66	14.11	13.44	14.04	12.38
4096	8.75	9.25	12.31	11.12	13.81	12.39	13.95	13.13	14.61	12.96
8192	7.50	8.00	12.00	12.12	13.34	12.97	14.67	12.67	14.72	13.18
16384	9.25	8.50	11.38	10.56	12.69	12.33	14.53	13.61	15.02	13.30
32768	7.25	7.50	12.12	10.44	13.00	12.59	13.96	13.00	15.17	13.46

Table 7
Comparing the size of cuttings computed by Matoušek's method, to the slightly improved method described in section 3*.*

Number of lines	Value of r									
	2		4		8		16		32	
		Impr'		Impr'		Impr'		Impr'		Impr'
4	8.75	5.25	—	—	—	—	—	—	—	—
8	10.75	6.25	8.25	6.56	—	—	—	—	—	—
16	10.25	7.25	9.19	7.44	8.14	7.27	—	—	—	—
32	10.00	7.00	9.38	7.56	8.64	7.64	8.06	7.63	—	—
64	10.00	7.25	9.81	7.25	8.66	7.73	8.32	7.82	8.04	7.81
128	10.50	6.50	9.31	7.06	8.72	7.70	8.38	7.90	8.15	7.91
256	11.00	7.50	9.81	7.44	8.81	7.73	8.37	7.90	8.18	7.92
512	11.00	6.75	10.06	7.69	8.81	7.84	8.41	7.88	8.17	7.94

4.7. Results—polygonal cuttings. The results for polygonal cuttings are presented in Table 8. As seen in the tables, the polygonal cutting methods perform well in practice. In particular, the `PolyTree` method generated cuttings of average size (roughly) $7.5r^2$, beating all the cutting methods that use vertical trapezoids.

As for triangles, the situation is even better: `PolyDeadLeaf` generates cuttings by triangles of size $\leq 12r^2$. (That is better by an additive factor of about $4r^2$ than the best theoretical bound.)

To summarize, polygonal cutting methods seem to be the clear winner in practice. They generate cuttings of a small size, with a small number of vertices and small number of triangles.

5. Conclusions. In this paper, we have presented a new approach, different from that of [CF90], for constructing cuttings in the plane. The new algorithm is rather simple and easy to implement. We have proved the correctness and bounded the expected output size and expected running time of several variants of the new algorithm and have also demonstrated that the new algorithms perform much better in practice than the algorithm of [CF90]. We believe that the results in this paper show that planar cuttings are practical and might be useful in practice when constructing

TABLE 8

Results for (1/8)*-cutting of* 1024 *lines. The best results (and fastest) were generated by* `PolyTree` *using polygons with at most* 8 *sides. The execution time for this variant was less than* 40 *seconds on a Pentium Pro* 200*MhZ computer.*

	Size / r^2			Triangles / r^2			Vertices / r^2			Cutting region
	min	avg	max	min	avg	max	min	avg	max	
ParallelInc	13.64	15.03	16.33	27.28	30.06	32.66	21.84	24.34	26.67	VTrapezoid
Chazelle–Friedman	44.30	55.73	79.48	88.59	111.47	158.97	52.78	65.05	89.16	VTrapezoid
Cut random Inc	12.00	12.77	13.52	24.00	25.53	27.03	14.52	15.50	16.30	VTrapezoid
Greedy random	11.17	11.69	12.72	22.34	23.38	25.44	13.52	14.19	15.69	VTrapezoid
Greedy line	10.36	11.02	11.77	20.72	22.05	23.53	12.53	13.19	14.03	VTrapezoid
Greedy weighted line	10.33	11.20	12.16	20.66	22.42	24.31	12.39	13.33	14.36	VTrapezoid
Matoušek		8.67			17.34			13.22		VTrapezoid
Matoušek-improved		7.72			15.44			11.78		VTrapezoid
Polytree	9.73	10.83	12.25	14.88	16.20	18.59	9.91	10.55	12.03	4-Polygon
Polytree	7.39	7.89	8.41	12.94	14.08	15.25	8.86	9.61	10.42	5-Polygon
Polytree	6.88	7.47	8.16	13.28	14.42	15.56	9.36	10.14	11.19	6-Polygon
Polytree	6.92	7.38	7.70	13.70	14.61	15.19	9.62	10.30	10.83	7-Polygon
Polytree	6.34	7.28	8.36	12.69	14.53	16.72	8.86	10.41	11.95	**8-Polygon**
Polytree	6.66	7.34	8.12	13.31	14.70	16.22	9.47	10.52	11.81	9-Polygon
Polytree	6.56	7.22	7.91	13.12	14.44	15.81	9.42	10.25	10.98	10-Polygon
Polytree	7.17	7.62	8.14	14.34	15.27	16.28	10.20	11.02	11.64	11-Polygon
PolyTriang ($\leq$ 4)	11.38	15.03	16.97	11.38	15.03	16.97	7.42	9.64	10.75	Triangle
PolyTriang ($\leq$ 5)	13.00	13.72	14.75	13.00	13.72	14.75	8.73	9.27	10.16	Triangle
PolyTriang ($\leq$ 6)	11.91	13.20	14.06	11.91	13.20	14.06	8.19	9.05	9.62	Triangle
PolyTriang ($\leq$ 7)	11.47	13.08	14.94	11.47	13.08	14.94	8.02	9.05	10.25	Triangle
PolyTriang ($\leq$ 8)	11.50	12.89	14.09	11.50	12.89	14.09	8.17	8.97	9.72	Triangle
PolyTriang ($\leq$ 9)	12.06	13.17	15.50	12.06	13.17	15.50	8.47	9.27	11.00	Triangle
PolyTriang ($\leq$ 10)	11.47	12.47	13.47	11.47	12.47	13.47	8.06	8.75	9.30	Triangle
PolyTriang ($\leq$ 11)	12.25	13.28	14.09	12.25	13.28	14.09	8.27	9.23	9.95	Triangle
PolyDeadLeaf ($\leq$ 4)	11.09	11.98	12.81	11.09	11.98	12.81	7.73	8.33	9.08	Triangle
PolyDeadLeaf ($\leq$ 5)	10.38	11.02	11.94	10.38	11.02	11.94	7.33	7.83	8.62	Triangle
PolyDeadLeaf ($\leq$ 6)	10.78	11.50	13.00	10.78	11.50	13.00	7.69	8.22	9.27	Triangle
PolyDeadLeaf ($\leq$ 7)	9.58	11.81	13.78	9.58	11.81	13.78	6.89	8.42	9.61	Triangle
PolyDeadLeaf ($\leq$ 8)	10.59	11.47	12.47	10.59	11.47	12.47	7.64	8.23	8.92	Triangle
PolyDeadLeaf ($\leq$ 9)	11.00	11.62	13.12	11.00	11.62	13.12	7.78	8.31	9.39	Triangle
PolyDeadLeaf ($\leq$ 10)	9.97	10.75	12.47	9.97	10.75	12.47	7.02	7.72	8.70	Triangle
PolyDeadLeaf ($\leq$ 11)	9.97	11.30	12.34	9.97	11.30	12.34	6.98	8.05	8.64	Triangle
PolyVertical	11.61	13.05	14.09	18.94	21.56	23.91	13.81	15.64	16.78	4-Polygon
PolyVertical	8.98	9.67	10.42	14.52	15.61	16.98	11.34	12.31	13.20	5-Polygon
PolyVertical	8.50	9.41	11.16	13.17	14.81	17.55	10.89	12.14	14.30	6-Polygon
PolyVertical	8.25	9.16	10.02	12.88	14.31	15.75	10.61	11.83	13.03	7-Polygon
PolyVertical	8.48	9.33	10.66	13.22	14.55	16.59	11.02	12.05	13.72	8-Polygon
PolyVertical	8.36	9.22	10.05	13.11	14.33	15.58	10.98	11.92	12.86	9-Polygon
PolyVertical	8.39	9.06	10.55	13.25	14.14	16.53	10.98	11.77	13.83	10-Polygon
PolyVertical	8.31	9.03	9.91	12.92	14.08	15.44	10.80	11.73	12.91	11-Polygon

data structures for range searching and related applications.

Moreover, the empirical results show that the size of the cutting constructed by the new algorithms is not considerably larger (and in some cases even smaller) than the cuttings that can be computed by the currently best theoretical algorithm (too slow to be useful in practice due to its $O(n^2)$ running time) of Matoušek [Mat98]. The empirical constants that we obtain are generally between 10 and 13 (for vertical trapezoids). For polygonal cuttings we get a constant of 8 by cutting by convex polygons (using `PolyTree`) having at most 6 vertices. Moreover, the various variants of `CutRandomInc` seem to produce constants that are rather close to each other. As noted above, the method described in Remark 3.4 generates the smallest cuttings by vertical trapezoids.

As for running time, the results we got in practice are the following: In computing a (1/8)-cutting of 1024 lines, the fastest algorithm was `PolyTree`, requiring about half a minute on average. The other polygonal methods lagged slightly behind. `CutRandomInc` was the fastest algorithm that produced cuttings by vertical trapezoids, being several times slower. Matoušek's method required several hours due to our naive (i.e., $O(n^2)$) implementation. This information should be taken with reservation, since no serious effort had gone into optimizing the code for speed, and those measurements tend to change from execution to execution. (Recall also that we use exact arithmetic, which slows down the running time significantly.)

Given these results, we recommend for use in practice one of the polygonal cutting methods. They perform well in practice, and they should be used whenever possible.

If we are restricted to a vertical trapezoid, `CutRandomInc` seems like a reasonable algorithm to be used in practice (without merging, as merging is the only "nontrivial" part in the implementation of the algorithm).

There are several interesting open problems for further research:

- Can one obtain provable bounds on the expected size of the cutting generated by the `PolyTree` method, and on its running time? (Remember that in the `PolyTree` method the cutting regions are convex polygons with a constant number of edges.) The same question applies for all the other methods we had implemented.
- Can one prove the existence of a cutting smaller than the one guaranteed by the algorithm in Remark 3.4 for specific values of r? For example, Table 1 suggests a smaller cutting should exist for $r = 2$. In particular, the test results hint that a cutting made out of 32 vertical trapezoids should exist, while the cutting size guaranteed by Matoušek's algorithm [Mat98] is 48.
- Can one generate smaller cuttings by modifying `CutRandomInc` to be smarter in its decision on when to merge trapezoids?
- Is there a simple and practical algorithm for computing cuttings in three and higher dimensions? The current algorithms seems to be far from practical.
- Can the following heuristic improve (substantially) the size of the cuttings in practice? After the cuttings are computed, pass over the cutting regions and merge adjacent regions that are adjacent and compatible, so that the resulting regions are still $(1/r)$-inactive. Inspecting the output of our test program in Figure 1 indicates that this indeed the case.

Acknowledgments. The author wishes to thank Pankaj Agarwal, Boris Aronov, Hervé Brönnimann, Bernard Chazelle, Jiří Matoušek, and Joe Mitchell for helpful discussions and suggestions concerning the problems studied in this paper and related problems.

The author also wishes to thank deeply Micha Sharir for his help and guidance in preparing the paper. He contributed several key ideas to the new simplified analysis of `CutRandomInc`.

REFERENCES

[Aga91] P. Agarwal, *Geometric partitioning and its applications*, in Computational Geometry: Papers from the DIMACS Special Year, J. Goodman, R. Pollack, and W. Steiger, eds., AMS, Providence, RI, 1991, pp. 1–37.

[AM95] P. Agarwal and J. Matoušek, *Dynamic half-space range reporting and its applications*, Algorithmica, 13 (1995), pp. 325–345.

[AHH+99] Y. Aharoni, D. Halperin, I. Hanniel, S. Har-Peled, and C. Linhart, *On-line zone construction in arrangements of lines in the plane*, in Proceedings of the Workshop on Algorithm Engineering, J. Vitter and C. Zaroliagis, eds., Lecture Notes in Comput. Sci. 1668, Springer, New York, 1999, pp. 139–153.

[BDH99] K.-F. Böhringer, B. Donald, and D. Halperin, *The area bisectors of a polygon and force equilibria in programmable vector fields*, Discrete Comput. Geom., 22 (1999), pp. 269–285.

[Cha93] B. Chazelle, *Cutting hyperplanes for divide-and-conquer*, Discrete Comput. Geom., 9 (1993), pp. 145–158.

[CF90] B. Chazelle and J. Friedman, *A deterministic view of random sampling and its use in geometry*, Combinatorica, 10 (1990), pp. 229–249.

[Cla87] K. L. Clarkson, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[CS89] K. L. Clarkson and P. W. Shor, *Applications of random sampling in computational geometry,* II, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[dBDS95] M. de Berg, K. Dobrindt, and O. Schwarzkopf, *On lazy randomized incremental construction*, Discrete Comput. Geom., 14 (1995), pp. 261–286.

[dBS95] M. de Berg and O. Schwarzkopf, *Cuttings and applications*, Internat. J. Comput. Geom. Appl., 5 (1995), pp. 343–355.

[dBvKOS97] M. de Berg, M. van Kreveld, M. H. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin, 1997.

[Ede87] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[HP98] S. Har-Peled, *Cuttings—demo program*, available online from http://www.math.tau.ac.il/~sariel/CG/cutting/cuttings.html, 1998.

[HW87] D. Haussler and E. Welzl, *Epsilon-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–151.

[Mat92] J. Matoušek, *Efficient partition trees*, Discrete Comput. Geom., 8 (1992), pp. 315–334.

[Mat98] J. Matoušek, *On constants for cuttings in the plane*, Discrete Comput. Geom., 20 (1998), pp. 427–448.

[MN95] K. Mehlhorn and S. Näher, *LEDA: A platform for combinatorial and geometric computing*, Commun. ACM, 38 (1995), pp. 96–102.

[OvL81] M. H. Overmars and J. van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. System Sci., 23 (1981), pp. 166–204.

[Sei91] R. Seidel, *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, Comput. Geom., 1 (1991), pp. 51–64.

[SA96] R. Seidel and C. R. Aragon, *Randomized search trees*, Algorithmica, 16 (1996), pp. 464–497.

[SA95] M. Sharir and P. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 29, No. 6, pp. 2040–2073

ON QUIESCENT RELIABLE COMMUNICATION[*]

MARCOS KAWAZOE AGUILERA[†], WEI CHEN[†], AND SAM TOUEG[†]

Abstract. We study the problem of achieving reliable communication with *quiescent* algorithms (i.e., algorithms that eventually stop sending messages) in asynchronous systems with process crashes and lossy links. We first show that it is impossible to solve this problem in asynchronous systems (with no failure detectors). We then show that, among failure detectors that output lists of suspects, the weakest one that can be used to solve this problem is $\diamond\mathcal{P}$, a failure detector that cannot be implemented. To overcome this difficulty, we introduce an implementable failure detector called *Heartbeat* and show that it can be used to achieve quiescent reliable communication. Heartbeat is novel: in contrast to typical failure detectors, it does not output lists of suspects and it is implementable without timeouts. With Heartbeat, many existing algorithms that tolerate only process crashes can be transformed into quiescent algorithms that tolerate both process crashes and message losses. This can be applied to consensus, atomic broadcast, k-set agreement, atomic commitment, etc.

Key words. algorithms, reliability, reliable communication, quiescence, asynchronous systems, heartbeat, crash failures, link failures, failure detection, fault-tolerance, message passing, processor failures

AMS subject classifications. 68M15, 68Q22

PII. S0097539798341296

1. Introduction.

1.1. Motivation.

We focus on the problem of *quiescent* reliable communication in asynchronous message-passing systems with process crashes and lossy links. To illustrate this problem consider a system of two processes, a sender s and a receiver r, connected by an asynchronous bidirectional link. Process s wishes to send some message m to r. Suppose first that no process may crash, but the link between s and r may lose messages (in both directions). If we put no restrictions on message losses it is obviously impossible to ensure that r receives m. An assumption commonly made to circumvent this problem is that the link is *fair*: if a message is sent infinitely often, then it is received infinitely often.

With such a link, s could repeatedly send copies of m forever, and r is guaranteed to eventually receive m. This is impractical, since s never stops sending messages. The obvious fix is the following protocol: (a) s sends a copy of m repeatedly until it receives $ack(m)$ from r, and (b) upon each receipt of m, r sends $ack(m)$ back to s. Note that this protocol is *quiescent*: eventually no process sends or receives messages.

The situation changes if, in addition to message losses, process crashes may also occur. The protocol above still works, but it is not quiescent anymore: for example, if r crashes before sending $ack(m)$, then s will send copies of m forever. Is there a *quiescent* protocol ensuring that if neither s nor r crashes then r eventually receives m? It turns out that the answer is no, even if one assumes that the link can only lose a finite number of messages.

[*]Received by the editors June 26, 1998; accepted for publication (in revised form) June 22, 1999; published electronically April 18, 2000. Research partially supported by NSF grant CCR-9402896 and CCR-9711403, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship. An extended abstract of part of this paper appeared in Proceedings of the 11th International Workshop on Distributed Algorithms, Lecture Notes in Comput. Sci. 1320, Springer-Verlag, New York, 1997, pp. 126–140.

http://www.siam.org/journals/sicomp/29-6/34129.html

[†]Computer Science Department, Cornell University, Ithaca, NY 14853-7501 (aguilera@cs.cornell.edu, weichen@cs.cornell.edu, sam@cs.cornell.edu).

Since process crashes and message losses are common types of failures, this negative result is an obstacle to the design of fault-tolerant distributed systems. In this paper, we explore the use of *unreliable failure detectors* to circumvent this obstacle. Roughly speaking, unreliable failure detectors provide (possibly erroneous) hints on the operational status of processes. Each process can query a local failure detector module that provides some information about which processes have crashed. This information is typically given in the form of a list of *suspects*.

In general, failure detectors can make mistakes: a process that has crashed is not necessarily suspected, and a process may be suspected even though it has not crashed. Moreover, the local lists of suspects dynamically change and lists of different processes do not have to agree (or even eventually agree). Introduced in [12], the abstraction of unreliable failure detectors has been used to solve several important problems such as consensus, atomic broadcast, group membership, nonblocking atomic commitment, and leader election [3, 20, 26, 28, 32, 34].

Our goal is to use unreliable failure detectors to achieve quiescence, but before we do so we must address the following important question. Note that any reasonable implementation of a failure detector in a message-passing system is itself *not* quiescent: a process being monitored by a failure detector must periodically send a message to indicate that it is still alive, and it must do so forever (if it stops sending messages it cannot be distinguished from a process that has crashed). Given that failure detectors are not quiescent, does it still make sense to use them as a tool to achieve quiescent applications (such as quiescent reliable broadcast, consensus, or group membership)?

The answer is yes for two reasons. First, a failure detector is intended to be a basic system service that is *shared* by many applications during the lifetime of the system, and so its cost is amortized over all these applications. Second, failure detection is a service that needs to be active forever—and so it is natural that it sends messages forever. In contrast, many applications (such as a single remote procedure call (RPC) or the reliable broadcast of a single message) should not send messages forever, i.e., they should be quiescent. Thus, there is no conflict between the goal of building quiescent applications and the use of a nonquiescent failure detection service as a tool to achieve this goal.

1.2. Achieving quiescent reliable communication using failure detectors. How can we use an unreliable failure detector to achieve quiescent reliable communication in the presence of process and link failures? This can be done with the *eventually perfect* failure detector $\diamond\mathcal{P}$ [12]. Intuitively, $\diamond\mathcal{P}$ satisfies the following two properties: (a) if a process crashes, then there is a time after which it is permanently suspected, and (b) if a process does not crash, then there is a time after which it is never suspected. Using $\diamond\mathcal{P}$, the following obvious algorithm solves our sender/receiver example: (a) while s has not received $ack(m)$ from r, it periodically does the following: s queries $\diamond\mathcal{P}$ and sends a copy of m to r if r is not currently suspected; (b) upon each receipt of m, r sends $ack(m)$ back to s. Note that this algorithm is *quiescent*: eventually no process sends or receives messages.

So $\diamond\mathcal{P}$ is sufficient to achieve quiescent reliable communication. But is it necessary? In the first part of the paper, we show that among all failure detectors that output lists of suspects, $\diamond\mathcal{P}$ is indeed the *weakest* one that can be used to solve this problem. Unfortunately, $\diamond\mathcal{P}$ is not implementable (this would violate a known impossibility result [17, 12]). Thus, at a first glance, it seems that achieving quiescent reliable communication requires a failure detector that cannot be implemented. In the second part of the paper, we show that this is not so.

In fact, we show that quiescent reliable communication can be achieved with a failure detector that *is implementable* in systems with process crashes and lossy links. This new failure detector, called *Heartbeat* and denoted $\mathcal{HB}$, is very simple. Roughly speaking, the failure detector module of $\mathcal{HB}$ at a process p outputs a vector of counters, one for each neighbor q of p. If neighbor q does not crash, its counter at p increases with no bound. If q crashes, its counter eventually stops increasing. The basic idea behind an implementation of $\mathcal{HB}$ is the obvious one: each process periodically sends an *I-am-alive* message (a "heartbeat") and every process receiving a heartbeat increases the corresponding counter.[1]

$\mathcal{HB}$ should not be confused with existing implementations of failure detectors (some of which have modules that are also called *heartbeat* [35, 10]). Even though existing failure detectors are also based on the repeated sending of a heartbeat, they use timeouts on heartbeats in order to derive lists of processes considered to be up or down, and applications can only see these lists. In contrast, $\mathcal{HB}$ does *not* use timeouts on the heartbeats of a process in order to determine whether this process has failed or not. $\mathcal{HB}$ just counts the *total number of heartbeats* received from each process and outputs these "raw" counters without any further processing or interpretation.

A remark is now in order regarding the practicality of $\mathcal{HB}$. As we mentioned above, $\mathcal{HB}$ outputs a vector of unbounded counters. In practice, these unbounded counters are not a problem for the following reasons. First, they are in *local memory* and not in messages—our $\mathcal{HB}$ implementations use bounded messages. Second, if we bound each local counter to 64 bits and assume a rate of one heartbeat per nanosecond, which is orders of magnitude higher than currently used in practice, then $\mathcal{HB}$ will work for more than 500 years.

1.3. Detailed outline of the results. We focus on two types of reliable communication mechanisms: *quasi-reliable send and receive*, and *reliable broadcast*. Roughly speaking, a pair of send and receive primitives is quasi-reliable if it satisfies the following property: if processes s and r are *correct* (i.e., they do not crash), then r receives a message from s exactly as many times as s sent that message to r. Reliable broadcast [23] ensures that if a correct process broadcasts a message m then all correct processes deliver m; moreover, all correct processes deliver the same set of messages. Our goal is to obtain quiescent implementations of these primitives in networks that do not partition permanently. More precisely, we consider networks in which processes may crash and links may lose messages, but every pair of correct processes are connected through some *fair path*, i.e., a path containing only fair links and correct processes.

We first show that in asynchronous systems (with no failure detectors), there is no quiescent implementation of quasi-reliable send and receive or of reliable broadcast in such networks (even if we assume that links can lose only a finite number of messages). We then show that the weakest failure detector *with bounded output size*[2] that can be used to solve these problems is $\diamond\mathcal{P}$—which is not implementable.

To overcome this difficulty, we introduce $\mathcal{HB}$, a failure detector that outputs unbounded counters, and show that $\mathcal{HB}$ is strong enough to achieve quiescent reliable communication but weak enough to be implementable. We consider two types of networks. In the first type, all links are bidirectional and fair. In the second one, some links are unidirectional, and some links have no restrictions on message losses, i.e.,

[1] As we will see, however, in some types of networks the actual implementation is not as easy.

[2] Note that a list of suspects has bounded size.

they are not fair. Examples of such networks are unidirectional rings that intersect. For the first type of networks, a common one in practice, the implementation of $\mathcal{HB}$ and the reliable communication algorithms are very simple and efficient. The algorithms for the second type are significantly more complex.

We then consider two stronger types of communication primitives, namely, *reliable send and receive* and *uniform reliable broadcast*, and give quiescent implementations that use $\mathcal{HB}$. These implementations assume that a majority of processes are correct (a result in [5] shows that this assumption is necessary).

We conclude the paper by showing how $\mathcal{HB}$ can be used to extend previous work in order to solve problems with algorithms that are both quiescent and tolerant of process crashes and messages losses. First, we explain how $\mathcal{HB}$ can be used to transform many existing algorithms that tolerate process crashes into quiescent algorithms that tolerate both process crashes and message losses (fair links). This transformation can be applied to the algorithms for consensus in [2, 7, 9, 12, 14, 16, 33], for atomic broadcast in [12], for k-set agreement in [13], for atomic commitment in [20], for approximate agreement in [15], etc. Next, we show that $\mathcal{HB}$ can be used to extend the work in [5] to obtain the following result. Let P be a problem. Suppose P is correct-restricted (i.e., its specification refers only to the behavior of correct processes), or a majority of processes are correct. If P is solvable with a quiescent protocol that tolerates only process crashes, then P is also solvable with a quiescent protocol that tolerates process crashes and message losses.[3]

To summarize, in this paper, we do the following.

1. We explore the use of unreliable failure detectors to achieve *quiescent* reliable communication in the presence of process crashes and lossy links—a problem that cannot be solved without failure detection.
2. We show that the weakest failure detector with bounded output size that can be used to solve this problem is $\diamond\mathcal{P}$—which is not implementable.
3. To overcome this obstacle, we introduce $\mathcal{HB}$: this failure detector can be used to achieve quiescent reliable communication, and it *is* implementable. In contrast to common failure detectors [3, 12, 20, 21, 28, 34], $\mathcal{HB}$ does not output a list of suspects, and it can be implemented without timeouts.
4. We show that $\mathcal{HB}$ can be used to extend existing algorithms for many fundamental problems (e.g., consensus, atomic broadcast, k-set agreement, atomic commitment, and approximate agreement) to tolerate message losses. It can also be used to extend the results of [5].

Result (2) above implies that failure detectors with bounded output size are either (a) too weak to achieve quiescent reliable communication, or (b) not implementable. Thus, failure detectors that output lists of processes, which are commonly used in practice, are not always the best ones to solve a problem: their power or applicability is limited. To the best of our knowledge, this is the first work that shows that failure detectors with bounded output size have inherent limitations.

The problem of achieving reliable communication despite failures has been extensively studied, especially in the context of data link protocols (see Chapter 22 of [29] for a compendium). Our work differs from previous results because we seek *quiescent* algorithms in systems where processes *and* links can fail (and this requires the use of unreliable failure detectors). The works that are the closest to ours are due to Moses and Roth [30] and Basu, Charron-Bost, and Toueg [5]. The main goal of [30] is to achieve quiescent reliable communication with algorithms that garbage-collect

[3]The link failure model in [5] is slightly different from the one used here (cf. section 10).

old messages in systems with lossy links (the issue of garbage collection is only briefly considered here). The algorithms in [30], however, are not resilient to process crashes. The protocols in [5] tolerate both process crashes and lossy links, but they are not quiescent (and they do not use failure detectors). In section 10, we use $\mathcal{HB}$ to extend the results of [5] and obtain quiescent protocols.

The paper is organized as follows. Our model is given in section 2. In section 3, we define the reliable communication primitives that we focus on. In section 4, we show that, without failure detectors, quiescent reliable communication is impossible. In section 5, we prove that $\diamond\mathcal{P}$ is the weakest failure detector with bounded output size that can be used to solve this problem (this proof is under a simplifying assumption; the proof without this assumption is given in the appendix). We then define the heartbeat failure detector $\mathcal{HB}$ in section 6. In section 7, we show how to use $\mathcal{HB}$ to achieve quiescent reliable communication. In section 8, we show how to implement $\mathcal{HB}$. In section 9, we consider two stronger types of communication primitives and give quiescent implementations that use $\mathcal{HB}$. In section 10, we explain how $\mathcal{HB}$ can be used to extend several previous results. We conclude the paper with some remarks about message buffering, quiescence versus termination, models of lossy links, and the generalization of our results to partitionable networks.

2. Model.

We consider asynchronous message-passing distributed systems in which there are no timing assumptions. In particular, we make no assumptions on the time it takes to deliver a message, or on relative process speeds. Processes can communicate with each other by sending messages through unidirectional links. We do not assume that the network is completely connected or that the links are bidirectional. The system can experience both process failures and link failures. Processes can fail by crashing, and links can fail by dropping messages. The model, based on the one in [11], is described next.

A network is a directed graph $G = (\Pi, \Lambda)$ where $\Pi = \{1, \dots, n\}$ is the set of processes, and $\Lambda \subseteq \Pi \times \Pi$ is the set of links. If there is a link from process p to process q, we denote this link by $p \rightarrow q$, and if, in addition, $q \neq p$, we say that q is a *neighbor* of p. The set of neighbors of p is denoted by $\mathit{neighbor}(p)$.

We assume the existence of a discrete global clock—this is merely a fictional device to simplify the presentation, and processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.1. Failures and failure patterns.

Processes can fail by crashing, i.e., by halting prematurely. A *process failure pattern* F_P is a function from $\mathcal{T}$ to 2^{Π}. Intuitively, $F_P(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover," i.e., for all t: $F_P(t) \subseteq F_P(t+1)$. We say p *crashes in* F_P if $p \in F_P(t)$ for some t; otherwise we say p *is correct in* F_P.

Some links in the network are fair. Roughly speaking, a fair link $p \rightarrow q$ may intermittently drop messages and may do so infinitely often, but it must satisfy the following "fairness" property: if p repeatedly sends some message to q and q does not crash, then q eventually receives that message. Link properties are made precise in section 2.5.

A *link failure pattern* F_L is a subset of the set of links Λ. Intuitively, F_L is the set of links that may fail to satisfy the above fairness property. If $p \rightarrow q \notin F_L$, we say that $p \rightarrow q$ *is fair in* F_L.

A *failure pattern* $F = (F_P, F_L)$ combines a process failure pattern and a link failure pattern, and $\mathit{correct\_proc}(F)$ and $\mathit{crashed\_proc}(F)$ denote the set of processes that are correct and crashed in F_P, respectively.

2.2. Network connectivity. The following definitions are with respect to a given failure pattern $F = (F_P, F_L)$. We say that a path $(p_1, \dots, p_k)$ in the network is *fair* if processes $p_1, \dots, p_k$ are correct and links $p_1 \to p_2, \dots, p_{k-1} \to p_k$ are fair. We assume that every pair of distinct correct processes is connected through a fair path. This precludes permanent network partitions.

2.3. Failure detectors. Each process has access to a local failure detector module that provides (possibly incorrect) information about the failure pattern that occurs in an execution. A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $H(p, t)$ is the output value of the failure detector module of process p at time t. *A failure detector $\mathcal{D}$* is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ (where $\mathcal{R}_{\mathcal{D}}$ denotes the range of the output of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

We now define the *eventually perfect failure detector* $\diamond\mathcal{P}$ [12].[4] Each failure detector module of $\diamond\mathcal{P}$ outputs a *set of processes* that are suspected to have crashed, i.e., $\mathcal{R}_{\diamond\mathcal{P}} = 2^{\Pi}$. For each failure pattern F, $\diamond\mathcal{P}(F)$ is the set of all failure detector histories H with range $\mathcal{R}_{\diamond\mathcal{P}}$ that satisfy the following properties.

1. *Strong completeness*. Eventually, every process that crashes is permanently suspected by every correct process. More precisely,

$$\exists t \in \mathcal{T}\, \forall p \in \mathit{crashed\_proc}(F),\, \forall q \in \mathit{correct\_proc}(F),\, \forall t' \geq t : p \in H(q, t').$$

2. *Eventual strong accuracy*. There is a time after which correct processes are not suspected by any correct process. More precisely,

$$\exists t \in \mathcal{T}\ \forall t' \geq t,\ \forall p, q \in \mathit{correct\_proc}(F) : p \notin H(q, t').$$

Sometimes we need to consider systems without failure detectors. For convenience, we model such systems by assuming that their failure detectors always output nil. More precisely, the *nil failure detector* $\mathcal{D}_{\perp}$ is the one where the failure detector modules of all processes always output $\perp$, independent of the failure pattern. A *system without failure detectors* is one whose failure detector is $\mathcal{D}_{\perp}$.

2.4. Runs of algorithms. An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in atomic *steps* of A. In each step, a process may receive a message from a process, get an external input, query its failure detector module, undergo a state transition, send a message to a neighbor, and issue an external output.

A *run of algorithm A using failure detector $\mathcal{D}$* is a tuple $R = (F, H_{\mathcal{D}}, I, S, T)$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a history of failure detector $\mathcal{D}$ for failure pattern F, I is an initial configuration of A, S is an infinite sequence of steps of A, and T is an infinite list of increasing time values indicating when each step in S occurs.

A run must satisfy the following properties for every process p. If p has crashed by time t, i.e., $p \in F_P(t)$, then p does not take a step at any time $t' \geq t$; if p is correct, i.e., $p \in \mathit{correct\_proc}(F)$, then p takes an infinite number of steps; if p takes a step at time t and queries its failure detector, then p gets $H_{\mathcal{D}}(p, t)$ as a response.

2.5. Link properties. Each run $R = (F, H_{\mathcal{D}}, I, S, T)$ must also satisfy some "link properties." First, no link creates or duplicates messages. More precisely, for every link $p \to q \in \Lambda$, the following must hold.

[4]In [12], $\diamond\mathcal{P}$ denotes a *class* of failure detectors.

1. *Uniform integrity*. For all $k \geq 1$, if q receives a message m from p exactly k times by time t, then p sent m to q at least k times before time t.

Moreover, every fair link transports any message that is repeatedly sent through it. More precisely, for every link $p \to q \notin F_L$, the following must hold.

2. *Fairness*. If p sends a message m to q an infinite number of times and q is correct, then q receives m from p an infinite number of times.

Note that any link, whether fair or not, may lose (or not lose) messages arbitrarily during any finite period of time. In particular, a fair link may lose all the messages sent during any finite period of time, while a link that is not fair may behave perfectly during that time.

2.6. Environments and problem solving. The correctness of an algorithm may depend on certain assumptions on the "environment," e.g., the maximum number of processes that may crash. For example, a consensus algorithm may need the assumption that a majority of processes is correct. Formally, an *environment* $\mathcal{E}$ is a set of failure patterns. Unless otherwise stated, the only restriction that we put on the environment in this paper is that every pair of distinct correct processes is connected through a fair path.

A problem P is defined by properties that sets of runs must satisfy. An algorithm A solves problem P using a failure detector $\mathcal{D}$ in environment $\mathcal{E}$ if the set of all runs $R = (F, H_{\mathcal{D}}, I, S, T)$ of A using $\mathcal{D}$, where $F \in \mathcal{E}$ satisfies the properties required by P.

Let $\mathcal{C}$ be a class of failure detectors. An algorithm A solves a problem P using $\mathcal{C}$ in environment $\mathcal{E}$ if for all $\mathcal{D} \in \mathcal{C}$, A solves P using $\mathcal{D}$ in $\mathcal{E}$. An algorithm implements $\mathcal{C}$ in environment $\mathcal{E}$ if it implements some $\mathcal{D} \in \mathcal{C}$ in $\mathcal{E}$.

3. Quiescent reliable communication. In this paper, we focus on quasi-reliable send and receive, and reliable broadcast, because these communication primitives are sufficient to solve many problems (see section 10.1). Stronger types of communication primitives—reliable send and receive, and uniform reliable broadcast—are briefly considered in section 9.

3.1. Quasi-reliable send and receive. Consider any two distinct processes s and r. We define *quasi-reliable send and receive from s to r* in terms of two primitives, $\mathsf{qr\text{-}send}_{s,r}$ and $\mathsf{qr\text{-}receive}_{r,s}$. We say that process s $\mathsf{qr\text{-}sends}$ *message m to process r* if s invokes $\mathsf{qr\text{-}send}_{s,r}(m)$. We assume that if s is correct, it eventually returns from this invocation. We allow process s to $\mathsf{qr\text{-}send}$ the same message m more than once through the same link. We say that process r $\mathsf{qr\text{-}receives}$ *message m from process s* if r returns from the invocation of $\mathsf{qr\text{-}receive}_{r,s}(m)$. Primitives $\mathsf{qr\text{-}send}_{s,r}$ and $\mathsf{qr\text{-}receive}_{r,s}$ satisfy the following properties.

1. *Uniform integrity*. For all $k \geq 1$, if r $\mathsf{qr\text{-}receives}$ m from s exactly k times by time t, then s $\mathsf{qr\text{-}sent}$ m to r at least k times before time t.

2. *Quasi no loss.*[5] For all $k \geq 1$, if both s and r are correct, and s $\mathsf{qr\text{-}sends}$ m to r exactly k times by time t, then r eventually $\mathsf{qr\text{-}receives}$ m from s at least k times.

Intuitively, quasi no loss together with uniform integrity implies that if s and r are correct, then r $\mathsf{qr\text{-}receives}$ m from s exactly as many times as s $\mathsf{qr\text{-}sends}$ m to r.

We want to implement quasi-reliable send and receive primitives using the communication service provided by the network links. Informally, such an implementation is *quiescent* if it sends only a finite number of messages when $\mathsf{qr\text{-}send}_{s,r}$ is invoked a

[5]A stronger property, called *No Loss*, is used in section 9.1 to define *reliable* send and receive.

finite number of times.[6]

3.2. Reliable broadcast. *Reliable broadcast* [9] is defined in terms of two primitives: broadcast(m) and deliver(m). We say that process p *broadcasts message* m if p invokes broadcast(m). We assume that every broadcast message m includes the following fields: the identity of its sender, denoted $sender(m)$, and a sequence number, denoted $seq(m)$. These fields make every message unique. We say that q *delivers message* m if q returns from the invocation of deliver(m). Primitives broadcast and deliver satisfy the following properties [23].

1. *Validity*. If a correct process broadcasts a message m, then it eventually delivers m.

2. *Agreement*. If a correct process delivers a message m, then all correct processes eventually deliver m.

3. *Uniform integrity*. For every message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

Validity and agreement imply that if a correct process broadcasts a message m, then all correct processes eventually deliver m.

We want to implement reliable broadcast using the communication service provided by the network links. Informally, such an implementation is *quiescent* if it sends only a finite number of messages when broadcast is invoked a finite number of times.

3.3. Relating reliable broadcast and quasi-reliable send and receive. From a quiescent implementation of quasi-reliable send and receive one can easily obtain a quiescent implementation of reliable broadcast, and vice-versa.

Remark 3.1. From any quiescent implementation of reliable broadcast, we can obtain a quiescent implementation of the quasi-reliable primitives $\text{qr-send}_{p,q}$ and $\text{qr-receive}_{q,p}$ for every pair of processes p and q.

Remark 3.2. Suppose that every pair of correct processes is connected through a path of correct processes. If we have a quiescent implementation of quasi-reliable primitives $\text{qr-send}_{p,q}$ and $\text{qr-receive}_{q,p}$ for all processes p and $q \in neighbor(p)$, then we can obtain a quiescent implementation of reliable broadcast.

To implement reliable broadcast from qr-send and qr-receive, one can use a simple diffusion algorithm (e.g., see [23]).

4. Impossibility of quiescent reliable communication. We now show that in a system without failure detectors, quiescent reliable communication cannot be achieved. This holds even if the network is completely connected and only a finite number of messages can be lost.

THEOREM 4.1. *Consider a system without failure detectors where every pair of processes is connected by a fair link and at most one process may crash. Let s and r be any two distinct processes. There is no quiescent implementation of quasi-reliable send and receive from s to r. This holds even if we assume that only a finite number of messages can be lost.*

Proof.[7] Assume, by contradiction, that there exists a quiescent implementation I of quasi-reliable $\text{qr-send}_{s,r}$ and $\text{qr-receive}_{r,s}$. The basic intuition behind the proof

[6] A quiescent implementation of $\text{qr-send}_{s,r}$ and $\text{qr-receive}_{r,s}$ is allowed to send a finite number of messages even if no $\text{qr-send}_{s,r}$ is invoked at all (e.g., some messages may be sent as part of an "initialization phase").

[7] This theorem is actually a corollary of Theorem A.12 and the fact that the eventually perfect failure detector $\diamond\mathcal{P}$ cannot be implemented. The proof of Theorem A.12, however, uses some complex arguments that obscure the intuition behind Theorem 4.1. We prefer to give a self-contained and direct proof that does not use Theorem A.12.

is to construct a run R_1 where s qr-sends a message to r, but r crashes. Since the implementation of qr-send and qr-receive is quiescent, only a finite number of messages are sent to r in R_1. We then construct a similar run R_2 where s qr-sends a message to r, r does not crash, but the finite number of messages sent to r are lost. Runs R_1 and R_2 are indistinguishable from the point of view of r, so r never qr-receives the message—a contradiction. It turns out that to construct run R_1, we need another run R_0. This is because we allow the quiescent implementation of qr-send and qr-receive to send a finite number of "initialization" messages (see footnote 6). We now describe runs R_0, R_1, and R_2 in more detail.

In run R_0, s qr-sends no messages, all processes are correct, processes take steps in round-robin fashion, and every time a process takes a step, it receives the earliest message sent to it that it did not yet receive. Since I is quiescent, there is a time t_0 after which no messages are sent or received. By the uniform integrity property of qr-send and qr-receive, process r never qr-receives any message.

Run R_1 is identical to run R_0 up to time t_0; at time $t_0 + 1$, s qr-sends M to r, and r crashes; after time $t_0 + 1$, no processes crash, and every time a process takes a step, it receives the earliest message sent to it that it did not yet receive. Since I is quiescent, there is a time $t_1 > t_0$ after which no messages are sent or received.

In run R_2, r behaves exactly as in run R_0 (in particular, r does not crash and r receives a message m in R_2 whenever it receives m in R_0); all other processes behave exactly as in run R_1 (in particular, a process $p \neq r$ receives a message m in R_2 whenever it receives m in R_1). Note that, in R_2, if messages are sent to or from r after time t_0, then they are never received.

We now show that in R_2 all links satisfy the uniform integrity property. Assume that for some $k \geq 1$, some process q receives m from some process p k times by time t. There are several cases. (1) If $q = r$, then r receives m from p k times in R_0 by time t (since r behaves in the same way in R_0 and R_2). In R_0, by the uniform integrity property of the links, p sends m to r at least k times before time t. This happens by time t_0, since there are no sends in R_0 after time t_0. Note that by time t_0, p behaves exactly in the same way in R_0, R_1, and R_2. Thus p sends m to r at least k times before time t in R_2. (2) If $q \neq r$ and $p = r$, then q receives m from r k times in R_1 by time t (since q behaves in the same way in R_1 and R_2). In R_1, by the uniform integrity property of the links, r sends m to q at least k times before time t. This happens by time t_0, since r crashes at time t_0+1 in R_1. By time t_0, r behaves exactly in the same way in R_0, R_1, and R_2. Thus r sends m to q at least k times before time t in R_2. (3) If $q \neq r$ and $p \neq r$, then q receives m from p k times in R_1 by time t (since q behaves in the same way in R_1 and R_2). In R_1, by the uniform integrity property of the links, p sends m to q at least k times before time t. Note that p behaves exactly in the same way in R_1 and R_2. Thus p sends m to q at least k times in R_2 before time t. Therefore, in R_2 all links satisfy the uniform integrity property.

We next show that in R_2 all links satisfy the fairness property, and in fact only a finite number of messages are lost. Note that r sends only a finite number of messages in R_0 (since it does not send messages after time t_0), and every process $p \neq r$ sends only a finite number of messages in R_1 (since it does not send messages after time t_1). So, by construction of R_2, all processes send only a finite number of messages in R_2. Therefore, only a finite number of messages are lost, and in R_2 all links satisfy the fairness property.

We conclude that R_2 is a possible run of I in a network with fair links that lose only a finite number of messages. Note that in R_2: (a) both s and r are correct;

(b) s qr-sends M to r; and (c) r does not qr-receive M. This violates the quasi no loss property of qr-send$_{s,r}$ and qr-receive$_{r,s}$, and so I is not an implementation of qr-send$_{s,r}$ and qr-receive$_{r,s}$—a contradiction. □

Theorem 4.1 and Remark 3.1 immediately imply the following corollary.

COROLLARY 4.2. *There is no quiescent implementation of reliable broadcast in a network where a process may crash and links may lose a finite number of messages.*

The above results show that quiescent reliable communication cannot be achieved in a system without failure detectors. The rest of this paper explores the use of failure detectors to solve this problem.

5. The weakest failure detector with bounded output size for quiescent reliable communication.

In practice, and in much of the previous literature, the output of a failure detector is just a set of processes suspected to have failed. One such failure detector, namely $\diamond\mathcal{P}$, can be used to achieve quiescent reliable communication. However, $\diamond\mathcal{P}$ is not implementable in asynchronous systems. Can we achieve quiescent reliable communication with a failure detector that outputs a set of suspects and is implementable?

In this section we show that the answer is no. In fact, we prove a stronger result: Among all failure detectors *with bounded output size* (these include all failure detectors that output a set of suspects), the weakest one for achieving quiescent reliable communication is $\diamond\mathcal{P}$—which is not implementable. In contrast, if we do not bound the output size, quiescent reliable communication can be solved with $\mathcal{HB}$—which is implementable. This shows that failure detectors with bounded output size have some inherent limitations.

We prove our result with respect to a problem that we call *single-shot reliable send and receive.* This problem is weaker than quasi-reliable send and receive, and reliable broadcast, and thus our result immediately applies to those problems as well.

In section 5.1, we explain what it means for a failure detector to be weaker than another one. In section 5.2, we define the single-shot reliable send and receive problem. We then proceed to prove our main result under some reasonable simplifying assumption. We first give a rough outline of this proof (section 5.3) and then the proof itself (sections 5.4 and 5.5). In the appendix, we give the full proof without the simplifying assumption.

5.1. Failure detector transformations.

Failure detectors can be compared via algorithmic transformations [12, 11]. A transformation algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses failure detector $\mathcal{D}$ to emulate $\mathcal{D}'$, as we now explain. At each process p, the algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ maintains a variable $\mathcal{D}'_p$ that emulates the output of $\mathcal{D}'$ at p. Let $H_{\mathcal{D}'}$ be the history of all the $\mathcal{D}'$ variables in a run R of $T_{\mathcal{D}\to\mathcal{D}'}$, i.e., $H_{\mathcal{D}'}(p,t)$ is the value of $\mathcal{D}'_p$ at time t in run R. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ *in environment* $\mathcal{E}$ if and only if for every $F \in \mathcal{E}$ and every run $R = (F, H_{\mathcal{D}}, I, S, T)$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, we have $H_{\mathcal{D}'} \in \mathcal{D}'(F)$. Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ provides at least as much information about process failures as $\mathcal{D}'$ does, and we say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$.

Note that, in general, $T_{\mathcal{D}\to\mathcal{D}'}$ need not emulate *all* the failure detector histories of $\mathcal{D}'$ (in environment $\mathcal{E}$); what we do require is that all the failure detector histories it emulates be histories of $\mathcal{D}'$ (in that environment).

5.2. Single-shot reliable send and receive.

The single-shot reliable send and receive problem is defined in terms of two communication primitives, called s-send and s-receive. Each process can s-send a single bit once to one process of its choice, if it wishes to do so (but it is also possible that no process in the system ever s-sends any bit). The s-send and s-receive primitives must satisfy the following property. For any two correct processes p and q, and any $b \in \{0, 1\}$, p s-sends b to q if and only if q s-receives b from p.

An implementation $\mathcal{I}$ of s-send and s-receive is quiescent if it sends only a finite number of messages throughout the network.

5.3. Intuitive overview of the simple proof. Let $\mathcal{D}$ be a failure detector with bounded output size, i.e., the range of $\mathcal{D}$ is finite. Suppose $\mathcal{D}$ can be used to solve the single-shot reliable send and receive problem with a quiescent algorithm $\mathcal{I}$ ($\mathcal{I}$ is also called the implementation of s-send and s-receive). We show that $\mathcal{D}$ can be transformed to $\diamond\mathcal{P}$.

The proof that follows makes the simplifying assumption that $\mathcal{I}$ does not have an "initialization phase" that requires the sending of messages. In other words, we assume that $\mathcal{I}$ is such that if no process ever s-sends any bit, then no process ever sends any messages. This reasonable assumption allows us to simplify the proof and illustrate the basic ideas. In the appendix, we give the full proof.

Since the range of $\mathcal{D}$ is finite, then for every failure detector history H of $\mathcal{D}$: (a) each failure detector module outputs some values infinitely often (these are the "limit values"), and (b) there is a time after which it outputs only limit values. Let v be a limit value for process p and H. A crucial observation is that with H it is possible to construct runs such that whenever p takes a step it always gets v from its failure detector module. It is easy to generalize the notion of a limit value for p to a limit vector for a set of processes P: A vector f (with a value for every process in the system) is a limit vector for P and H if, for each process p in P, the failure detector module of p outputs $f(p)$ infinitely often in H. Note that with H it is possible to construct runs such that whenever a process p in P takes a step, it obtains $f(p)$ from its failure detector module. We say that vector f hints that P is the set of all correct processes, if f could occur as a limit vector for P when P is the set of correct processes (more precisely, f is a limit vector for P in a history $H \in \mathcal{D}(F)$ where $\mathit{correct\_proc}(F) = P$).

Consider a failure detector history H that can occur when P is the set of all correct processes. Let f be any limit vector for P and H. Clearly, f hints that P is the set of all correct processes. Can f also hint that a proper subset P' of P is the set of all correct processes? The answer is no. As we argue next, this is because with $\mathcal{D}$, a process in P' should be able to s-send a bit to a process q in $P \setminus P'$ and to do so quiescently using $\mathcal{I}$.

Suppose, for contradiction, that f hints that P' is the set of all correct processes. Then we can construct a run R_1 of $\mathcal{I}$ where (a) P' is indeed the set of all correct processes, (b) processes in P' are scheduled such that whenever they take steps they get f from their failure detector module, (c) some process p in P' s-sends a bit b to some process q in $P \setminus P'$, and (d) processes in $P \setminus P'$ never take a step. Because the implementation $\mathcal{I}$ is quiescent, in R_1 eventually all processes in P' (including p) stop sending messages—they give up on trying to transmit b to q.

Since f also hints that P is the set of correct processes, we can create another run R_2 of $\mathcal{I}$ where (a) P is the set of correct processes, (b) processes in P are scheduled such that whenever they take steps they get f from their failure detector module, (c) p s-sends b to q, and (d) messages sent between processes in P' and processes in $P \setminus P'$ are lost. Note that from the point of view of processes in P', run R_2 is indistinguishable from run R_1. Thus in R_2 eventually all processes in P' stop sending messages—they give up on trying to transmit b to q. So, in R_2 process q never receives any messages, and thus it does not s-receive b from p. Since p and q are correct in R_2, the implementation $\mathcal{I}$ of s-send and s-receive is incorrect—a contradiction. Thus f cannot hint that P' is the set of all correct processes.

Let E_P be the set of all vectors that hint that P is the set of correct processes (this set is determined by $\mathcal{D}$). The algorithm that transforms $\mathcal{D}$ to $\diamond\mathcal{P}$ uses a predetermined "table of hints" containing, for each possible P, the set E_P.

The transformation algorithm works as follows. Each process p periodically sends its current failure detector output to every process and maintains two variables: f and *Order*. Vector f stores the last failure detector value received from each process, and *Order* is an ordered set of processes. Whenever p receives a failure detector value from another process q, it records that value in $f(q)$ and moves q to the front of *Order*. Let P be the set of correct processes in this run. Note that (a) eventually f is a limit vector for P, and (b) the correct processes percolate to the front of *Order* (processes that crash end up at the tail), so that eventually P is some prefix of *Order*.

To satisfy the properties of $\diamond\mathcal{P}$, p must eventually output the complement of P. By (b) above, eventually P is the largest prefix of *Order* that contains correct processes. To find this maximal prefix, p repeatedly uses its current value of f and the predetermined table of hints, as follows. For each prefix P' of *Order*, in order of increasing size, p checks if f hints that P' is the set of all correct processes, i.e., $f \in E_{P'}$, and if so p outputs the complement of P'. This works because, as we argued above, any limit vector f for P: (1) hints that P is the set of all correct processes, and (2) cannot hint that a proper subset P' of P is the set of all correct processes. This concludes the overview of the proof (the reader should understand why the argument above breaks down without the simplifying assumption).

We next give the actual proof. The transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ uses a table which is determined a priori from $\mathcal{D}$ (this is the "table of hints" in our intuitive explanation). We first define this table and show some of its properties (section 5.4). We then describe and prove the correctness of the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ that uses this table (section 5.5).

5.4. The predetermined table. Let $\mathcal{E}$ be an environment and $\mathcal{D}$ be any failure detector with finite range $\mathcal{R} = \{v_1, v_2, \dots, v_\ell\}$. Let $\mathcal{I}$ be a quiescent implementation of s-send and s-receive that uses $\mathcal{D}$ in environment $\mathcal{E}$. Assume that if no process s-sends any bit, then $\mathcal{I}$ does not send any messages (this simplifying assumption is removed in the appendix).

Given $v_j \in \mathcal{R}$, a process $p \in \Pi$, and a failure detector history H with range $\mathcal{R}$, we say that v_j is *a limit value for p and H* if, for infinitely many t, $H(p,t) = v_j$. Let f be an assignment of failure detector values to every process in Π, i.e., $f : \Pi \longrightarrow \mathcal{R}$. Let P be a nonempty set of processes. We say that *f is a limit vector for P and H* if for all $p \in P$, $f(p)$ is a limit value for p and H. The set of all limit vectors for P and H is denoted $L_P(H)$. Let $E_P^{\mathcal{D},\mathcal{E}} = \{f \mid \exists F \in \mathcal{E}, \exists H \in \mathcal{D}(F) : P = \mathit{correct\_proc}(F) \text{ and } f \in L_P(H)\}$. Roughly speaking, $E_P^{\mathcal{D},\mathcal{E}}$ is the set of limit vectors that could occur when P is the set of correct processes.

The table used by the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ consists of all the sets $E_P^{\mathcal{D},\mathcal{E}}$ where P ranges over all nonempty subsets of processes. Note that this table is finite. We omit the superscript $\mathcal{D},\mathcal{E}$ from $E_P^{\mathcal{D},\mathcal{E}}$ whenever it is clear from the context.

LEMMA 5.1. *Let $F \in \mathcal{E}$, $P = \mathit{correct\_proc}(F)$, and $H \in \mathcal{D}(F)$. Assume $P \neq \emptyset$. If $f \in L_P(H)$, then $f \in E_P$ and $f \notin E_{P'}$ for every P' such that $\emptyset \subset P' \subset P$.*

Proof. Let $f \in L_P(H)$. The fact that $f \in E_P$ is immediate from the definition of

E_P. Let P' be such that $\emptyset \subset P' \subset P$. Suppose, for contradiction, that $f \in E_{P'}$. Then there exists a failure pattern $F' \in \mathcal{E}$ and $H' \in \mathcal{D}(F')$ such that $P' = \mathit{correct\_proc}(F')$ and $f \in L_{P'}(H')$.

We now obtain a contradiction by using the quiescent implementation $\mathcal{I}$ of s-send and s-receive. Let p be a process in P' and q be a process in $P \setminus P'$. We construct two runs, R_1 and R_2 of $\mathcal{I}$ using $\mathcal{D}$, as follows.

1. Run R_1 has failure pattern F' and failure detector history H'. Initially p s-sends some bit b to q. Processes in P' take steps and those in $\Pi \setminus P'$ do not. Processes in P' take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (since $f \in L_{P'}(H')$, $f(r)$ is a limit value for r and H'). Moreover, every process in P' receives every message sent to it.

Since $\mathcal{I}$ is quiescent, there is a time t_1 after which no messages are sent or received. Assume without loss of generality that at time t_1 all processes in P' took the same number k of steps (otherwise, choose another time $t'_1 > t_1$). Note that all messages in R_1 are sent within the finite period of time $[0, t_1]$. Thus the fact that all processes in P' receive all the messages sent to them is consistent with the link failure pattern of F' (even if in F' some of the links are not fair).

2. Run R_2 has failure pattern F and failure detector history H. Initially, processes in R_2 behave as in R_1: p s-sends some bit b to q; moreover, each process in P' takes the same k steps as in R_1, and processes in $\Pi \setminus P'$ do not take any steps. More precisely, processes in P' take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (since $f \in L_P(H)$ and $r \in P' \subset P$, $f(r)$ is a limit value for r and H). Moreover, every process in P' receives every message sent to it, and all messages sent to processes in $\Pi \setminus P'$ are lost. This goes on until each process in P' takes k steps, exactly as in R_1.[8] Let t_2 be the time when this happens. After t_2, processes in P take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (it does not matter what a process $r \in P \setminus P'$ gets from its failure detector module, as long as it is compatible with H). Moreover, after t_2 no process s-sends any bit. This completes the description of run R_2.

In R_2, at time t_2, each process in P' is in the same state as in run R_1 at time t_1. Moreover, each process in $P \setminus P'$ is in its initial state. By a simple induction argument we can show that after time t_2 in R_2, (a) processes in P' continue to behave as in R_1, (b) processes in $P \setminus P'$ behave as if they were in a run of $\mathcal{I}$ in which no process ever s-sends any bit, and (c) no process sends any message (this induction uses the simplifying assumption that in a run in which there are no s-sends, no process sends any message). Therefore, in R_2, process q (which is in $P \setminus P'$) never receives any messages. This implies that q does not s-receive b from p.

Note that in R_2: (a) both p and q are correct; (b) p s-sends b to q; and (c) q does not s-receive b from p. Thus $\mathcal{I}$ is not a correct implementation of s-send and s-receive—a contradiction. ∎

5.5. The transformation algorithm. The algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}' = \diamond\mathcal{P}$ in environment $\mathcal{E}$ is shown in Figure 5.1. $T_{\mathcal{D}\to\mathcal{D}'}$ uses the table of sets E_P (for all nonempty subsets of processes P) that has been determined a priori from the given $\mathcal{D}$ and $\mathcal{E}$. It also uses an implementation

[8]This behavior of the links is consistent with F because for any finite period of time, any link (whether fair or not in F) may lose or not lose any message.

```
For every process $p$:

    Initialization:
        for all $q \in \Pi$ do $f[q] \leftarrow \perp$
        $Order \leftarrow \emptyset$
        $\mathcal{D}'_p \leftarrow \emptyset$
        { For each $\emptyset \subset P \subseteq \Pi$, the set $E_P^{\mathcal{D},\mathcal{E}}$ is determined a priori from $\mathcal{D}$ and $\mathcal{E}$ }

    cobegin
        || Task 1:
            repeat periodically
                $v \leftarrow \mathcal{D}_p$                                              {query $\mathcal{D}$}
                for all $q \in \Pi$ do qr-send $v$ to $q$

        || Task 2:
            upon qr-receive $w$ from $q$ do {upon receipt of a failure detector value from $q$}
                $f[q] \leftarrow w$
                $Order \leftarrow q \mid\mid (Order \setminus \{q\})$     {process $q$ is moved to the front of $Order$}
                if for some $k \geq 1$, $f \in E_{Order[1..k]}^{\mathcal{D},\mathcal{E}}$ then
                    let $k_0$ be the smallest such $k$
                    $\mathcal{D}'_p \leftarrow \Pi \setminus Order[1..k_0]$         {suspect processes not in $Order[1..k_0]$}
    coend
```

FIG. 5.1. *Transformation of $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}'$ in environment $\mathcal{E}$.*

of qr-send and qr-receive between every pair of processes. A simple implementation is by repeated retransmissions and diffusion (it does not have to be quiescent).

All variables are local to each process. Vector f stores the last failure detector value that p qr-received from each process; *Order* is an ordered set that records the order in which the last failure detector value from each process was qr-received; $\mathcal{D}'_p$ denotes the output of the eventually perfect failure detector that p is simulating (a set of processes that p currently suspects).

In Task 1, each process p periodically qr-sends the output of its failure detector module $\mathcal{D}_p$ to every process q. Upon the qr-receipt of a failure detector value from process q in Task 2, process p enters it into $f[q]$ and moves q to the front of *Order*. Then p checks if there is some prefix $Order[1..k]$ of *Order* such that $f \in E_{Order[1..k]}$. If there is, it sets $\mathcal{D}'$ to the complement of the smallest such prefix.

We now show that the failure detector constructed by this algorithm, namely $\mathcal{D}'$, is an eventually perfect failure detector. Consider a run of this algorithm with failure pattern $F \in \mathcal{E}$ and failure detector history $H \in \mathcal{D}(F)$, such that $correct\_proc(F) \neq \emptyset$. Let t be the number of processes that crash in F, i.e., $t = |\Pi \setminus correct\_proc(F)|$. Henceforth, p denotes a correct process in F, and variables f and *Order* are local to p.

LEMMA 5.2. *There is a time after which* (1) $Order\ [1..n-t] = correct\_proc(F)$, *and* (2) $f \in L_{Order[1..n-t]}(H)$.[9]

Proof. Part (1) is clear from the way *Order* is updated, the fact that p keeps qr-receiving failure detector values from every correct process, and the fact that p

[9]This does not mean that eventually the values of variables f and *Order* at p stop changing. It means that, although they may continue to change forever, eventually the predicates (1) and (2) are true forever at p.

eventually stops qr-receiving messages from processes that crash. Part (2) of the lemma follows from part (1) and the fact that the range $\mathcal{R}$ of $\mathcal{D}$ is finite. ∎

COROLLARY 5.3. *There is a time after which* (1) $f \in E_{Order[1..n-t]}$, *and* (2) *for all* $1 \leq k < n-t$, $f \notin E_{Order[1..k]}$.

Proof. By Lemma 5.2, there is a time t_0 after which $f \in L_{Order[1..n-t]}(H)$ and $Order[1..n-t] = correct\_proc(F)$. So after time t_0, by Lemma 5.1, $f \in E_{Order[1..n-t]}$. This shows (1). Let k be such that $1 \leq k < n-t$. After t_0, $\emptyset \subset Order[1..k] \subset correct\_proc(F)$, and $f \in L_{correct\_proc(F)}(H)$. So, by Lemma 5.1, $f \notin E_{Order[1..k]}$. This shows (2). ∎

COROLLARY 5.4. *There is a time after which* $\mathcal{D}'_p = \Pi \setminus correct\_proc(F)$.

Proof. By Corollary 5.3 and the algorithm, there is a time after which the k_0 selected in line 20 is always $n-t$. Now apply Lemma 5.2 part (1). ∎

By Corollary 5.4, we have the following theorem.

THEOREM 5.5. *Consider an asynchronous system subject to process crashes and message losses. Suppose failure detector $\mathcal{D}$ with finite range can be used to solve the single-shot reliable send and receive problem in environment $\mathcal{E}$, and that the implementation is quiescent. Assume further that if no process ever* s-sends *any bit, then this implementation does not send any messages. Then $\mathcal{D}$ can be transformed (in environment $\mathcal{E}$) to the eventually perfect failure detector $\diamond\mathcal{P}$.*

Theorems 5.5 and A.12 imply that if we restrict ourselves to failure detectors that output a set of suspects, we cannot achieve quiescent reliable communication with a failure detector that can be implemented. Thus we next introduce $\mathcal{HB}$, a failure detector that does *not* output a set of suspects. $\mathcal{HB}$ can be used to achieve quiescent reliable communication and it is implementable.

6. Definition of $\mathcal{HB}$.

A *heartbeat failure detector* $\mathcal{D}$ has the following features. The output of $\mathcal{D}$ at each process p is a list $(p_1, n_1), (p_2, n_2), \ldots, (p_k, n_k)$, where $p_1, p_2, \ldots, p_k$ are the neighbors of p, and each n_j is a nonnegative integer. Intuitively, n_j increases while p_j has not crashed, and stops increasing if p_j crashes. We say that n_j is *the heartbeat value of* p_j *at* p. The output of $\mathcal{D}$ at p at time t, namely $H(p,t)$, will be regarded as a vector indexed by the set $\{p_1, p_2, \ldots, p_k\}$. Thus $H(p,t)[p_j]$ is n_j. The *heartbeat sequence of* p_j *at* p is the sequence of the heartbeat values of p_j at p as time increases. $\mathcal{D}$ satisfies the following properties.

- *$\mathcal{HB}$-completeness*. At each correct process, the heartbeat sequence of every neighbor that crashes is bounded:

$$\forall F, \forall H \in \mathcal{D}(F), \forall p \in correct\_proc(F), \forall q \in crashed\_proc(F) \cap neighbor(p),$$
$$\exists K \in \boldsymbol{N}, \forall t \in \mathcal{T} : H(p,t)[q] \leq K.$$

- *$\mathcal{HB}$-accuracy*.
  - At each process, the heartbeat sequence of every neighbor is nondecreasing:

$$\forall F, \forall H \in \mathcal{D}(F), \forall p \in \Pi, \forall q \in neighbor(p), \forall t \in \mathcal{T} : H(p,t)[q] \leq H(p,t+1)[q].$$

  - At each correct process, the heartbeat sequence of every correct neighbor is unbounded:

$$\forall F, \forall H \in \mathcal{D}(F), \forall p \in correct\_proc(F), \forall q \in correct\_proc(F) \cap neighbor(p),$$
$$\forall K \in \boldsymbol{N}, \exists t \in \mathcal{T} : H(p,t)[q] > K.$$

The class of all heartbeat failure detectors is denoted $\mathcal{HB}$. By a slight abuse of notation, we sometimes use $\mathcal{HB}$ to refer to an arbitrary member of that class.

It is easy to generalize the definition of $\mathcal{HB}$ so that the failure detector module at each process p outputs the heartbeat of every process in the system [1], rather than just the heartbeats of the neighbors of p, but we do not need this generality here.

7. Quiescent reliable communication using $\mathcal{HB}$.

The communication networks that we consider are not necessarily completely connected, but we assume that every pair of correct processes is connected through a fair path. We first consider a simple type of such networks, in which every link is assumed to be bidirectional[10] and fair (Figure 7.1a). This assumption, a common one in practice, allows us to give efficient and simple algorithms. We then drop this assumption and treat a more general type of networks, in which some links may be unidirectional and/or not fair (Figure 7.1b). For both network types, we give quiescent reliable communication algorithms that use $\mathcal{HB}$. Our algorithms have the following feature: processes do not need to know the entire network topology or the number of processes in the system; they only need to know the identity of their neighbors.

In our algorithms, $\mathcal{D}_p$ denotes the current output of the failure detector $\mathcal{D}$ at process p.

7.1. The simple network case.

We assume that all links in the network are bidirectional and fair (Figure 7.1a). In this case, the algorithms are very simple. We first give a quiescent implementation of quasi-reliable qr-send$_{s,r}$ and qr-receive$_{r,s}$ for the case $r \in neighbor(s)$. For s to qr-send a message m to r, it repeatedly sends m to r every time the heartbeat of r increases, until s receives $ack(m)$ from r. Process r qr-receives m from s the first time it receives m from s, and r sends $ack(m)$ to s every time it receives m from s.

From this implementation and Remark 3.2, we can obtain a quiescent implementation of reliable broadcast. Then, from Remark 3.1, we can obtain a quiescent implementation of quasi-reliable send and receive for every pair of processes.

7.2. The general network case.

In this case (Figure 7.1b), some links may be unidirectional, e.g., the network may contain several unidirectional rings that intersect with each other. Moreover, some links may not be fair (and processes do not know which ones are fair).

Achieving quiescent reliable communication in this type of network is significantly more complex than before. For instance, suppose that we seek a quiescent implementation of quasi-reliable send and receive. In order for the sender s to qr-send a message m to the receiver r, it has to use a diffusion mechanism, even if r is a neighbor of s (since the link $s \to r$ may not be fair). Because of intermittent message losses, this diffusion mechanism needs to ensure that m is repeatedly sent over fair links. But when should this repeated send stop? One possibility is to use an acknowledgement mechanism. Unfortunately, the link in the reverse direction may not be fair (or may not even be part of the network), and so the acknowledgement itself has to be diffused. But diffusing the acknowledgements quiescently and reliably introduces a "chicken and egg" problem. We now explain how our algorithms avoid this problem.

We give a quiescent implementation of reliable broadcast in Figure 7.2. This implementation can be used to obtain quasi-reliable send and receive between every pair of processes (see Remark 3.1). For each message m that is broadcast, each process p maintains a variable $got_p[m]$ containing a set of processes. Intuitively, a process q

[10]In our model, this means that link $p \to q$ is in the network if and only if link $q \to p$ is in the network. In other words, $q \in neighbor(p)$ if and only if $p \in neighbor(q)$.

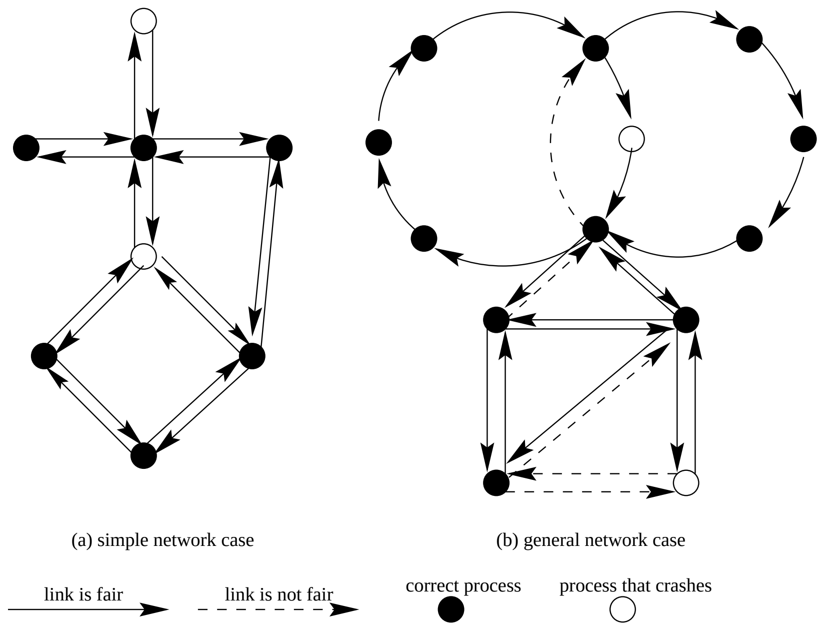


FIG. 7.1. *Examples of the simple and general network cases.*

is in $got_p[m]$ if p has evidence that q has delivered m. All the messages sent by a process p in the reliable broadcast algorithm are of the form $(m, got\_msg, path)$ where got_msg is the current value of $got_p[m]$, and $path$ is the sequence of processes that this copy of $(m, got\_msg, path)$ has traversed so far.

In order to reliably broadcast a message m, p first delivers m; then p initializes variable $got_p[m]$ to $\{p\}$ and forks task $diffuse(m)$; finally p returns from the invocation of broadcast(m). The task $diffuse(m)$ runs in the background. In this task, p periodically checks if, for some neighbor $q \notin got_p[m]$, the heartbeat of q at p has increased, and, if so, p sends $(m, got_p[m], p)$ to all neighbors whose heartbeat increased—even to those who are already in $got_p[m]$.[11] The task terminates when all neighbors of p are contained in $got_p[m]$.

Upon the receipt of a message $(m, got\_msg, path)$, process p first checks if it has already delivered m, and, if not, it delivers m and forks task $diffuse(m)$. Then p adds the contents of got_msg to $got_p[m]$ and appends itself to $path$. Finally, p forwards the new message $(m, got_p[m], path)$ to all its neighbors that appear at most once in $path$.

The code consisting of lines 19 through 27 is executed atomically.[12] Each concurrent execution of the $diffuse$ task (lines 9 to 17) has its own copy of all the local variables in this task.

We now outline the proof that, for the general network case, Figure 7.2 is a

[11]It may appear that p does not need to send this message to processes in $got_p[m]$, since they have already gotten m! But with this "optimization" the algorithm is no longer quiescent; in the proof of Lemma 7.8 we will indicate exactly where the sending to *every* neighbor whose heartbeat increased is necessary.

[12]A process p executes a region of code atomically if at any time there is at most one thread of p in this region.

```
1   For every process $p$:
2
3       To execute broadcast$(m)$:
4           deliver$(m)$
5           $got[m] \leftarrow \{p\}$
6           fork task $diffuse(m)$
7           return
8
9       task $diffuse(m)$:
10          for all $q \in neighbor(p)$ do $prev\_hb[q] \leftarrow -1$
11          repeat periodically
12              $hb \leftarrow \mathcal{D}_p$                    { query the heartbeat failure detector }
13              if for some $q \in neighbor(p)$, $q \notin got[m]$ and $prev\_hb[q] < hb[q]$ then
14                  for all $q \in neighbor(p)$ such that $prev\_hb[q] < hb[q]$ do
15                      send $(m, got[m], p)$ to $q$
16                  $prev\_hb \leftarrow hb$
17          until $neighbor(p) \subseteq got[m]$
18
19      upon receive $(m, got\_msg, path)$ from $q$ do
20          if $p$ has not previously executed deliver$(m)$ then
21              deliver$(m)$
22              $got[m] \leftarrow \{p\}$
23              fork task $diffuse(m)$
24          $got[m] \leftarrow got[m] \cup got\_msg$
25          $path \leftarrow path \cdot p$
26          for all $q$ such that $q \in neighbor(p)$ and $q$ appears at most once in $path$ do
27              send $(m, got[m], path)$ to $q$
```

FIG. 7.2. *General network case—quiescent implementation of* broadcast *and* deliver *using* $\mathcal{HB}$.

quiescent implementation of reliable broadcast that uses $\mathcal{HB}$. The first few lemmas are obvious.

LEMMA 7.1 (uniform integrity). *For every message m, every process delivers message m at most once, and only if m was previously broadcast by sender(m).*

LEMMA 7.2 (validity). *If a correct process broadcasts a message m, then it eventually delivers m.*

LEMMA 7.3. *For any processes p and q,* (1) *if at some time t, $q \in got_p[m]$, then $q \in got_p[m]$ at every time $t' \geq t$;* (2) *when $got_p[m]$ is initialized, $p \in got_p[m]$; and* (3) *if $q \in got_p[m]$, then q delivered m.*

LEMMA 7.4. *For every m and path, there is a finite number of distinct messages of the form $(m, *, path)$.*

LEMMA 7.5. *If some process sends a message of the form $(m, *, path)$, then no process appears more than twice in path.*

LEMMA 7.6. *Suppose link $p \to q$ is fair, and p and q are correct processes. If p delivers a message m, then q eventually delivers m.*

Proof. Suppose, by contradiction, that p delivers m and q never delivers m. Since p delivers m and it is correct, it forks task *diffuse*(m). Since q does not deliver m, by Lemma 7.3 part (3) q never belongs to $got_p[m]$. Since p is correct, this implies that p executes the loop in lines 11–17 an infinite number of times. Since q is a correct neighbor of p, the $\mathcal{HB}$-accuracy property guarantees that the heartbeat sequence of q at p is nondecreasing and unbounded. Thus the condition in line 13 evaluates to true an infinite number of times. Therefore, p executes line 14 an infinite number of times, and so p sends a message of the form $(m, *, p)$ to q an infinite number of times. By Lemma 7.4, there exists a subset $g_0 \subseteq \Pi$ such that p sends message (m, g_0, p) infinitely often to q. So, by the fairness property of link $p \rightarrow q$, q eventually receives (m, g_0, p). Therefore, q delivers m. This contradicts the assumption that q does not deliver m. ∎

LEMMA 7.7 (agreement). *If a correct process p delivers a message m, then every correct process q eventually delivers m.*

Proof (sketch). By successive applications of Lemma 7.6 over any fair path from p to q. ∎

We now show that the algorithm in Figure 7.2 is quiescent. In order to do so, we focus on a single invocation of broadcast and show that it causes the sending of only a finite number of messages. This implies that the implementation sends only a finite number of messages when broadcast is invoked a finite number of times.

Let m be a message and consider an invocation of broadcast(m). This invocation can only cause the sending of messages of form $(m, *, *)$. Thus, all we need to show is that every process eventually stops sending messages of this form.

LEMMA 7.8. *Let p be a correct process and q be a correct neighbor of p. If p forks task diffuse(m), then eventually condition $q \in got_p[m]$ holds forever.*

Proof. By Lemma 7.3 part (1), we only need to show that eventually q belongs to $got_p[m]$. Suppose, by contradiction, that q never belongs to $got_p[m]$. Let $(p_1, p_2, \ldots, p_{k'})$ be a simple fair path[13] from p to q with $p_1 = p$ and $p_{k'} = q$. Let $(p_{k'}, p_{k'+1}, \ldots, p_k)$ be a simple fair path from q to p with $p_k = p$. For $1 \leq j < k$, let $P_j = (p_1, p_2, \ldots, p_j)$. Note that a process can appear at most twice in P_k. Thus, for $1 \leq j < k$, process p_{j+1} appears at most once in P_j.

We claim that for each $j \in \{1, \ldots, k-1\}$, there is a set g_j containing $\{p_1, p_2, \ldots, p_j\}$ such that p_j sends (m, g_j, P_j) to p_{j+1} an infinite number of times. For $j = k-1$, this claim together with the fairness property of link $p_{k-1} \rightarrow p_k$ immediately implies that $p_k = p$ eventually receives (m, g_{k-1}, P_{k-1}). Upon the receipt of such a message, p adds the contents of g_{k-1} to its variable $got_p[m]$. Since g_{k-1} contains $p_{k'} = q$, this contradicts the fact that q never belongs to $got_p[m]$.

We show the claim by induction on j. For the base case note that, since q never belongs to $got_p[m]$ and q is a neighbor of $p_1 = p$, then p_1 executes the loop in lines 11–17 an infinite number of times. Since q is a correct neighbor of p_1, the $\mathcal{HB}$-accuracy property guarantees that the heartbeat sequence of q at p_1 is nondecreasing and unbounded. Thus, the condition in line 13 evaluates to true an infinite number of times. So p_1 executes line 14 infinitely often. Since p_2 is a correct neighbor of p_1, its heartbeat sequence is nondecreasing and unbounded, and so p_1 sends messages of the form $(m, *, p_1)$ to p_2 an infinite number of times.[14] By Lemma 7.4, there is some g_1 such that p_1 sends (m, g_1, p_1) to p_2 an infinite number of times. Note that Lemma 7.3 parts (1) and (2) imply that $p_1 \in g_1$. This shows the base case.

For the induction step, suppose that for $j < k-1$, p_j sends (m, g_j, P_j) to p_{j+1} an infinite number of times for some g_j containing $\{p_1, p_2, \ldots, p_j\}$. By the fairness property of link $p_j \rightarrow p_{j+1}$, p_{j+1} receives (m, g_j, P_j) from p_j an infinite number of times. Since p_{j+2} is a neighbor of p_{j+1} and appears at most once in P_{j+1}, each time p_{j+1} receives (m, g_j, P_j), it sends a message of the form $(m, *, P_{j+1})$ to p_{j+2}. It is easy

[13]A path is *simple* if all processes in that path are distinct.

[14]This is where the proof uses the fact that p sends message containing m to all its neighbors whose heartbeat increased—even to those (such as p_2) that may already be in $got_p[m]$ (cf. line 14 of the algorithm).

to see that each such message is (m, g, P_{j+1}) for some g that contains both g_j and p_{j+1}. By Lemma 7.4, there exists $g_{j+1} \subseteq \Pi$ such that g_{j+1} contains $\{p_1, p_2, \ldots, p_{j+1}\}$ and p_{j+1} sends (m, g_{j+1}, P_{j+1}) to p_{j+2} an infinite number of times. □

COROLLARY 7.9. *If a correct process p forks task diffuse(m), then eventually p stops sending messages in task diffuse(m).*

Proof. For every neighbor q of p, there are two cases. If q is correct, then eventually condition $q \in got_p[m]$ holds forever by Lemma 7.8. If q crashes, then the $\mathcal{HB}$-completeness property guarantees that the heartbeat sequence of q at p is bounded, and so eventually condition $prev\_hb_p[q] \geq hb_p[q]$ holds forever. Therefore, there is a time after which the guard in line 13 is always false. Hence, p eventually stops sending messages in task *diffuse*(m). □

LEMMA 7.10 (quiescence). *Eventually every process stops sending messages of the form $(m, *, *)$.*

Proof. Suppose, for a contradiction, that the lemma is not true. Then there exists a process p such that p never stops sending messages of the form $(m, *, *)$. By Lemma 7.5, the third component of a message of the form $(m, *, *)$ ranges over a finite set of values. Therefore, there is some fixed *path* such that p sends an infinite number of messages of the form $(m, *, path)$.

Now let $path_0$ to be the shortest path such that there exists some process p_0 that sends messages of the form $(m, *, path_0)$ an infinite number of times. Note that p_0 must be correct. Corollary 7.9 shows that there is a time after which p_0 stops sending messages in its task *diffuse*(m). Since p_0 only sends a message in task *diffuse*(m) or in line 27, then p_0 sends messages of the form $(m, *, path_0)$ in line 27 an infinite number of times. For each $(m, *, path_0)$ that p_0 sends in line 27, p_0 must have previously received a message of the form $(m, *, path_1)$ such that $path_0 = path_1 \cdot p_0$. So p_0 receives a message of the form $(m, *, path_1)$ an infinite number of times. By the uniform integrity property of the links, some process p_1 sends a message of form $(m, *, path_1)$ to p_0 an infinite number of times. But $path_1$ is shorter than $path_0$—a contradiction to the minimality of $path_0$. □

From Lemmas 7.1, 7.2, 7.7, and 7.10 we have the following theorem.

THEOREM 7.11. *For the general network case, the algorithm in Figure* 7.2 *is a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$.*

From this theorem and Remark 3.1 we have the following corollary.

COROLLARY 7.12. *In the general network case, quasi-reliable send and receive between every pair of processes can be implemented with a quiescent algorithm that uses $\mathcal{HB}$.*

8. Implementations of $\mathcal{HB}$.

We now give implementations of $\mathcal{HB}$ for the two types of communication networks that we considered in the previous sections. These implementations do not use timeouts.

8.1. The simple network case.

We assume all links in the network are bidirectional and fair (Figure 7.1a). In this case, the implementation is obvious. Each process periodically sends a HEARTBEAT message to all its neighbors; upon the receipt of such a message from process q, p increases the heartbeat value of q.

8.2. The general network case.

In this case some links are unidirectional and/or not fair (Figure 7.1b). The implementation is more complex than before because each HEARTBEAT has to be diffused, and this introduces the following problem: when a process p receives a HEARTBEAT message it has to relay it even if this is not the first time p receives such a message. This is because this message could be a

```
For every process $p$:

    Initialization:
        for all $q \in neighbor(p)$ do $\mathcal{D}_p[q] \leftarrow 0$

    cobegin
        || Task 1:
            repeat periodically
                for all $q \in neighbor(p)$ do send $(\mathsf{HEARTBEAT}, p)$ to $q$

        || Task 2:
            upon receive $(\mathsf{HEARTBEAT}, path)$ from $q$ do
                for all $q$ such that $q \in neighbor(p)$ and $q$ appears in $path$ do
                    $\mathcal{D}_p[q] \leftarrow \mathcal{D}_p[q] + 1$
                $path \leftarrow path \cdot p$
                for all $q$ such that $q \in neighbor(p)$ and $q$ does not appear in $path$ do
                    send $(\mathsf{HEARTBEAT}, path)$ to $q$
    coend
```

FIG. 8.1. *General network case—implementation of $\mathcal{HB}$.*

new "heartbeat" from the originating process. But this could also be an "old" heartbeat that cycled around the network and came back, and p must avoid relaying such heartbeats.

The implementation is given in Figure 8.1. Every process p executes two concurrent tasks. In the first task, p periodically sends message $(\mathsf{HEARTBEAT}, p)$ to all its neighbors. The second task handles the receipt of messages of the form $(\mathsf{HEARTBEAT}, path)$. Upon the receipt of such message from process q, p increases the heartbeat values of all its neighbors that appear in *path*. Then p appends itself to *path* and forwards message $(\mathsf{HEARTBEAT}, path)$ to all its neighbors that do not appear in *path*.

We now show that, for the general network case, the algorithm in Figure 8.1 implements $\mathcal{HB}$.

LEMMA 8.1. *At every process p, the heartbeat sequence of every neighbor q is nondecreasing.*

Proof. The proof is obvious. □

LEMMA 8.2. *At each correct process p, the heartbeat sequence of every correct neighbor q is unbounded.*

Proof (sketch). It is possible that link $q \to p$ is not fair or not even in the network. However, there is a simple fair path $P = (p_1, \ldots, p_k)$ from q to p with $p_1 = q$ and $p_k = p$. Process $p_1 = q$ sends its heartbeat to all its neighbors infinitely often. Since the links $p_1 \to p_2, \ldots, p_{k-1} \to p_k$ are fair and each p_j is correct, the heartbeats of q are relayed infinitely often through that path, and $p_k = p$ receives them infinitely often. □

COROLLARY 8.3 ($\mathcal{HB}$-accuracy). *At each process the heartbeat sequence of every neighbor is nondecreasing, and at each correct process the heartbeat sequence of every correct neighbor is unbounded.*

Proof. The proof follows from Lemmas 8.1 and 8.2. □

The proofs of the next two lemmas are obvious.

LEMMA 8.4. *If some process p sends $(\mathsf{HEARTBEAT}, path)$ then (1) p is the last process in path, and (2) no process appears twice in path.*

LEMMA 8.5. *Let p, q be processes, and let path be a nonempty sequence of processes. If p receives message* $(\mathsf{HEARTBEAT}, path \cdot q)$ *an infinite number of times, then q receives message* $(\mathsf{HEARTBEAT}, path)$ *an infinite number of times.*

LEMMA 8.6 ($\mathcal{HB}$-completeness). *At each correct process, the heartbeat sequence of every neighbor that crashes is bounded.*

Proof (sketch). Let p be a correct process, and let q be a neighbor of p that crashes. Suppose that the heartbeat sequence of q at p is not bounded. Then p increments $\mathcal{D}_p[q]$ an infinite number of times. So, for an infinite number of times, p receives messages of the form $(\mathsf{HEARTBEAT}, *)$ with a second component that contains q. By Lemma 8.4 part (2), the second component of a message of the form $(\mathsf{HEARTBEAT}, *)$ ranges over a finite set of values. Thus there exists a *path* containing q such that p receives $(\mathsf{HEARTBEAT}, path)$ an infinite number of times.

Let $path = (p_1, \dots, p_k)$. Then, for some $j \le k$, $p_j = q$. If $j = k$, then, by the uniform integrity property of the links and by Lemma 8.4 part (1), q sends $(\mathsf{HEARTBEAT}, path)$ to p an infinite number of times. This contradicts the fact that q crashes. If $j < k$, then, by repeated applications of Lemma 8.5, we conclude that p_{j+1} receives message $(\mathsf{HEARTBEAT}, (p_1, \dots, p_j))$ an infinite number of times. Therefore, by the uniform integrity property of the links and Lemma 8.4 part (1), p_j sends $(\mathsf{HEARTBEAT}, (p_1, \dots, p_j))$ to p_{j+1} an infinite number of times. Since $p_j = q$, this contradicts the fact that q crashes. ☐

By Corollary 8.3 and the above lemma, we have the following theorem.

THEOREM 8.7. *For the general network case, the algorithm in Figure* 8.1 *implements $\mathcal{HB}$.*

9. Stronger communication primitives.

Quasi-reliable send and receive and reliable broadcast are sufficient to solve many problems (see section 10.1). However, stronger types of communication primitives, namely, *reliable send and receive* and *uniform reliable broadcast*, are sometimes needed. We now give quiescent implementations of these primitives for systems with process crashes and message losses.

Let t be the number of processes that may crash. [5] shows that if $t \ge n/2$ (i.e., half of the processes may crash), these primitives cannot be implemented, even if we assume that links may lose only a finite number of messages and we do not require that the implementation be quiescent.

We now show that if $t < n/2$, then there are quiescent implementations of these primitives for the two types of network considered in this paper. The implementations that we give here are simple and modular but are inefficient (in terms of number of messages sent). More efficient ones can be obtained by modifying the algorithms in sections 7.1 and 7.2. Hereafter, we assume that $t < n/2$.

9.1. Reliable send and receive.

Consider any two distinct processes s and r. We define *reliable send and receive from s to r* in terms of two primitives: $\mathsf{r\text{-}send}_{s,r}$ and $\mathsf{r\text{-}receive}_{r,s}$. We require that if a correct process invokes r-send, it eventually returns from this invocation. If a process s returns from the invocation of $\mathsf{r\text{-}send}_{s,r}(m)$, we say that *$s$ completes the* r-send *of message m to r.* With quasi-reliable send and receive, it is possible that s completes the qr-send of m to r, then s crashes, and r never qr-receives m (even though it does not crash). In contrast, with reliable send and receive primitives, if s completes the r-send of message m to a correct process r, then r eventually r-receives m (even if s crashes). More precisely, reliable send and receive satisfies the following properties.

1. *Uniform integrity.* For all $k \ge 1$, if r r-receives m from s exactly k times by time t, then s r-sent m to r at least k times before time t.

```
1   For process $s$:
2
3       Initialization:
4           $seq \leftarrow 0$                          { $seq$ is the current sequence number }
5
6       To execute r-send$_{s,r}(m)$:
7           $seq \leftarrow seq + 1$
8           $lseq \leftarrow seq$
9           broadcast$(m, lseq, s, r)$
10          wait until qr-received (ACK, $lseq$) from $t+1$ processes
11          return
12
13  For every process $p$:
14
15      upon deliver$(m, lseq, s, r)$ do
16          qr-send$_{p,s}$(ACK, $lseq$)
17          if $p = r$ then r-receive$_{r,s}(m)$
```

FIG. 9.1. *Quiescent implementation of* r-send$_{s,r}$ *and* r-receive$_{r,s}$ *for* $n > 2t$.

2. *No loss.* For all $k \geq 1$, if r is correct and s completes the r-send of m to r exactly k times by time t, then r eventually r-receives m from s at least k times.[15]

A quiescent implementation of r-send and r-receive can be obtained using a quiescent implementation of reliable broadcast and of qr-send/qr-receive between every pair of processes. Roughly speaking, when s wishes to r-send m to r, it broadcasts a message that contains m, s, r, and a fresh sequence number, and then waits to qr-receive $t+1$ acknowledgements for that message before returning from this invocation of r-send. When a process p delivers this broadcast message, it qr-sends an acknowledgement back to s, and if $p = r$, then it also r-receives m from s. This algorithm is shown in Figure 9.1 (the code consisting of lines 7 and 8 is executed atomically).

9.2. Uniform reliable broadcast. The agreement property of reliable broadcast states that if a *correct* process delivers a message m, then all correct processes eventually deliver m. This requirement allows a *faulty* process (i.e., one that subsequently crashes) to deliver a message that is never delivered by the correct processes. This behavior is undesirable in some applications, such as *atomic commitment* in distributed databases [4, 19, 22]. For such applications, a stronger version of reliable broadcast is more suitable, namely, *uniform reliable broadcast* which satisfies uniform integrity, validity (section 3.2), and the following.

- *Uniform agreement* [31]. If *any* process delivers a message m, then all correct processes eventually deliver m.

A quiescent implementation of uniform reliable broadcast can be obtained using quiescent implementations of reliable broadcast, and of quasi-reliable send and receive between every pair of processes. Roughly speaking, when p wishes to uniform-broadcast m, it broadcasts m. Upon the delivery of m, each process r qr-sends an acknowledgement to every process, waits for the qr-receipt of such acknowledgements from $t+1$ processes, and then uniform-delivers m.

[15]The no loss and quasi no loss properties are very similar to the strong validity and validity properties in section 6 of [23].

10. Using $\mathcal{HB}$ to extend previous work.

$\mathcal{HB}$ can be used to extend previous work in order to solve problems with algorithms that are both quiescent and tolerant of process crashes and messages losses.

10.1. Extending existing algorithms to tolerate link failures.

$\mathcal{HB}$ can be used to transform many existing algorithms that tolerate process crashes into quiescent algorithms that tolerate both process crashes and message losses. For example, consider the randomized consensus algorithms of [7, 14, 16, 33], the failure-detector based ones of [2, 12], the probabilistic one of [9], and the algorithms for atomic broadcast in [12], k-set agreement in [13], atomic commitment in [20], and approximate agreement in [15]. All these algorithms tolerate process crashes. Moreover, it is easy to verify that the only communication primitives that they actually need are quasi-reliable send and receive and/or reliable broadcast. Thus in systems where $\mathcal{HB}$ is available, all these algorithms can be made to tolerate both process crashes and message losses (with fair links) by simply plugging in the quiescent communication primitives given in section 7.[16] The resulting algorithms tolerate message losses and are quiescent.

10.2. Extending results of Basu, Charron-Bost, and Toueg [5].

Another way to solve problems with quiescent algorithms that tolerate both process crashes and message losses is obtained by extending the results of [5]. That work addresses the following question: given a problem that can be solved in a system where the only possible failures are process crashes, is the problem still solvable if links can also fail by losing messages? One of the models of lossy links considered in [5] is called *fair lossy*. Roughly speaking, a fair lossy link $p \to q$ satisfies the following property. If p sends an infinite number of messages to q and q is correct, then q receives an infinite number of messages from p (see section 11.3 for a brief comparison between fair lossy and fair links).

[5] establishes the following result: any problem P that can be solved in systems with process crashes can also be solved in systems with process crashes and fair lossy links, provided P is *correct-restricted*[17] or a majority of processes are correct. For each of these two cases, [5] shows how to transform any algorithm that solves P in a system with process crashes into one that solves P in a system with process crashes and fair lossy links. The algorithms that result from these transformations, however, are not quiescent: each transformation requires processes to repeatedly send messages forever.

Given $\mathcal{HB}$, we can modify the transformations in [5] to ensure that if the original algorithm is quiescent, then so is the transformed one. Roughly speaking, the modification consists of (1) adding message acknowledgements; (2) suppressing the sending of a message from p to q if either (a) p has received an acknowledgement for that message from q, or (b) the heartbeat of q has not increased since the last time p sent a message to q; and (3) modifying the meaning of the operation "append $Queue_1$ to $Queue_2$" so that only the elements in $Queue_1$ that are not in $Queue_2$ are actually appended to $Queue_2$. The results in [5], combined with the above modification, show that if a problem P can be solved with a quiescent algorithm in a system with crash failures only, and either P is correct-restricted or a majority of processes are correct,

[16] This can also be done to algorithms that require reliable send/receive or uniform reliable broadcast by plugging in the implementations given in section 9, provided a majority of processes are correct.

[17] Intuitively, a problem P is correct-restricted if its specification does not refer to the behavior of faulty processes [6, 18].

then P is solvable with a quiescent algorithm that uses $\mathcal{HB}$ in a system with crash failures and fair lossy links.

11. Concluding remarks.

11.1. About message buffering. We now address the issue of message buffering in the implementation of quasi-reliable send and receive, and of reliable broadcast (section 7). Soon after a process p crashes, its heartbeat ceases everywhere and processes stop sending messages to p. However, they do have to keep the messages they intended to send to p, just in case p is merely very slow, and the heartbeat of p resumes later on. In theory, they have to keep these messages forever. In practice, however, the system will eventually decide that p is indeed useless and will "remove" p (e.g., via a group membership protocol). All the stored messages addressed to p can then be discarded. The removal of p may take a long time,[18] but the heartbeat mechanism ensures that processes stop sending messages to p soon after p actually crashes, and much before its removal.

11.2. Quiescence versus termination. In this paper, we considered reliable communication protocols that tolerate process crashes and message losses, and we focused on achieving quiescence. What about achieving termination? A *terminating* protocol guarantees that every process eventually reaches a halting state from which it cannot take further actions. A terminating protocol is obviously quiescent, but the converse is not necessarily true. For example, consider the protocol described at the beginning of section 1. In this protocol, (a) s sends a copy of m repeatedly until it receives $ack(m)$ from r, and then it halts; and (b) upon each receipt of m, r sends $ack(m)$ back to s. In the absence of process crashes this protocol is quiescent. However, the protocol is not terminating because r never halts: r remains (forever) ready to reply to the receipt of a possible message from s.

Can we use $\mathcal{HB}$ to obtain reliable communication protocols that are *terminating*? The answer is no, *even for systems with no process crashes*, as we now explain. Consider a system with message losses (fair links) and no process crashes. [27] proves that for any terminating protocol P and any initial configuration of P, there are runs of P in which all processes halt without receiving any message. This implies that a terminating protocol cannot solve the reliable communication problem (in systems with fair links).

To deal with this problem, we propose a layering that allows *applications* to terminate. This layering, shown in Figure 11.1, separates applications, reliable communication, and failure detection. At the lowest level, there are failure detectors, such as $\mathcal{HB}$. Of course, these are neither quiescent nor terminating. At the middle level, there are reliable communication protocols, such as those that we described in sections 3 and 9. These communication protocols are quiescent (thanks to the failure detectors at the lower level) but not terminating. Finally, at the top level, there are applications, such as concurrent instances of consensus, atomic broadcast, atomic commitment protocols, etc. Applications are both quiescent *and* terminating: they achieve termination thanks to the reliable communication layer. For example, consider an instance of consensus. Once a process decides, it delegates the task of broadcasting the decision value to the reliable communication layer, and then it terminates (without waiting for the broadcast to terminate). Since every correct process eventually decides and terminates, this instance of consensus terminates.

[18]In some group membership protocols, the timeout used to remove a process is on the order of minutes: killing a process is expensive and so timeouts are set conservatively.

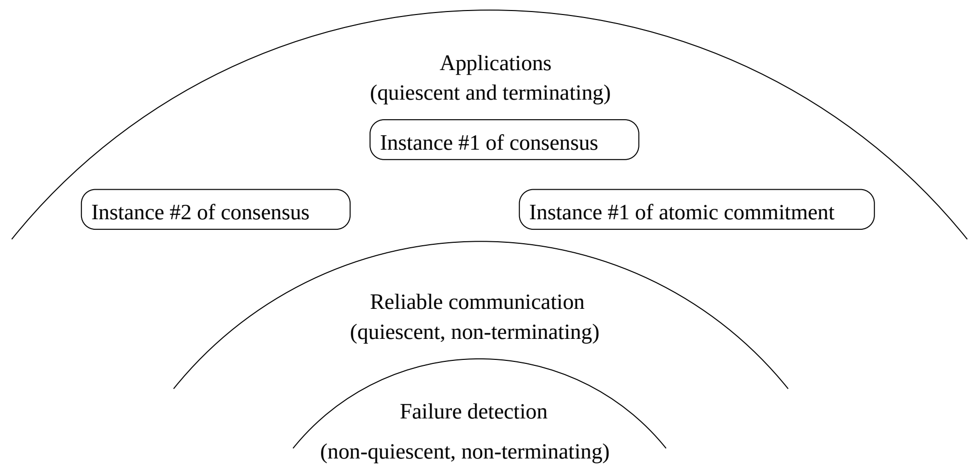

Applications
(quiescent and terminating)
Instance #1 of consensus
Instance #2 of consensus
Instance #1 of atomic commitment
Reliable communication
(quiescent, non-terminating)
Failure detection
(non-quiescent, non-terminating)

FIG. 11.1. *Layering that separates applications, reliable communication, and failure detection.*

If necessary, termination in the reliable communication layer can also be achieved in practice, as we now explain. A reliable communication protocol is unable to terminate when processes cannot determine whether a nonresponsive process has crashed or it is only very slow. However, as we mentioned in our discussion of message buffering, a process that actually crashes is eventually removed by the operating system or a group membership protocol (and the remaining processes are notified accordingly). When this happens, the communication protocol can terminate. Note that with the heartbeat mechanism quiescence can be achieved long before termination (this is because when a process crashes, it may take a relatively long time to decide that it actually crashed, but its heartbeat count at other processes stops increasing almost immediately).

As a final remark, we note that some communication protocols, such as standard *data link* protocols, are inherently nonterminating: they are shared communication services that are always "ready" for message transmission. The reliable communication protocols (in our middle level) could also be viewed in the same way, namely, as nonterminating shared services that are always ready for message transmission.

11.3. Fair links versus fair lossy links. Fair links and fair lossy links are two typical models of lossy links considered in the literature.[19] Roughly speaking, a fair link guarantees that for every m, if p sends m to q an infinite number of times, and q is correct, then q receives m an infinite number of times. On the other hand, a fair lossy link guarantees that if p sends an infinite number of messages to q, and q is correct, then q receives an infinite number of messages from p. Fair lossy links and fair links differ in a subtle way. For instance, if process p sends the infinite sequence of distinct messages $m_1, m_2, m_3, \dots$ to q and $p \to q$ is fair lossy, then q is guaranteed to receive an infinite subsequence, whereas if $p \to q$ is fair, q may receive nothing (because each distinct message is sent only once). On the other hand, if p sends the infinite sequence $m_1, m_2, m_1, m_2, \dots$ and $p \to q$ is fair lossy, q may never receive a copy of m_2 (while

[19]In [29], these links correspond to the strong and weak loss limitation properties, respectively.

it receives m_1 infinitely often), whereas if $p \to q$ is fair, q is guaranteed to receive an infinite number of copies of both m_1 and m_2.

In this paper, we chose the fair links model. A natural question is whether our results still hold with fair lossy links instead. It turns out that the answer is yes, as we now explain. First note that Theorems 4.1, 5.5, and A.12 still hold because their proofs rely only on the fact that lossy links may lose (or not lose) messages arbitrarily during any finite period of time—a behavior allowed by fair lossy links. Moreover, the algorithms in sections 7 and 8 can be easily modified to work with fair lossy links through the use of piggybacking; namely, every time a process wishes to send a message, it piggybacks all the messages that it previously sent.[20] Finally, the algorithms in section 9 are still correct because they do not directly use the communication links; rather, they rely only on the communication algorithms of section 7.

11.4. Quiescent versus nonquiescent transformations. We proved that if $\mathcal{D}$ is a failure detector with finite range that can be used to solve quiescent reliable communication, then $\mathcal{D}$ can be transformed to $\diamond\mathcal{P}$. Our transformation is not quiescent: to "extract" $\diamond\mathcal{P}$ out of $\mathcal{D}$, processes keep on sending messages forever. This, however, does not invalidate the two facts that we wanted to show, namely:

1. $\mathcal{D}$ encodes at least as much information as $\diamond\mathcal{P}$.
2. $\mathcal{D}$ *cannot* be implemented (this follows from the transformation from $\mathcal{D}$ to $\diamond\mathcal{P}$, and the fact that $\diamond\mathcal{P}$ cannot be implemented).

This shows that finite-range failure detectors have some inherent limitation (because there is a failure detector with infinite range, namely $\mathcal{HB}$, that can be used to solve quiescent reliable communication such that (1) $\mathcal{HB}$ does not encode $\diamond\mathcal{P}$, and (2) $\mathcal{HB}$ can be implemented).

Even though a nonquiescent transformation was sufficient to establish our results, quiescent transformations are necessary when comparing the power of failure detectors to solve tasks with quiescent algorithms, as we now explain. If $\mathcal{D}$ can be transformed to $\mathcal{D}'$, we can conclude that $\mathcal{D}$ is (at least) as powerful as $\mathcal{D}'$ in terms of task solving (intuitively, a *task* is a relation between inputs and outputs [8, 25]). If the transformation from $\mathcal{D}$ to $\mathcal{D}'$ is not quiescent, however, $\mathcal{D}$ may not be as powerful as $\mathcal{D}'$ in terms of solving tasks *quiescently*: there may be a task that can be solved quiescently with $\mathcal{D}'$ but not with $\mathcal{D}$. On the other hand, if the transformation from $\mathcal{D}$ to $\mathcal{D}'$ *is* quiescent, we can conclude that $\mathcal{D}$ is (at least) as powerful as $\mathcal{D}'$ in terms of solving tasks with quiescent algorithms. The study of quiescent transformations is a new and interesting subject of research.

11.5. Extension to partitionable networks. In this paper, we considered networks that do not partition: we assumed that every pair of correct processes are reachable from each other through fair paths. In a subsequent paper [1], we drop this assumption and consider *partitionable networks.* We first generalize the definition of $\mathcal{HB}$ and show how to implement it in such networks. We then consider generalized versions of reliable communication and of consensus for partitionable networks and use $\mathcal{HB}$ to solve these problems with quiescent protocols (to solve consensus we also use a generalization of the *eventually strong* failure detector [12]).

Appendix. Removing the simplifying assumption from Theorem 5.5.

We now give an extended, more complex proof of Theorem 5.5 without the simplifying assumption.

[20] With the fair links used in this paper, this expensive piggybacking is avoided. We believe that in practice, links that intermittently lose messages are *both* fair and fair lossy.

Let $\mathcal{E}$ be an environment and $\mathcal{D}$ be any failure detector with finite range $\mathcal{R} = \{v_1, v_2, \ldots, v_\ell\}$. Let $\mathcal{I}$ be a quiescent implementation of s-send and s-receive that uses $\mathcal{D}$ in environment $\mathcal{E}$.

As in the simpler proof in section 5, the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ uses a finite table that is predetermined from $\mathcal{D}$. We first define this table and show some of its properties (section A.1). We then describe and prove the correctness of the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ that uses this table (section A.2).

A.1. The predetermined table. For the definitions in this proof, let

- v_j be a failure detector value, i.e., $v_j \in \mathcal{R}$,
- p be a process, i.e., $p \in \Pi$,
- F be a failure pattern,
- H be a failure detector history with range $\mathcal{R}$,
- f be an assignment of failure detector values to every process in Π, i.e., $f : \Pi \longrightarrow \mathcal{R}$,
- P and P_0 be a nonempty set of processes,
- $p_0, p_1, \ldots, p_{m-1}$ be the processes in P (where $m = |P|$ and $p_0 < p_1 < \cdots < p_{m-1}$).

DEFINITION A.1. *We say that* v_j *is* a limit value for p and H *if, for infinitely many* t, $H(p,t) = v_j$.

DEFINITION A.2. *We say that* f is a limit vector for P and H *if for all* $p \in P$, $f(p)$ *is a limit value for* p *and* H*. The set of all limit vectors for* P *and* H *is denoted* $L_P(H)$.

DEFINITION A.3. *RRIRounds*(P, f) *is defined as follows.*

Consider the round-robin execution of implementation $\mathcal{I}$ *in which* (a) *processes in* P *take steps forever in a round-robin fashion*[21] *and processes in* $\Pi \setminus P$ *do not take any steps,* (b) *no process ever* s-send*s any bit,* (c) *every time a process* $p \in P$ *queries its failure detector module,* p *gets* $f(p)$, (d) *every time a process* $p \in P$ *takes a step,* p *receives the earliest message sent to it that it did not yet receive (thus, every* $p \in P$ *eventually receives each message sent to it), and* (e) *all messages sent to processes in* $\Pi \setminus P$ *are lost.*[22]

There are two possible cases in the above round-robin execution of $\mathcal{I}$.

1. Every process eventually stops sending messages. *In this case, after some number* k *of round-robin rounds, no process ever receives any messages. We say that "round-robin initialization (r.r.i.) occurs in* k *rounds," and define* *RRIRounds*$(P, f) = k$.
2. Some process never stops sending messages. *In this case, we define* *RRIRounds*$(P, f) = \infty$.

Intuitively, we say that F and H allow r.r.i. for P and f if the following hold: (a) in the above execution with P and f, r.r.i. occurs in k rounds for some k, and (b) there is a schedule compatible with F and H that allows this k-round r.r.i. More precisely, we have the following definition.

DEFINITION A.4. *We say that* F and H allow r.r.i. for P and f *if*

(a) *RRIRounds*$(P, f) = k$ *for some* k, *and*

(b) *there are times* $t_0 < t_1 < \cdots < t_{mk-1}$ *such that for every* $0 \le j \le mk - 1$, (1) $p_{j \bmod m}$ *is not crashed at time* t_j, *i.e.,* $p_{j \bmod m} \notin F(t_j)$, *and* (2)

[21] That is, p_0 takes the first step, then p_1 takes a step, and so on, so that the jth step is taken by process $p_{(j-1) \bmod m}$.

[22] It is possible that this is *not* a valid execution of $\mathcal{I}$ using $\mathcal{D}$ in environment $\mathcal{E}$.

the failure detector module of $p_{j \bmod m}$ *at time* t_j *outputs* $f(p_{j \bmod m})$, *i.e.,* $H(p_{j \bmod m}, t_j) = f(p_{j \bmod m})$.

DEFINITION A.5. $L_{P,P_0}(F,H) = \{f \mid f \in L_P(H)$, *and* F *and* H *allow r.r.i. for* P_0 *and* $f\}$.

DEFINITION A.6. $E^{\mathcal{D},\mathcal{E}}_{P,P_0} = \{f \mid \exists F \in \mathcal{E}, \exists H \in \mathcal{D}(F) : P = \mathit{correct\_proc}(F)$, *and* $f \in L_{P,P_0}(F,H)\}$.

Roughly speaking, $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ is the set of limit vectors f that could occur when P is the set of correct processes and it is possible to have r.r.i. for P_0 and f.

The table used by the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ consists of all the sets $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ where P and P_0 range over all nonempty subsets of processes. Note that this table is finite. We omit the superscript $\mathcal{D},\mathcal{E}$ from $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ whenever it is clear from the context.

LEMMA A.7. *Let* $F \in \mathcal{E}$, $P = \mathit{correct\_proc}(F)$, $H \in \mathcal{D}(F)$, *and* $f \in L_P(H)$. *Assume* $P \neq \emptyset$. *Then* $\mathit{RRIRounds}(P,f) < \infty$.

Proof. We can construct a run R of implementation $\mathcal{I}$ using $\mathcal{D}$ with $F \in \mathcal{E}$, such that all processes behave exactly as in the round-robin execution of $\mathcal{I}$ that was used to define $\mathit{RRIRounds}(P,f)$. To see this, note that since $F \in \mathcal{E}$, $P = \mathit{correct\_proc}(F)$, $H \in \mathcal{D}(F)$, and $f \in L_P(H)$, we can find times for the round-robin steps of correct processes such that, for each time u at which a process p takes a step, the output $H(p,u)$ of its failure detector module is $f(p)$. Since $\mathcal{I}$ is quiescent, there is a time after which no process sends any message in run R. Thus, $\mathit{RRIRounds}(P,f) < \infty$. ∎

LEMMA A.8. *Let* $F \in \mathcal{E}$, $P = \mathit{correct\_proc}(F)$, *and* $H \in \mathcal{D}(F)$. *Assume* $P \neq \emptyset$ *and let* P_0 *be such that* $P \subseteq P_0 \subseteq \Pi$. *If* $f \in L_{P,P_0}(F,H)$, *then* $f \in E_{P,P_0}$ *and* $f \notin E_{P',P_0}$ *for all* P' *such that* $\emptyset \subset P' \subset P$.

Proof. Let $f \in L_{P,P_0}(F,H)$. The fact that $f \in E_{P,P_0}$ is immediate from the definition of E_{P,P_0}. Let P' be such that $\emptyset \subset P' \subset P$. Suppose, for contradiction, that $f \in E_{P',P_0}$. Then there exists a failure pattern $F' \in \mathcal{E}$ and $H' \in \mathcal{D}(F')$ such that $P' = \mathit{correct\_proc}(F')$ and $f \in L_{P',P_0}(F',H')$.

We now obtain a contradiction by using the quiescent implementation $\mathcal{I}$. Let p be a process in P' and q be a process in $P \setminus P'$. We construct three runs of $\mathcal{I}$, namely, R_0, R_1, and R_2. Roughly speaking, each one of these runs starts with an r.r.i. for P_0 and f. After this initialization, in R_0 nothing else happens, in R_1 process p s-sends some bit to q but q crashes, and in R_2 process p s-sends the same bit to q and q is correct. We will reach a contradiction by arguing that in R_2 process q behaves as in R_0, and thus it never s-receives any bit from p—this violates the defining property of s-send and s-receive.

Runs R_0, R_1, and R_2 are defined as follows.[23]

1. Run R_0 has failure pattern F and failure detector history H. Since $f \in L_{P,P_0}(F,H)$, $f \in L_P(H)$, and F and H allow r.r.i. for P_0 and f. R_0 consists initially of an r.r.i. for P_0 and f. More precisely, initially: (a) processes in P_0 take steps in a round-robin fashion and processes in $\Pi \setminus P_0$ do not take any steps, (b) no process s-sends any bit, (c) every time a process $r \in P_0$ queries its failure detector module, r gets $f(r)$, (d) every time a process $r \in P_0$ takes a step, r receives the earliest message sent to it that it did not yet receive, and (e) all messages sent to processes in $\Pi \setminus P_0$ are lost. This goes on until each process in P_0 has taken $\mathit{RRIRounds}(P_0,f)$ steps. Let

[23]In each one of these runs, we will require that for a certain *finite* period of time, some messages are lost while others are not. As we explained in our model (section 2.5), this behavior is consistent with any link failure pattern.

t_0 be the time when this happens. After t_0, processes in P take steps in a round-robin fashion such that every time a process $r \in P$ takes a step, it obtains $f(r)$ from its failure detector module (this is possible because $f \in L_P(H)$); moreover, no process s-sends any bit. Note that since both p and q are in $P = \mathit{correct\_proc}(F)$, and p does not s-send any bit to q, it must be that q does not s-receive any bit from p. Furthermore, after time t_0, no processes send or receive any messages.

2. Run R_1 has failure pattern F' and failure detector history H'. Since $f \in L_{P',P_0}(F', H')$, $f \in L_{P'}(H')$, and F' and H' allow r.r.i. for P_0 and f. Initially, processes in R_1 behave as in R_0, i.e., R_1 starts with an r.r.i. for P_0 and f. Then, execution proceeds as follows: (a) p s-sends some bit b to q, (b) processes in P' take steps in round-robin fashion and processes in $\Pi \setminus P'$ take no steps, (c) every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module, (d) every time a process $r \in P'$ takes a step, r receives the earliest message sent to it that it did not yet receive, and (e) all messages sent to processes in $\Pi \setminus P'$ are lost.

Note that, since implementation $\mathcal{I}$ is quiescent, there is a time t_1 after which no messages are sent or received. Assume without loss of generality that at time t_1 every process in P' took the same number k of steps (otherwise, choose another time $t_1' > t_1$).

3. Run R_2 has failure pattern F and failure detector history H. Initially, processes in R_2 behave as in R_1: R_2 starts with an r.r.i. for P_0 and f, and then p s-sends b to q and execution continues as in R_1, until each process in P' has taken k steps (this is possible because $f \in L_P(H)$ and $P' \subseteq P$). Let t_2 be the time when this happens. After t_2, execution proceeds as follows: (a) no process s-sends any bit, (b) processes in P take steps in round-robin fashion and processes in $\Pi \setminus P$ take no steps, and (c) every time a process $r \in P$ takes a step, it obtains $f(r)$ from its failure detector module (this is possible because $f \in L_P(H)$).

In R_2, at time t_2, each process in P' is in the same state as in run R_1 at time t_1, and each process in $P \setminus P'$ is in the same state as in run R_0 at time t_0. A simple induction on the steps taken shows that, in R_2, (1) processes in P' have the same behavior as in run R_1; (2) processes in $P \setminus P'$ have the same behavior as in run R_0; (3) no messages are sent or received after time t_2. Since $q \in P \setminus P'$ and q does not s-receive any bit from p in R_0, it does not s-receive any bit from p in R_2.

In summary, in R_2: (a) both p and q are correct; (b) p s-sends b to q; and (c) q does not s-receive b from p. Thus, $\mathcal{I}$ is not a correct implementation of s-send and s-receive—a contradiction. ∎

A.2. The transformation algorithm. The algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}' = \diamond\mathcal{P}$ in environment $\mathcal{E}$ is shown in Figure A.1. $T_{\mathcal{D}\to\mathcal{D}'}$ uses the table of sets E_{P,P_0} (for all nonempty subsets P and P_0 of processes) that has been determined a priori from the given $\mathcal{D}$ and $\mathcal{E}$. It also uses an implementation of qr-send and qr-receive between every pair of processes. A simple implementation is by repeated retransmissions and diffusion (it does not have to be quiescent).

All variables are local to each process. *Sequences* is a finite set of finite sequences of pairs (p, v) where $p \in \Pi$ is a process and $v \in \mathcal{R}$ is a failure detector value. It stores possible schedules that could have resulted from F and H. Vector f stores the last failure detector value that p qr-received from each process. *Order* is an ordered set that records the order in which the last failure detector value from each process was qr-received. $\mathcal{D}'_p$ denotes the output of the eventually perfect failure detector that p is simulating (a set of processes that p currently suspects). *AllowsRRI* is a Boolean

```
For every process $p$:

    Initialization:
        for all $q \in \Pi$ do $f[q] \leftarrow \bot$
        $Order \leftarrow \emptyset$
        $Sequences \leftarrow \{\lambda\}$
        $\mathcal{D}'_p \leftarrow \emptyset$
        { For each $\emptyset \subset P, P_0 \subseteq \Pi$, the set $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ is determined a priori from $\mathcal{D}$ and $\mathcal{E}$ }

    cobegin
        || Task 1:
            repeat periodically
                $v \leftarrow \mathcal{D}_p$                                        {query $\mathcal{D}$}
                append $(p, v)$ to each sequence in $Sequences$
                for all $q \in \Pi$ do qr-send $(Sequences, v)$ to $q$

        || Task 2:
            upon qr-receive $(Sequences', v')$ from $q$ do
                $f[q] \leftarrow v'$
                $Order \leftarrow q \mid\mid (Order \setminus \{q\})$      {process $q$ is moved to the front of $Order$}
                $Sequences \leftarrow Sequences \cup Sequences'$
                if for some $k \geq 1$, $AllowsRRI(Sequences, Order[1..k], f)$ then
                    let $k_0$ be the largest such $k$
                    if for some $k' \geq 1$, $f \in E^{\mathcal{D},\mathcal{E}}_{Order[1..k'],Order[1..k_0]}$ then
                        let $k_1$ be the smallest such $k'$
                        $\mathcal{D}'_p \leftarrow \Pi \setminus Order[1..k_1]$    {suspect processes not in $Order[1..k_1]$}
    coend
```

FIG. A.1. *Transformation of $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}'$.*

function that takes three parameters: a set *Sequences*, a set $P = \{p_0, p_1, \ldots, p_{m-1}\} \subseteq \Pi$ (where $p_0 < p_1 < \cdots < p_{m-1}$), and a vector f. It returns true if and only if for some sequence $s \in Sequences$, there exists a subsequence of s that consists of $RRIRounds(P, f)$ repetitions of $(p_0, f(p_0)), (p_1, f(p_1)), \ldots, (p_{m-1}, f(p_{m-1}))$.

In Task 1, each process p periodically queries its failure detector module, appends a new pair to each sequence in *Sequences*, and then qr-sends *Sequences* and the output of its failure detector module $\mathcal{D}_p$ to every process. Upon the qr-receipt of $(Sequences', v')$ from process q in Task 2, process p enters v' into $f[q]$, moves q to the front of *Order*, and updates *Sequences*. Then, p uses the function *AllowsRRI* to check whether there is some k such that r.r.i. could have occurred for $Order[1..k]$ and f. If there is, it sets k_0 to the *largest* such k and then checks if for some k', $f \in E_{Order[1..k'],Order[1..k_0]}$. If so, it sets k_1 to the *smallest* such k' and sets $\mathcal{D}'$ to the complement of $Order[1..k_1]$.

We now show that the failure detector constructed by this algorithm, namely $\mathcal{D}'$, is an eventually perfect failure detector. Consider a run of this algorithm with failure pattern $F \in \mathcal{E}$ and failure detector history $H \in \mathcal{D}(F)$, such that $correct\_proc(F) \neq \emptyset$. Let t be the number of processes that crash in F, i.e., $t = |\Pi \setminus correct\_proc(F)|$. Henceforth, p denotes a correct process in F, and f, *Order*, and *Sequences* are variables local to p.

LEMMA A.9. *There is a time t_0 after which* (1) $Order[1..n-t] = correct\_proc(F)$, (2) $f \in L_{Order[1..n-t]}(H)$, *and* (3) $AllowsRRI(Sequences, Order[1..n-t], f)$.[24]

[24]This does not mean that eventually the values of variables f, *Sequences*, and *Order* at p stop

Proof. Note that p eventually stops qr-receiving messages from processes that crash, and p never stops qr-receiving messages from correct processes. From the way *Order* is updated, there is a time t_1 after which (1) holds.

Let $P = \mathit{correct\_proc}(F)$. Variable f ranges over a finite number of values, so there are functions $f_1, f_2, \ldots, f_N : \Pi \to \mathcal{R}$ such that (a) for every $1 \le j \le N$, variable f is equal to f_j an infinite number of times, and (b) there is a time t_2 after which the predicate $f \in \{f_1, f_2, \ldots, f_N\}$ holds. We now show that for every $1 \le j \le N$, $f_j \in L_P(H)$, and there is a time τ_j after which $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. Together with (1) and (b), this implies that after time $t_0 = \max\{t_1, t_2, \tau_1, \tau_2, \ldots, \tau_N\}$, both (2) and (3) hold.

Let $1 \le j \le N$. We first claim that each process $q \in P$ obtains $f_j(q)$ from $\mathcal{D}$ in line 13 an infinite number of times—this immediately implies $f_j \in L_P(H)$. To show the claim, note that process p qr-receives a message from q and updates $f[q]$ an infinite number of times. Together with (a), this implies that p qr-receives a message containing $f_j(q)$ from q an infinite number of times, and this implies the claim.

We now show that there is a time τ_j after which $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. Since $f_j \in L_P(H)$, by Lemma A.7, $\mathit{RRIRounds}(P, f_j) = k$ for some $k < \infty$. Let $p_0 < p_1 < \cdots < p_{m-1}$ be the processes in P. By the claim, at some time u_0, p_0 obtains $f_j(p_0)$ from $\mathcal{D}$ in line 13. After doing so, p_0 appends $(p_0, f_j(p_0))$ to all sequences in *Sequences* and qr-sends a message containing *Sequences* to all processes. At some time $u_1' > u_0$, p_1 qr-receives this message and updates *Sequences*. By the claim, at some time $u_1 > u_1'$, p_1 obtains $f_j(p_1)$ from $\mathcal{D}$ in line 13. After doing so, p_1 appends $(p_1, f_j(p_1))$ to all sequences in *Sequences* and so p_1 obtains a sequence containing $(p_0, f_j(p_0))$ before $(p_1, f_j(p_1))$. We can repeat this argument for all the processes in P in a round-robin order, for $k+1$ rounds, and conclude that eventually $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. □

LEMMA A.10. *There is a time t_1 after which for every $m_0 \ge n - t$ such that $\mathit{AllowsRRI}(\mathit{Sequences}, \mathit{Order}[1..m_0], f)$ holds:* (1) $f \in E_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}$ *and* (2) *for all* $1 \le m_1 < n - t$, $f \notin E_{\mathit{Order}[1..m_1], \mathit{Order}[1..m_0]}$.

Proof. By Lemma A.9, there is a time t_0 after which (a) $\mathit{Order}[1..n-t] = \mathit{correct\_proc}(F)$, and (b) $f \in L_{\mathit{Order}[1..n-t]}(H)$. Let $t_1 = t_0$. Suppose that at some time $t_1' > t_1$, $\mathit{AllowsRRI}(\mathit{Sequences}, \mathit{Order}[1..m_0], f)$ holds for some $m_0 \ge n - t$. This implies that F and H allow r.r.i. for $\mathit{Order}[1..m_0]$ and f. From (b), $f \in L_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}(F, H)$ holds at time t_1'. By Lemma A.8, $f \in E_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}$.

Let $1 \le m_1 < n - t$. By (a), $\emptyset \subset \mathit{Order}[1..m_1] \subset \mathit{correct\_proc}(F) \subseteq \mathit{Order}[1..m_0]$ holds at time t_1'. Note that $f \in L_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}(F, H)$ holds at time t_1'. By Lemma A.8, $f \notin E_{\mathit{Order}[1..m_1], \mathit{Order}[1..m_0]}$. □

COROLLARY A.11. *There is a time after which $\mathcal{D}_p' = \Pi \setminus \mathit{correct\_proc}(F)$.*

Proof. By Lemma A.9 part (3), there is a time t_0 after which every time p qr-receives some message, the **if** in line 22 evaluates to true and the k_0 selected in line 23 is at least $n - t$. After time t_0, by Lemma A.10, there is a time after which every time p qr-receives some message, the **if** in line 24 evaluates to true and the k_1 selected in line 25 is $n - t$. Now apply Lemma A.9 part (1). □

By Corollary A.11, we have the following theorem.

THEOREM A.12. *Consider an asynchronous system subject to process crashes and message losses. Suppose that failure detector $\mathcal{D}$ with finite range can be used to*

changing. It means that, although they may continue to change forever, eventually the predicates (1), (2), and (3) are true forever at p.

solve the single-shot reliable send and receive problem in environment $\mathcal{E}$ and that the implementation is quiescent. Then $\mathcal{D}$ can be transformed (in environment $\mathcal{E}$) to the eventually perfect failure detector $\diamond\mathcal{P}$.

Acknowledgments. We are grateful to Anindya Basu and Bernadette Charron-Bost for having provided extensive comments that improved the presentation of this paper. We would also like to thank Tushar Deepak Chandra for suggesting the name Heartbeat and for providing insightful comments regarding the simplifying assumption of Theorem 5.5 which helped us write the appendix. We are also grateful to the anonymous referees for their helpful comments.

REFERENCES

[1] M. K. Aguilera, W. Chen, and S. Toueg, *Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks*, Theoret. Comput. Sci., 220 (1999), pp. 3–30.

[2] M. K. Aguilera and S. Toueg, *Failure detection and randomization: A hybrid approach to solve consensus*, SIAM J. Comput., 28 (1999), pp. 890–903.

[3] Ö. Babaoğlu, R. Davoli, and A. Montresor, *Partitionable Group Membership: Specification and Algorithms*, Tech. Rep. UBLCS-97-1, Dept. of Computer Science, University of Bologna, Bologna, Italy, January 1997.

[4] Ö. Babaoğlu and S. Toueg, *Non-blocking atomic commitment*, in Distributed Systems, S. J. Mullender, ed., Addison-Wesley, Reading, MA, 1993, ch. 6.

[5] A. Basu, B. Charron-Bost, and S. Toueg, *Simulating reliable links with unreliable links in the presence of process crashes*, in Proceedings of the 10th International Workshop on Distributed Algorithms, Lecture Notes in Comput. Sci. 1151, Springer-Verlag, New York, 1996, pp. 105–122.

[6] R. Bazzi and G. Neiger, *Simulating crash failures with many faulty processors*, in Proceedings of the 6th International Workshop on Distributed Algorithms, A. Segal and S. Zaks, eds., Lecture Notes in Comput. Sci. 647, Springer-Verlag, New York, 1992, pp. 166–184.

[7] M. Ben-Or, *Another advantage of free choice: Completely asynchronous agreement protocols*, in Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing, ACM, New York, 1983, pp. 27–30.

[8] O. Biran, S. Moran, and S. Zaks, *A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor*, in Proceedings of the 7th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1988, pp. 263–275.

[9] G. Bracha and S. Toueg, *Asynchronous consensus and broadcast protocols*, J. ACM, 32 (1985), pp. 824–840.

[10] T. D. Chandra, *private communication*, Cornell University, Ithaca, NY, 1997.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg, *The weakest failure detector for solving consensus*, J. ACM, 43 (1996), pp. 685–722.

[12] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, J. ACM, 43 (1996), pp. 225–267.

[13] S. Chaudhuri, *More* choices *allow more* faults*: Set consensus problems in totally asynchronous systems*, Inform. and Comput., 105 (1993), pp. 132–158.

[14] B. Chor, M. Merritt, and D. B. Shmoys, *Simple constant-time consensus protocols in realistic failure models*, J. ACM, 36 (1989), pp. 591–614.

[15] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, *Reaching approximate agreement in the presence of faults*, J. ACM, 33 (1986), pp. 499–516.

[16] P. Feldman and S. Micali, *An Optimal Algorithm for Synchronous Byzantine Agreement*, Tech. Rep. MIT/LCS/TM-425, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, J. ACM, 32 (1985), pp. 374–382.

[18] A. Gopal, *Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination*, Ph.D. thesis, Cornell University, Ithaca, NY, 1992.

[19] J. N. Gray, *Notes on database operating systems*, in Operating Systems: An Advanced Course, R. Bayer, R. M. Graham, and G. Seegmuller, eds., Lecture Notes in Comput. Sci. 66, Springer-Verlag, New York, 1978. Also appears as IBM Research Laboratory Technical report RJ2188.

[20] R. Guerraoui, *Revisiting the relationship between non-blocking atomic commitment and consensus*, in Proceedings of the 9th International Workshop on Distributed Algorithms, Lecture Notes in Comput. Sci. 1972, Springer-Verlag, New York, 1995, pp. 87–100.

[21] R. Guerraoui, M. Larrea, and A. Schiper, *Non blocking atomic commitment with an unreliable failure detector*, in Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems, IEEE Computer Society, Los Alamitos, CA, 1995, pp. 41–50.

[22] V. Hadzilacos, *On the relationship between the atomic commitment and consensus problems*, in Proceedings of the Workshop on Fault-Tolerant Distributed Computing, Lecture Notes in Comput. Sci. 448, Springer-Verlag, New York, 1986, pp. 201–208.

[23] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, in Distributed Systems, S. J. Mullender, ed., Addison-Wesley, Reading, MA, 1993, ch. 5, pp. 97–145. A revised and extended version appears in [24].

[24] V. Hadzilacos and S. Toueg, *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*, Tech. Rep. 94-1425, Department of Computer Science, Cornell University, Ithaca, New York, 1994.

[25] M. Herlihy and N. Shavit, *The asynchronous computability theorem for t-resilient tasks*, in Proceedings of the 25th ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 111–120.

[26] M. Hurfin, A. Mostefaoui, and M. Raynal, *Consensus in Asynchronous Systems where Processes Can Crash and Recover*, Tech. Rep. 1144, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Rennes, France, 1997.

[27] R. Koo and S. Toueg, *Effects of message loss on the termination of distributed protocols*, Inform. Process. Lett., 27 (1988), pp. 181–188.

[28] W.-K. Lo and V. Hadzilacos, *Using failure detectors to solve consensus in asynchronous shared-memory systems*, in Proceedings of the 8th International Workshop on Distributed Algorithms, Terschelling, The Netherlands, 1994, pp. 280–295.

[29] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[30] Y. Moses and G. Roth, *On reliable message diffusion*, in Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1989, pp. 119–128.

[31] G. Neiger and S. Toueg, *Automatically increasing the fault-tolerance of distributed algorithms*, J. Algorithms, 11 (1990), pp. 374–419.

[32] R. Oliveira, R. Guerraoui, and A. Schiper, *Consensus in the Crash-Recover Model*, Tech. Rep. 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, 1997.

[33] M. Rabin, *Randomized Byzantine generals*, in Proceedings of the 24th Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 403–409.

[34] L. S. Sabel and K. Marzullo, *Election vs. Consensus in Asynchronous Systems*, Tech. Rep. 95-1488, Department of Computer Science, Cornell University, Ithaca, New York, 1995.

[35] R. van Renesse, *private communication*, Cornell University, Ithaca, NY, 1997.

SIAM J. COMPUT.
Vol. 29, No. 6, pp. 2074–2097

GADGETS, APPROXIMATION, AND LINEAR PROGRAMMING[*]

LUCA TREVISAN[†], GREGORY B. SORKIN[‡], MADHU SUDAN[§], AND DAVID P. WILLIAMSON[‡]

Abstract. We present a linear programming-based method for finding "gadgets," i.e., combinatorial structures reducing constraints of one optimization problem to constraints of another. A key step in this method is a simple observation which limits the search space to a *finite* one. Using this new method we present a number of new, computer-constructed gadgets for several different reductions. This method also answers a question posed by Bellare, Goldreich, and Sudan [*SIAM J. Comput.,* 27 (1998), pp. 804–915] of how to prove the optimality of gadgets: linear programming duality gives such proofs.

The new gadgets, when combined with recent results of Håstad [*Proceedings of the* 29*th ACM Symposium on Theory of Computing*, 1997, pp. 1–10], improve the known inapproximability results for MAX CUT and MAX DICUT, showing that approximating these problems to within factors of $16/17+\epsilon$ and $12/13+\epsilon$, respectively, is NP-hard for every $\epsilon > 0$. Prior to this work, the best-known inapproximability thresholds for both problems were 71/72 (M. Bellare, O. Goldreich, and M. Sudan [*SIAM J. Comput.*, 27 (1998), pp. 804–915]). Without using the gadgets from this paper, the best possible hardness that would follow from Bellare, Goldreich, and Sudan and Håstad is 18/19. We also use the gadgets to obtain an improved approximation algorithm for MAX3 SAT which guarantees an approximation ratio of .801. This improves upon the previous best bound (implicit from M. X. Goemans and D. P. Williamson [*J. ACM*, 42 (1995), pp. 1115–1145]; U. Feige and M. X. Goemans [*Proceedings of the Third Israel Symposium on Theory of Computing and Systems*, 1995, pp. 182–189]) of .7704.

Key words. combinatorial optimization, approximation algorithms, reductions, intractability, NP-completeness, probabilistic proof systems

AMS subject classification. 68Q15

PII. S0097539797328847

1. Introduction.

A "gadget" is a finite combinatorial structure which translates a given constraint of one optimization problem into a set of constraints of a second optimization problem. A classical example is in the reduction from 3SAT to MAX 2SAT, due to Garey, Johnson, and Stockmeyer [6]. Given an instance of 3SAT on variables $X_1, \ldots, X_n$ and with clauses $C_1, \ldots, C_m$, the reduction creates an instance of MAX 2SAT on the original or "primary" variables $X_1, \ldots, X_n$ along with new or "auxiliary" variables $Y^1, \ldots, Y^m$. The clauses of the MAX 2SAT instance are obtained by replacing each clause of length 3 in the 3SAT instance with a "gadget," in this case a collection of 10 2SAT clauses. For example, the clause $C_k = X_1 \vee X_2 \vee X_3$ would be replaced with the following 10 clauses on the variables X_1, X_2, X_3 and a

[*]Received by the editors October 10, 1997; accepted for publication (in revised form) August 30, 1999; published electronically April 18, 2000. An extended abstract of this paper appears in the Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, Burlington, VT, October 14–16, 1996, pp. 617–626.

http://www.siam.org/journals/sicomp/29-6/32884.html

[†]Department of Computer Science, Columbia University, 1214 Amsterdam Avenue, New York, NY 10027 (luca@cs.columbia.edu). Part of this work was done while the author was at the University of Rome "La Sapienza" and visiting IBM Research.

[‡]IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (sorkin@watson.ibm.com, dpw@watson.ibm.com).

[§]Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, MA 02139 (madhu@mit.edu). The work of this author was supported in part by an Alfred P. Sloan Foundation fellowship. Part of this work was done while the author was at the IBM T. J. Watson Research Center.

new auxiliary variable Y^k:

$$X_1,\ X_2,\ X_3,\ \neg X_1 \vee \neg X_2,\ \neg X_2 \vee \neg X_3,\ \neg X_3 \vee \neg X_1,\\ Y^k,\ X_1 \vee \neg Y^k,\ X_2 \vee \neg Y^k,\ X_3 \vee \neg Y^k.$$

The property satisfied by this gadget is that for any assignment to the primary variables, if clause C_k is satisfied, then 7 of the 10 new clauses can be satisfied by setting Y^k appropriately; otherwise only 6 of the 10 are satisfiable. (Notice that the gadget associated with each clause C_k uses its own auxiliary variable Y^k, and thus Y^k may be set independently of the values of variables not appearing in C_k's gadget.) Using this simple property of the gadget it is easy to see that the maximum number of clauses satisfied in the MAX 2SAT instance by any assignment is $7m$ if and only if the instance of 3SAT is satisfiable. This was used by [6] to prove the NP-hardness of solving MAX 2SAT. We will revisit the 3SAT-to-2SAT reduction in Lemma 6.5.

Starting with the work of Karp [12], gadgets have played a fundamental role in showing the hardness of optimization problems. They are the core of any reduction between combinatorial problems, and they retain this role in the spate of new results on the nonapproximability of optimization problems.

Despite their importance, the construction of gadgets has always been a "black art" with no general methods of construction known. In fact, until recently no one had even proposed a concrete definition of a gadget; Bellare, Goldreich, and Sudan [2] finally did so, with a view to quantifying the role of gadgets in nonapproximability results. Their definition is accompanied by a seemingly natural "cost" measure for a gadget. The more costly the gadget, the weaker the reduction. However, first, finding a gadget for a given reduction remained an ad hoc task. Second, it remained hard to prove that a gadget's cost was optimal.

This paper addresses these two issues. We show that for a large class of reductions, the space of potential gadgets that need to be considered is actually *finite*. This is not entirely trivial, and the proof depends on properties of the problem that is being reduced to. However, the method is very general and encompasses a large number of problems. An immediate consequence of the finiteness of the space is the existence of a search procedure to find an optimal gadget. But a naive search would be impracticably slow, and search-based proofs of the optimality (or the nonexistence) of a gadget would be monstrously large.

Instead, we show how to express the search for a gadget as a linear program (LP) whose constraints guarantee that the potential gadget is indeed valid, and whose objective function is the cost of the gadget. Central to this step is the idea of working with weighted versions of optimization problems rather than unweighted ones. (Weighted versions result in LPs, while unweighted versions would result in integer programs (IPs).) This seemingly helps only in showing hardness of weighted optimization problems, but a result due to Crescenzi, Silvestri, and Trevisan [3] shows that for a large class of optimization problems (including all the ones considered in this paper), the weighted versions are exactly as hard with respect to approximation as the unweighted ones. Therefore, working with a weighted version is as good as working with an unweighted one.

The LP representation has many benefits. First, we are able to search for much more complicated gadgets than is feasible manually. Second, we can use the theory of LP duality to present short(er) proofs of optimality of gadgets and nonexistence of gadgets. Last, we can solve relaxed or constrained versions of the LP to obtain upper and lower bounds on the cost of a gadget, which can be significantly quicker than solving the actual LP. Being careful in the relaxing/constraining process (and with a bit of luck), we can often get the bounds to match, thereby producing optimal gadgets with even greater efficiency!

Armed with this tool for finding gadgets (and an RS/6000, OSL, and often APL2[1]), we examine some of the known gadgets and construct many new ones. (In what follows we often talk of "gadgets reducing problem X to problem Y" when we mean "gadgets used to construct a reduction from problem X to problem Y.") Bellare, Goldreich, and Sudan [2] presented gadgets reducing the computation of a "verifier" for a probabilistically checkable proof system (PCP) to several problems, including MAX 3SAT, MAX 2SAT, and MAX CUT. We examine these in turn and show that the gadgets in [2] for MAX 3SAT and MAX 2SAT are optimal, but their MAX CUT gadget is not. We improve on the efficiency of the last, thereby improving on the factor to which approximating MAX CUT can be shown to be NP-hard. We also construct a new gadget for the MAX DICUT problem, thereby strengthening the known bound on its hardness. Plugging our gadget into the reduction (specifically Lemma 4.15) of [2] shows that approximating MAX CUT to within a factor of 60/61 is NP-hard, as is approximating MAX DICUT to within a factor of 44/45.[2] For both problems, the hardness factor proved in [2] was 71/72. The PCP machinery of [2] has since been improved by Håstad [9]. Our gadgets and Håstad's result show that, for every $\epsilon > 0$, approximating MAX CUT to within a factor of $16/17 + \epsilon$ is NP-hard, as is approximating MAX DICUT to within a factor of $12/13 + \epsilon$. Using Håstad's result in combination with the gadgets of [2] would have given a hardness factor of $18/19 + \epsilon$ for both problems for every $\epsilon > 0$.

Obtaining better reductions between problems can also yield improved approximation algorithms (if the reduction goes the right way!). We illustrate this point by constructing a gadget reducing MAX 3SAT to MAX 2SAT. Using this new reduction in combination with a technique of Goemans and Williamson [7, 8] and the state-of-the-art .931-approximation algorithm for MAX 2SAT due to Feige and Goemans [5] (which improves upon the previous .878-approximation algorithm of [8]), we obtain a .801-approximation algorithm for MAX 3SAT. The best result that could be obtained previously, by combining the technique of [7, 8] and the bound of [5], was .7704. (The best previously published result is a .769 approximation algorithm by Ono, Hirata, and Asano [14].)

Finally, our reductions have implications for PCPs. Let $\mathrm{PCP}_{c,s}[\log, q]$ be the class of languages that admit membership proofs that can be checked by a probabilistic verifier that uses a logarithmic number of random bits, reads at most q bits of the proof, accepts correct proofs of strings in the language with probability at least c, and accepts purported proofs of strings not in the language with probability at most s. We show the following: first, for any $\epsilon > 0$, there exist constants c and s, $c/s > 10/9 - \epsilon$, such that $\mathrm{NP} \subseteq \mathrm{PCP}_{c,s}[\log, 2]$; and second, for all c, s with $c/s > 2.7214$, $\mathrm{PCP}_{c,s}[\log, 3] \subseteq \mathrm{P}$. The best bound for the former result obtainable from [2, 9] is $22/21 - \epsilon$; the best previous bound for the latter was 4 [16].

All the gadgets we use are computer constructed. In the final section, we present an example of a lower bound on the performance of a gadget. The bound is not

[1]Respectively, an IBM RiscSystem/6000 workstation, the IBM Optimization Subroutine Library, which includes an LP package, and (not that we are partisan) IBM's APL2 programming language.

[2]Approximation ratios in this paper for maximization problems are less than 1 and represent the weight of the solution achievable by a polynomial-time algorithm divided by the weight of the optimal solution. This matches the convention used in [18, 7, 8, 5] and is the reciprocal of the measure used in [2].

computer constructed and cannot be, by the nature of the problem. The bound still relies on defining an LP that describes the optimal gadget and extracting the lower bound from the LP's dual.

Subsequent work. Subsequent to the original presentation of this work [17], the approximability results presented in this paper have been superseded. Karloff and Zwick [10] present a 7/8-approximation algorithm for MAX 3SAT. This result is tight unless NP = P [9]. The containment result $\text{PCP}_{c,s}[\log, 3] \subseteq \text{P}$ has also been improved by Zwick [19] and shown to hold for any $c/s \geq 2$. This result is also tight, again by [9]. Finally, the gadget construction methods of this paper have found at least two more applications. Håstad [9] and Zwick [19] use gadgets constructed by these techniques to show hardness results for two problems they consider: MAX 2LIN and MAX NAE3SAT, respectively.

Version. An extended abstract of this paper appeared as [17]. This version corrects some errors, pointed out by Karloff and Zwick [11], in the extended abstract. This version also presents inapproximability results resting on the improved PCP constructions of Håstad [9], while mentioning the results that could be obtained otherwise.

Organization of this paper. The next section introduces precise definitions which formalize the preceding outline. Section 3 presents the finiteness proof and the LP-based search strategy. Section 4 contains negative (nonapproximability) results and the gadgets used to derive them. Section 5 briefly describes our computer system for generating gadgets. Section 6 presents the positive result for approximating MAX 3SAT. Section 7 presents proofs of optimality of the gadgets for some problems and lower bounds on the costs of others. It includes a mix of computer-generated and hand-generated lower bounds.

2. Definitions.

We begin with some definitions we will need before giving the definition of a gadget from [2]. In what follows, for any positive integer n, let $[n]$ denote the set $\{1, \ldots, n\}$.

DEFINITION 2.1. *A* (k-ary) constraint function *is a Boolean function* $f : \{0,1\}^k \to \{0,1\}$.

We refer to k as the arity of a k-ary constraint function f. When it is applied to variables $X_1, \ldots, X_k$ (see the following definitions), the function f is thought of as imposing the constraint $f(X_1, \ldots, X_k) = 1$.

DEFINITION 2.2. *A* constraint family $\mathcal{F}$ *is a collection of constraint functions. The* arity *of* $\mathcal{F}$ *is the maximum of the arity of the constraint functions in* $\mathcal{F}$.

DEFINITION 2.3. *A* constraint C *over a variable set* $X_1, \ldots, X_n$ *is a pair* $C = (f, (i_1, \ldots, i_k))$, *where* $f : \{0,1\}^k \to \{0,1\}$ *is a constraint function and* $i_1, \ldots, i_k$ *are distinct members of* $[n]$. *The constraint* C *is said to be* satisfied *by an assignment* $\vec{a} = a_1, \ldots, a_n$ *to* $X_1, \ldots, X_n$ *if* $C(a_1, \ldots, a_n) \overset{\text{def}}{=} f(a_{i_1}, \ldots, a_{i_k}) = 1$. *We say that constraint* C *is* from $\mathcal{F}$ *if* $f \in \mathcal{F}$.

Constraint functions, constraint families, and constraints are of interest due to their defining role in a variety of NP optimization problems.

DEFINITION 2.4. *For a finitely specified constraint family* $\mathcal{F}$, MAX $\mathcal{F}$ *is the optimization problem given by*

INPUT: *An instance consisting of* m *constraints* $C_1, \ldots, C_m$, *on* n *Boolean variables* $X_1, \ldots, X_n$, *with nonnegative real weights* $w_1, \ldots, w_m$. *(An instance is thus a triple* $(\vec{X}, \vec{C}, \vec{w})$.*)*

GOAL: *Find an assignment* $\vec{b}$ *to the variables* $\vec{X}$ *which maximizes the weight* $\sum_{j=1}^{m} w_j C_j(\vec{b})$ *of satisfied constraints.*

Constraint functions and families and the class $\{\text{MAX } \mathcal{F} \mid \mathcal{F}\}$ allow descriptions of optimization problems and reductions in a uniform manner. For example, if $\mathcal{F} =$ 2SAT is the constraint family consisting of all constraint functions of arity at most 2 that can be expressed as the disjunction of up to 2 literals, then MAX 2SAT is the corresponding MAX $\mathcal{F}$ problem. Similarly MAX 3SAT is the MAX $\mathcal{F}$ problem defined using the constraint family $\mathcal{F} =$ 3SAT consisting of all constraint functions of arity up to 3 that can be expressed as the disjunction of up to 3 literals.

One of the motivations for this work is to understand the "approximability" of many central optimization problems that can be expressed as MAX $\mathcal{F}$ problems, including MAX 2SAT and MAX 3SAT. For $\beta \in [0, 1]$, an algorithm $\mathcal{A}$ is said to be a β-approximation algorithm for the MAX $\mathcal{F}$ problem if on every instance $(\vec{X}, \vec{C}, \vec{w})$ of MAX $\mathcal{F}$ with n variables and m constraints, $\mathcal{A}$ outputs an assignment $\vec{a}$ subject to (s.t.) $\sum_{j=1}^{m} w_j C_j(\vec{a}) \geq \beta \max_{\vec{b}}\{\sum_{j=1}^{m} w_j C_j(\vec{b})\}$. We say that the problem MAX $\mathcal{F}$ is β-approximable if there exists a *polynomial time*-bounded algorithm $\mathcal{A}$ that is a β-approximation algorithm for MAX $\mathcal{F}$. We say that MAX $\mathcal{F}$ is hard to approximate to within a factor β (β-inapproximable) if the existence of a polynomial-time β-approximation algorithm for MAX $\mathcal{F}$ implies NP = P.

Recent research has yielded a number of new approximability results for several MAX $\mathcal{F}$ problems (see [7, 8]) and a number of new results yielding hardness of approximations (see [2, 9]). One of our goals is to construct efficient reductions between MAX $\mathcal{F}$ problems that allow us to translate "approximability" and "inapproximability" results. As we saw in the opening example, such reductions may be constructed by constructing "gadgets" reducing one constraint family to another. More specifically, the example shows how a reduction from 3SAT to 2SAT results from the availability, for every constraint function f in the family 3SAT, of a gadget reducing f to the family 2SAT. This notion of a gadget reducing a constraint function f to a constraint family $\mathcal{F}$ is formalized in the following definition.

Definition 2.5 (gadget [2]). *For $\alpha \in \mathcal{R}^+$, a constraint function $f : \{0,1\}^k \to \{0,1\}$, and a constraint family $\mathcal{F}$, an* α-gadget *(or "gadget with performance α") reducing f to $\mathcal{F}$ is a set of variables $Y_1, \ldots, Y_n$, a finite collection of real weights $w_j \geq 0$, and associated constraints C_j from $\mathcal{F}$ over* primary variables $X_1, \ldots, X_k$ *and* auxiliary variables $Y_1, \ldots, Y_n$, *with the property that, for Boolean assignments $\vec{a}$ to $X_1, \ldots, X_k$ and $\vec{b}$ to $Y_1, \ldots, Y_n$, the following are satisfied:*

$$(\forall \vec{a} : f(\vec{a}) = 1)\ (\forall \vec{b}) : \sum_j w_j C_j(\vec{a}, \vec{b}) \leq \alpha, \tag{2.1}$$

$$(\forall \vec{a} : f(\vec{a}) = 1)\ (\exists \vec{b}) : \sum_j w_j C_j(\vec{a}, \vec{b}) = \alpha, \tag{2.2}$$

$$(\forall \vec{a} : f(\vec{a}) = 0)\ (\forall \vec{b}) : \sum_j w_j C_j(\vec{a}, \vec{b}) \leq \alpha - 1. \tag{2.3}$$

The gadget is strict *if, in addition,*

$$(\forall \vec{a} : f(\vec{a}) = 0)\ (\exists \vec{b}) : \sum_j w_j C_j(\vec{a}, \vec{b}) = \alpha - 1. \tag{2.4}$$

We use the shorthand notation $\Gamma = (\vec{Y}, \vec{C}, \vec{w})$ to denote the gadget described above.

It is straightforward to verify that the introductory example yields a strict 7-gadget reducing the constraint function $f(X_1, X_2, X_3) = X_1 \vee X_2 \vee X_3$ to the family 2SAT.

Observe that an α-gadget $\Gamma = (\vec{Y}, \vec{C}, \vec{w})$ can be converted into an $\alpha' > \alpha$ gadget by "rescaling," i.e., multiplying every entry of the weight vector $\vec{w}$ by α'/α (although strictness is not preserved). This indicates that a "strong" gadget is one with a small α; in the extreme, a 1-gadget would be the "optimal" gadget. This intuition will be confirmed in the role played by gadgets in the construction of reductions. Before describing this, we first list the constraints and constraint families that are of interest to us.

For convenience we now give a comprehensive list of all the constraints and constraint families used in this paper.

DEFINITION 2.6.

- Parity check (PC) *is the constraint family* $\{\mathrm{PC}_0, \mathrm{PC}_1\}$, *where, for* $i \in \{0,1\}$, PC_i *is defined as follows:*

$$\mathrm{PC}_i(a,b,c) = \begin{cases} 1 & \textit{if } a \oplus b \oplus c = i, \\ 0 & \textit{otherwise.} \end{cases}$$

Henceforth we will simply use terms such as MAX PC *to denote the optimization problem* MAX $\mathcal{F}$, *where* $\mathcal{F} = \mathrm{PC}$. MAX PC *(referred to as MAX 3LIN in* [9]*) is the source of all our inapproximability results.*

- *For any* $k \geq 1$, Exactly-k-SAT (EkSAT) *is the constraint family* $\{f : \{0,1\}^k \to \{0,1\} : |\{\vec{a} : f(\vec{a}) = 0\}| = 1\}$, *that is, the set of* k*-ary* disjunctive *constraints.*
- *For any* $k \geq 1$, kSAT *is the constraint family* $\bigcup_{l \in [k]} \mathrm{E}l\mathrm{SAT}$.
- SAT *is the constraint family* $\bigcup_{l \geq 1} \mathrm{E}l\mathrm{SAT}$.

The problems MAX 3SAT, MAX 2SAT, *and* MAX SAT *are by now classical optimization problems. They were considered originally in* [6]*; subsequently their central role in approximation was highlighted in* [15]*; and recently, novel approximation algorithms were developed in* [7, 8, 5]. *The associated families are typically the targets of gadget constructions in this paper. Shortly, we will describe a lemma which connects the inapproximability of* MAX $\mathcal{F}$ *to the existence of gadgets reducing* PC_0 *and* PC_1 *to* $\mathcal{F}$. *This method has so far yielded in several cases tight, and in other cases the best-known, inapproximability results for* MAX $\mathcal{F}$ *problems.*

In addition to 3SAT*'s use as a target, its members are also used as sources; gadgets reducing members of* MAX 3SAT *to* MAX 2SAT *help give an improved* MAX 3SAT *approximation algorithm.*

- 3-Conjunctive SAT (3ConjSAT) *is the constraint family* $\{f_{000}, f_{100}, f_{110}, f_{111}\}$, *where*
  (1) $f_{000}(a,b,c) = a \wedge b \wedge c$,
  (2) $f_{001}(a,b,c) = a \wedge b \wedge \neg c$,
  (3) $f_{011}(a,b,c) = a \wedge \neg b \wedge \neg c$,
  (4) $f_{111}(a,b,c) = \neg a \wedge \neg b \wedge \neg c$.

Members of 3ConjSAT *are sources in gadgets reducing them to* 2SAT. *These gadgets enable a better approximation algorithm for the* MAX 3ConjSAT *problem, which in turn sheds light on the class* $\mathrm{PCP}_{c,s}[\log, 3]$.

- CUT: $\{0,1\}^2 \to \{0,1\}$ *is the constraint function given by* $\mathrm{CUT}(a,b) = a \oplus b$. CUT/0 *is the family of constraints* $\{\mathrm{CUT}, \mathrm{T}\}$, *where* $\mathrm{T}(a) = 0 \oplus a = a$. CUT/1 *is the family of constraints* $\{\mathrm{CUT}, \mathrm{F}\}$, *where* $\mathrm{F}(a) = 1 \oplus a = \neg a$.

MAX CUT *is again a classical optimization problem. It has attracted attention due to the recent result of Goemans and Williamson* [8] *providing a* 0.878 *approximation algorithm. An observation from Bellare, Goldreich, and Sudan* [2] *shows that the approximabilities of* MAX CUT/0, MAX CUT/1, *and* MAX CUT *are all identical; this* *is also formalized in Proposition* 4.1 *below. Hence* MAX CUT/0 *becomes the target of gadget constructions in this paper, allowing us to get inapproximability results for these three problems.*

- DICUT: $\{0,1\}^2 \to \{0,1\}$ *is the constraint function given by* $\text{DICUT}(a,b) = \neg a \wedge b$.

MAX DICUT *is another optimization problem to which the algorithmic results of* [8, 5] *apply. Gadgets whose target is* DICUT *will enable us to get inapproximability results for* MAX DICUT.

- 2CSP *is the constraint family consisting of all* 16 *binary functions, i.e.,* $\text{2CSP} = \{f : \{0,1\}^2 \to \{0,1\}\}$.

MAX 2CSP *was considered in* [5]*, which gives a* 0.859 *approximation algorithm; here we provide inapproximability results.*

- Respect of monomial basis check (RMBC) *is the constraint family* $\{\text{RMBC}_{ij} | i,j \in \{0,1\}\}$, *where*

$$\text{RMBC}_{ij}(a,b,c,d) = \begin{cases} 1 & \textit{if } a = 0 \textit{ and } b = c \oplus i, \\ 1 & \textit{if } a = 1 \textit{ and } b = d \oplus j, \\ 0 & \textit{otherwise.} \end{cases}$$

  RMBC_{00} *may be thought of as the test* $(c,d)[a] \overset{?}{=} b$, RMBC_{01} *as the test* $(c,\neg d)[a] \overset{?}{=} b$, RMBC_{10} *as the test* $(\neg c, d)[a] \overset{?}{=} b$, *and* RMBC_{11} *as the test* $(\neg c, \neg d)[a] \overset{?}{=} b$, *where the notation* $(v_1, \ldots, v_n)[i]$ *refers to the* $(i+1)$*'st coordinate of the vector* $(v_1, \ldots, v_n)$.

Our original interest in RMBC *came from the work of Bellare, Goldreich, and Sudan* [2]*, which derived hardness results for* MAX $\mathcal{F}$ *using gadgets reducing every constraint function in* PC *and* RMBC *to* $\mathcal{F}$. *This work has been effectively superseded by Håstad's* [9]*, which only requires gadgets reducing members of* PC *to* $\mathcal{F}$. *However we retain some of the discussion regarding gadgets with* RMBC *functions as a source, since these constructions were significantly more challenging, and some of the techniques applied to overcome the challenges may be applicable in other gadget constructions. A summary of all the gadgets we found, with their performances and lower bounds, is given in Table* 2.1.

We now put forth a theorem, essentially from [2] (and obtainable as a generalization of its Lemmas 4.7 and 4.15), that relates the existence of gadgets with $\mathcal{F}$ as target to the hardness of approximating MAX $\mathcal{F}$. Since we will not be using this theorem except as a motivation for studying the family RMBC, we do not prove it here.

THEOREM 2.7. *For any family* $\mathcal{F}$, *if there exists an* α_1*-gadget reducing every function in* PC *to* $\mathcal{F}$ *and an* α_2*-gadget reducing every function in* RMBC *to* $\mathcal{F}$, *then for any* $\epsilon > 0$, *MAX* $\mathcal{F}$ *is hard to approximate to within* $1 - \frac{.15}{.6\alpha_1 + .4\alpha_2} + \epsilon$.

In this paper we will use the following, stronger, result by Håstad.

THEOREM 2.8 (see [9]). *For any family* $\mathcal{F}$, *if there exists an* α_0*-gadget reducing* PC_0 *to* $\mathcal{F}$ *and an* α_1*-gadget reducing* PC_1 *to* $\mathcal{F}$, *then for any* $\epsilon > 0$, MAX $\mathcal{F}$ *is hard to approximate to within* $1 - \frac{1}{\alpha_0 + \alpha_1} + \epsilon$.

Thus, using CUT/0, DICUT, 2CSP, EkSAT, and kSAT as the target of gadget constructions from PC_0 and PC_1, we can show the hardness of MAX CUT, MAX DICUT, MAX 2CSP, MAX EkSAT, and MAX kSAT, respectively. Furthermore, minimizing the value of α in the gadgets gives better hardness results.

Table 2.1

All gadgets described are provably optimal and strict. The sole exception (†) is the best possible strict gadget; there is a nonstrict 3-gadget. All "previous" results quoted are interpretations of the results in [2], *except the gadget reducing* 3*SAT to* 2*SAT, which is due to* [6], *and the gadget reducing* PC *to* 3*SAT, which is folklore.*

Source $f \longrightarrow$ target $\mathcal{F}$	Previous α	Our α	Lower bound
3SAT $\longrightarrow$ 2SAT	7	3.5	3.5
3ConjSAT $\longrightarrow$ 2SAT(†)		4	4
PC $\longrightarrow$ 3SAT	4		4
PC $\longrightarrow$ 2SAT	11		11
PC $\longrightarrow$ 2CSP	11	5	5
$\text{PC}_0 \longrightarrow$ CUT/0	10	8	8
$\text{PC}_0 \longrightarrow$ DICUT		6.5	6.5
$\text{PC}_1 \longrightarrow$ CUT/0	9		9
$\text{PC}_1 \longrightarrow$ DICUT		6.5	6.5
RMBC $\longrightarrow$ 2CSP	11	5	5
RMBC $\longrightarrow$ 3SAT	4		4
RMBC $\longrightarrow$ 2SAT	11		11
$\text{RMBC}_{00} \longrightarrow$ CUT/0	11	8	8
$\text{RMBC}_{00} \longrightarrow$ DICUT		6	6
$\text{RMBC}_{01} \longrightarrow$ CUT/0	12	8	8
$\text{RMBC}_{01} \longrightarrow$ DICUT		6.5	6.5
$\text{RMBC}_{10} \longrightarrow$ CUT/0	12	9	9
$\text{RMBC}_{10} \longrightarrow$ DICUT		6.5	6.5
$\text{RMBC}_{11} \longrightarrow$ CUT/0	12	9	9
$\text{RMBC}_{11} \longrightarrow$ DICUT		7	7

3. The basic procedure.

The key aspect of making the gadget search spaces finite is to limit the number of auxiliary variables by showing that duplicates (in a sense to be clarified) can be eliminated by means of proper *substitutions.* In general, this is possible if the target of the reduction is a "hereditary" family as defined below.

Definition 3.1. *A constraint family $\mathcal{F}$ is* hereditary *if for any $f \in \mathcal{F}$ of arity k and any two indices $i, j \in [k]$, the function f when restricted to $X_i \equiv X_j$ and considered as a function of $k-1$ variables is identical (up to the order of the arguments) to some other function $f' \in \mathcal{F} \cup \{0, 1\}$ (where 0 and 1 denote the constant functions).*

Definition 3.2. *A family $\mathcal{F}$ is* complementation closed *if it is hereditary and, for any $f \in \mathcal{F}$ of arity k and any index $i \in [k]$, the function f' given by $f'(X_1, \ldots, X_k) = f(X_1, \ldots, X_{i-1}, \neg X_i, X_{i+1}, \ldots, X_k)$ is contained in $\mathcal{F}$.*

Definition 3.3 (partial gadget). *For $\alpha \in \mathcal{R}^+$, $S \subseteq \{0, 1\}^k$, a constraint function $f : \{0, 1\}^k \to \{0, 1\}$, and a constraint family $\mathcal{F}$, an S-*partial α-gadget *(or "S-partial gadget with performance α") reducing f to $\mathcal{F}$ is a finite collection of constraints $C_1, \ldots, C_m$ from $\mathcal{F}$ over primary variables $X_1, \ldots, X_k$ and finitely many auxiliary variables $Y_1, \ldots, Y_n$, and a collection of nonnegative real weights $w_1, \ldots, w_m$, with the property that, for Boolean assignments $\vec{a}$ to $X_1, \ldots, X_k$ and $\vec{b}$ to $Y_1, \ldots, Y_n$, the following are satisfied:*

$$(\forall \vec{a} \in \{0,1\}^k : f(\vec{a}) = 1)\ (\forall \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) \le \alpha, \tag{3.1}$$

$$(\forall \vec{a} \in S : f(\vec{a}) = 1)\ (\exists \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) = \alpha, \tag{3.2}$$

$$(\forall \vec{a} \in \{0,1\}^k : f(\vec{a}) = 0)\ (\forall \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) \le \alpha - 1, \tag{3.3}$$

$$(\forall \vec{a} \in S : f(\vec{a}) = 0)\ (\exists \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) = \alpha - 1. \tag{3.4}$$

We use the shorthand notation $\Gamma = (\vec{Y}, \vec{C}, \vec{w})$ *to denote the partial gadget.*

The following proposition follows immediately from the definitions of a gadget and a partial gadget.

PROPOSITION 3.4. *For a constraint function* $f : \{0,1\}^k \to \{0,1\}$, *let* $S_1 = \{\vec{a} \in \{0,1\}^k : f(\vec{a}) = 1\}$ *and let* $S_2 = \{0,1\}^k$. *Then for every* $\alpha \in \mathcal{R}^+$ *and constraint family* $\mathcal{F}$,

(1) *an* S_1*-partial* α*-gadget reducing* f *to* $\mathcal{F}$ *is an* α*-gadget reducing* f *to* $\mathcal{F}$*;*

(2) *an* S_2*-partial* α*-gadget reducing* f *to* $\mathcal{F}$ *is a strict* α*-gadget reducing* f *to* $\mathcal{F}$*.*

DEFINITION 3.5. *For* $\alpha \ge 1$ *and* $S \subseteq \{0,1\}^k$, *let* $\Gamma = (\vec{Y}, \vec{C}, \vec{w})$ *be an* S*-partial* α*-gadget reducing a constraint* $f : \{0,1\}^k \to \{0,1\}$ *to a constraint family* $\mathcal{F}$. *We say that the function* $b : S \to \{0,1\}^n$ *is a* witness *for the partial gadget, witnessing the set* S, *if* $b(\vec{a})$ *satisfies equations* (3.2) *and* (3.4). *Specifically,*

$$(\forall \vec{a} \in S : f(\vec{a}) = 1) : \sum_{j=1}^{m} w_j C_j(\vec{a}, b(\vec{a})) = \alpha \qquad \text{and}$$

$$(\forall \vec{a} \in S : f(\vec{a}) = 0) : \sum_{j=1}^{m} w_j C_j(\vec{a}, b(\vec{a})) = \alpha - 1.$$

The witness function can also be represented as an $|S| \times (k+n)$ *matrix* W_b *whose rows are the vectors* $(\vec{a}, b(\vec{a}))$. *Notice that the columns of the matrix correspond to the variables of the gadget, with the first* k *columns corresponding to primary variables, and the last* n *corresponding to auxiliary variables. In what follows we shall often prefer the matrix notation.*

DEFINITION 3.6. *For a set* $S \subseteq \{0,1\}^k$, *let* M_S *be the matrix whose rows are the vectors* $\vec{a} \in S$, *let* k'_S *be the number of distinct columns in* M_S, *and let* k''_S *be the number of columns in* M_S *distinct up to complementation. Given a constraint* f *of arity* k *and a hereditary constraint family* $\mathcal{F}$ *that is* not *complementation closed, an* $(S, f, \mathcal{F})$-canonical witness matrix *(for an* S*-partial gadget reducing* f *to* $\mathcal{F}$*) is the* $|S| \times (2^{|S|} + k - k'_S)$ *matrix* W *whose first* k *columns correspond to the* k *primary variables and whose remaining columns are all possible column vectors that are distinct from one another and from the columns corresponding to the primary variables. If* $\mathcal{F}$ is *complementation closed, then a canonical witness matrix is the* $|S| \times (2^{|S|-1} + k - k''_S)$ *matrix* W *whose first* k *columns correspond to the* k *primary variables and whose remaining columns are all possible column vectors that are distinct up to complementation from one another and from the columns corresponding to the primary variables.*

The following lemma is the crux of this paper and establishes that the optimal gadget reducing a constraint function f to a hereditary family $\mathcal{F}$ is finite. To motivate the lemma, we first present an example, due to Karloff and Zwick [11], showing that this need not hold if the family $\mathcal{F}$ is not hereditary. Their counterexample has $f(a) = a$ and $\mathcal{F} = \{\mathrm{PC}_1\}$. Using k auxiliary variables, $Y_1, \ldots, Y_k$, one may construct a gadget for the constraint X using the constraints $X \oplus Y_i \oplus Y_j$, $1 \leq i < j \leq k$, with each constraint having the same weight. For an appropriate choice of this weight it may be verified that this yields a $(2 - 2/k)$-gadget for even k; thus the performance tends to 2 in the limit. On the other hand it can be shown that any gadget with k auxiliary variables has performance at most $2 - 2^{1-k}$; thus no finite gadget achieves the limit. It is clear that for this example the lack of hereditariness is critical: any hereditary family containing PC_1 would also contain f, providing a trivial 1-gadget.

To see why the hereditary property helps in general, consider an α-gadget Γ reducing f to $\mathcal{F}$, and let W be a witness matrix for Γ. Suppose two columns of W, corresponding to auxiliary variables Y_1 and Y_2 of Γ, are identical. Then we claim that Γ does not really need the variable Y_2. In every constraint containing Y_2, replace it with Y_1 to yield a new collection of weighted constraints. By the hereditary property of $\mathcal{F}$, all the resulting constraints are from $\mathcal{F}$. And, the resulting instance satisfies all the properties of an α-gadget. (The universal properties follow trivially, while the existential properties follow from the fact that in the witness matrix Y_1 and Y_2 have the same assignment.) Thus this collection of constraints forms a gadget with fewer variables and performance at least as good. The finiteness follows from the fact that a witness matrix with distinct columns has a bounded number of columns. The following lemma formalizes this argument. In addition it also describes the canonical witness matrix for an optimal gadget—something that will be of use later.

LEMMA 3.7. *For $\alpha \geq 1$, set $S \subset \{0,1\}^k$, constraint $f : \{0,1\}^k \to \{0,1\}$, and hereditary constraint family $\mathcal{F}$, if there exists an S-partial α-gadget Γ reducing f to $\mathcal{F}$ with witness matrix W, then for any $(S, f, \mathcal{F})$-canonical witness matrix W' and some $\alpha' \leq \alpha$, there exists an α'-gadget Γ' reducing f to $\mathcal{F}$ with W' as a witness matrix.*

Proof. We first consider the case where $\mathcal{F}$ is not complementation closed. Let $\Gamma = (\vec{Y}, \vec{C}, \vec{w})$ be an S-partial α-gadget reducing f to $\mathcal{F}$ and let W be a witness matrix for Γ. We create a gadget Γ' with $n' = 2^{|S|} - k'$ auxiliary variables $Y'_1, \ldots, Y'_{n'}$, one associated with each column of the matrix W' other than the first k.

With each variable Y_i of Γ we associate a variable Z such that the column corresponding to Y_i in W is the same as the column corresponding to Z in W'. Notice that Z may be one of the primary variables $X_1, \ldots, X_k$ or one of the auxiliary variables $Y'_1, \ldots, Y'_{n'}$. By definition of a canonical witness, such a column and hence variable Z does exist.

Now for every constraint C_j on variables $Y_{i_1}, \ldots, Y_{i_k}$ in Γ with weight w_j, we introduce the constraint C_j on variables $Y'_{i'_1}, \ldots, Y'_{i'_k}$ in Γ' with weight w_j where $Y'_{i'_l}$ is the variable associated with Y_{i_l}. Notice that in this process the variables involved with a constraint do not necessarily remain distinct. This is where the *hereditary property* of $\mathcal{F}$ is used to ensure that a constraint $C_j \in \mathcal{F}$, when applied to a tuple of nondistinct variables, remains a constraint in $\mathcal{F}$. In the process we may arrive at some constraints which are either always satisfied or never satisfied. For the time being, we assume that the constraints $\mathbf{0}$ and $\mathbf{1}$ are contained in $\mathcal{F}$, so this occurrence does not cause a problem. Later we show how this assumption is removed.

This completes the description of Γ'. To verify that Γ' is indeed an S-partial α-gadget, we notice that the universal constraints (conditions (3.1) and (3.3) in Definition 3.3) are trivially satisfied, since Γ' is obtained from Γ by renaming some variables and possibly identifying some others. To see that the existential constraints (conditions (3.2) and (3.4) in Definition 3.3) are satisfied, notice that the assignments to the variables $\vec{Y}$ that witness these conditions in Γ are allowable assignments to the corresponding variables in $\vec{Y}'$ and in fact this is what dictated our association of variables in $\vec{Y}$ to the variables in $\vec{Y}'$. Thus Γ' is indeed an S-partial α-gadget reducing f to $\mathcal{F}$, and, by construction, has W' as a witness matrix.

Last, we remove the assumption that Γ' must include constraints $\mathbf{0}$ and $\mathbf{1}$. Any constraints $\mathbf{0}$ can be safely thrown out of the gadget without changing any of the parameters, since such constraints are never satisfied. On the other hand, constraints $\mathbf{1}$ do affect α. If we throw away a $\mathbf{1}$ constraint of weight w_j, this reduces the total weight of satisfied clauses in every assignment by w_j. Throwing away all such constraints reduces α by the total weight of the $\mathbf{1}$ constraints, producing a gadget of (improved) performance $\alpha' \leq \alpha$.

Finally, we describe the modifications required to handle the case where $\mathcal{F}$ is complementation closed (in which case the definition of a canonical witness changes). Here, for each variable Y_i and its associated column of W, either there is an equal column in W', in which case we replace Y_i with the column's associated variable $Y'_{i'}$, or there is a complementary column in W', in which case we replace Y_i with the negation of the column's associated variable, $\neg Y'_{i'}$, The rest of the construction proceeds as above, and the proof of correctness is the same. □

It is an immediate consequence of Lemma 3.7 that an optimum gadget reducing a constraint function to a hereditary family does not need to use more than an explicitly bounded number of auxiliary variables.

COROLLARY 3.8. *Let f be a constraint function of arity k with s satisfying assignments. Let $\mathcal{F}$ be a constraint family and $\alpha \geq 1$ be such that there exists an α-gadget reducing f to $\mathcal{F}$.*

(1) *If $\mathcal{F}$ is hereditary, then there exists an α'-gadget with at most $2^s - k'$ auxiliary variables reducing f to $\mathcal{F}$, where $\alpha' \leq \alpha$ and k' is the number of distinct variables among the satisfying assignments of f.*

(2) *If $\mathcal{F}$ is complementation closed, then there exists an α'-gadget with at most $2^{s-1} - k''$ auxiliary variables reducing f to $\mathcal{F}$ for some $\alpha' \leq \alpha$, where k'' is the number of distinct variables, up to complementation, among the satisfying assignments of f.*

COROLLARY 3.9. *Let f be a constraint function of arity k. Let $\mathcal{F}$ be a constraint family and $\alpha \geq 1$ be such that there exists a strict α-gadget reducing f to $\mathcal{F}$.*

(1) *If $\mathcal{F}$ is hereditary, then there exists a strict α'-gadget with at most $2^{2^k} - k$ auxiliary variables reducing f to $\mathcal{F}$ for some $\alpha' \leq \alpha$.*

(2) *If $\mathcal{F}$ is complementation closed, then there exists a strict α'-gadget with at most $2^{2^k-1} - k$ auxiliary variables reducing f to $\mathcal{F}$ for some $\alpha' \leq \alpha$.*

We will now show how to cast the search for an optimum gadget as an LP.

DEFINITION 3.10. *For a constraint function f of arity k, constraint family $\mathcal{F}$, and $s \times (k+n)$ witness matrix M,* $\mathrm{LP}(f, \mathcal{F}, M)$ *is an LP defined as follows:*

- *Let $C_1, \ldots, C_m$ be all the possible distinct constraints that arise from applying a constraint function from $\mathcal{F}$ to a set of $n+k$ Boolean variables. Thus for every j, $C_j : \{0,1\}^{k+n} \to \{0,1\}$. The LP variables are $w_1, \ldots, w_m$, where w_j corresponds to the weight of the constraint C_j. Additionally the LP has one more variable α.*
- *Let $S \subseteq \{0,1\}^k$ and $b : S \to \{0,1\}^n$ be such that $M = W_b$ (i.e., M is the witness matrix corresponding to the witness function b for the set S). The LP inequalities correspond to the definition of an S-partial gadget:*

$$(3.5)\quad (\forall \vec{a} \in \{0,1\}^k : f(\vec{a}) = 1)\ (\forall \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) \leq \alpha,$$

$$(\forall \vec{a} \in S : f(\vec{a}) = 1) : \sum_{j=1}^{m} w_j C_j(\vec{a}, b(\vec{a})) = \alpha, \tag{3.6}$$

$$(\forall \vec{a} \in \{0,1\}^k : f(\vec{a}) = 0)\ (\forall \vec{b} \in \{0,1\}^n) : \sum_{j=1}^{m} w_j C_j(\vec{a}, \vec{b}) \le \alpha - 1, \tag{3.7}$$

$$(\forall \vec{a} \in S : f(\vec{a}) = 0) : \sum_{j=1}^{m} w_j C_j(\vec{a}, b(\vec{a})) = \alpha - 1. \tag{3.8}$$

Finally the LP has the inequalities $w_j \ge 0$.

- *The objective of the LP is to minimize α.*

PROPOSITION 3.11. *For any constraint function f of arity k, constraint family $\mathcal{F}$, and $s \times (k+n)$ witness matrix M witnessing the set $S \subseteq \{0,1\}^k$, if there exists an S-partial gadget reducing f to $\mathcal{F}$ with witness matrix M, then* LP$(f, \mathcal{F}, M)$ *finds such a gadget with the minimum possible α.*

Proof. The LP-generated gadget consists of k primary variables $X_1, \ldots, X_k$ corresponding to the first k columns of M; n auxiliary variables $Y_1, \ldots, Y_n$ corresponding to the remaining n columns of M; constraints $C_1, \ldots, C_m$ as defined in Definition 3.10; and weights $w_1, \ldots, w_m$ as returned by LP$(f, \mathcal{F}, M)$. By construction the LP solution returns the minimum possible α for which an S-partial α-gadget reducing f to $\mathcal{F}$ with witness M exists. ☐

THEOREM 3.12 (main). *Let f be a constraint function of arity k with s satisfying assignments. Let k' be the number of distinct variables of f and k'' be the number of distinct variables up to complementation. Let $\mathcal{F}$ be a hereditary constraint family with functions of arity at most l. Then*

- *if there exists an α-gadget reducing f to $\mathcal{F}$, then there exists such a gadget with at most v auxiliary variables, where $v = 2^{s-1} - k''$ if $\mathcal{F}$ is complementation closed and $v = 2^s - k'$ otherwise;*
- *if there exists a strict α-gadget reducing f to $\mathcal{F}$, then there exists such a gadget with at most v auxiliary variables, where $v = 2^{2^k - 1} - k''$ if $\mathcal{F}$ is complementation closed and $v = 2^{2^k} - k'$ otherwise.*

Furthermore such a gadget with smallest performance can be found by solving an LP with at most $|\mathcal{F}| \times (v+k)^l$ variables and 2^{v+k} constraints.

Remark 3.12. The sizes given above are upper bounds. In specific instances, the sizes may be much smaller. In particular, if the constraints of $\mathcal{F}$ exhibit symmetries, or are not all of the same arity, then the number of variables of the LP will be much smaller.

Proof. By Proposition 3.11 and Lemma 3.7, we have that LP$(f, \mathcal{F}, W_S)$ yields an optimal S-partial gadget if one exists. By Proposition 3.4 the setting $S = S_1 = \{\vec{a} | f(\vec{a}) = 1\}$ gives a gadget, and the setting $S = S_2 = \{0,1\}^k$ gives a strict gadget. Corollaries 3.8 and 3.9 give the required bound on the number of auxiliary variables; and the size of the LP then follows from the definition. ☐

To conclude this section, we mention some (obvious) facts that become relevant when searching for large gadgets. First, if $S' \subseteq S$, then the performance of an S'-partial gadget reducing f to $\mathcal{F}$ is also a lower bound on the performance of an S-partial gadget reducing f to $\mathcal{F}$. The advantage here is that the search for an S'-partial gadget may be much faster. Similarly, to get upper bounds on the performance of an S-partial gadget, one may use other witness matrices for S (rather than the canonical one), in particular ones with (many) fewer columns. This corresponds to making a choice of auxiliary variables not to be used in such a gadget.

4. Improved negative results.

4.1. MAX CUT. We begin by showing an improved hardness result for the MAX CUT problem. It is not difficult to see that no gadget per Definition 2.5 can reduce any member of PC to CUT: for any setting of the variables which satisfies (2.2), the complementary setting has the opposite parity (so that it must be subject to inequality (2.3)), but the values of all the CUT constraints are unchanged, so that the gadget's value is still α, violating (2.3). Following [2], we use instead the fact that MAX CUT and MAX CUT/0 are equivalent with respect to approximation as shown below.

Proposition 4.1. MAX CUT *is equivalent to* MAX CUT/0. *Specifically, given an instance $\mathcal{I}$ of either problem, we can create an instance $\mathcal{I}'$ of the other with the same optimum and with the feature that an assignment satisfying constraints of total weight W to the latter can be transformed into an assignment satisfying constraints of the same total weight in $\mathcal{I}$.*

Proof. The reduction from MAX CUT to MAX CUT/0 is trivial, since the family CUT/0 contains CUT, and thus the identity map provides the required reduction.

In the reverse direction, given an instance $(\vec{X}, \vec{C}, \vec{w})$ of MAX CUT/0 with n variables and m clauses, we create an instance $(\vec{X}', \vec{C}', \vec{w})$ of MAX CUT with $n+1$ variables and m clauses. The variables are simply the variables $\vec{X}$ with one additional variable called 0. The constraints of $\vec{C}$ are transformed as follows. If the constraint is a CUT constraint on variables X_i and X_j it is retained as is. If the constraint is $T(X_i)$ it is replaced with the constraint $\mathrm{CUT}(X_i, 0)$. Given an assignment $\vec{a}$ to the vector $\vec{X}'$, notice that its complement also satisfies the same number of constraints in $\mathcal{I}'$. We pick the one among the two that sets the variable 0 to 0, and then observe that the induced assignment to $\vec{X}$ satisfies the corresponding clauses of $\mathcal{I}$. □

Thus we can look for reductions to CUT/0. Notice that the CUT/0 constraint family is hereditary, since identifying the two variables in a CUT constraint yields the constant function 0. Thus by Theorem 3.12, if there is an α-gadget reducing PC_0 to CUT/0, then there is an α-gadget with at most 13 auxiliary variables (16 variables in all). Only $\binom{16}{2} = 120$ CUT constraints are possible on 16 variables. Since we only need to consider the cases when $Y_1 = 0$, we can construct an LP as above with $2^{16-1} + 4 = 32{,}772$ constraints to find the optimal α-gadget reducing PC_0 to CUT/0. An LP of the same size can similarly be constructed to find a gadget reducing PC_1 to CUT/0.

Lemma 4.2. *There exists an* 8*-gadget reducing* PC_0 *to* CUT/0, *and it is optimal and strict.*

We show the resulting gadget in Figure 4.1 as a graph. The primary variables are labeled x_1, x_2, and x_3, while 0 is a special variable. The unlabeled vertices are auxiliary variables. Each constraint of nonzero weight is shown as an edge. An edge between the vertex 0 and some vertex x corresponds to the constraint $T(x)$. Any other edge between x and y represents the constraint $\mathrm{CUT}(x, y)$. Note that some of the 13 possible auxiliary variables do not appear in any positive weight constraint and thus are omitted from the graph. All nonzero weight constraints have weight .5.

By the same methodology, we can prove the following.

Lemma 4.3. *There exists a* 9*-gadget reducing* PC_1 *to* CUT/0, *and it is optimal and strict.*

The gadget is similar to the previous one, but the old vertex 0 is renamed Z, and a new vertex labeled 0 is joined to Z by an edge of weight 1.

The two lemmas along with Proposition 4.1 above imply the following theorem.

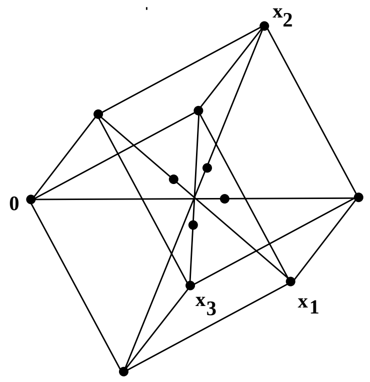


FIG. 4.1. *8-gadget reducing* PC$_0$ *to* CUT. *Every edge has weight .5. The auxiliary variable which is always* 0 *is labeled* 0.

THEOREM 4.4. *For every* $\epsilon > 0$, MAX CUT *is hard to approximate to within* $16/17 + \epsilon$.

Proof. Combining Theorem 2.8 with Lemmas 4.2 and 4.3, we find that MAX CUT/0 is hard to approximate to within $16/17 + \epsilon$. The theorem then follows from Proposition 4.1. ◻

RMBC *gadgets.* Finding RMBC gadgets was more difficult. We discuss this point since it leads to ideas that can be applied in general when finding large gadgets. Indeed, it turned out that we couldn't exactly apply the technique above to find an optimal gadget reducing, say, RMBC$_{00}$ to CUT/0. (Recall that the RMBC$_{00}(a_1, a_2, a_3, a_4)$ is the function $(a_3, a_4)[a_1] \overset{?}{=} a_2$.) Since there are 8 satisfying assignments to the 4 variables of the RMBC$_{00}$ constraint, by Theorem 3.12, we would need to consider $2^8 - 4 = 252$ auxiliary variables, leading to an LP with $2^{252} + 8$ constraints, which is somewhat beyond the capacity of current computing machines. To overcome this difficulty, we observed that for the RMBC$_{00}$ function, the value of a_4 is irrelevant when $a_1 = 0$ and the value of a_3 is irrelevant when $a_1 = 1$. This led us to try only restricted witness functions for which $\vec{b}(0, a_2, a_3, 0) = \vec{b}(0, a_2, a_3, 1)$ and $\vec{b}(1, a_2, 0, a_4) = \vec{b}(1, a_2, 1, a_4)$ (dropping from the witness matrix columns violating the above conditions), even though it is not evident a priori that a gadget with a witness function of this form exists. The number of distinct variable columns that such a witness matrix can have is at most 16. Excluding auxiliary variables identical to a_1 or a_2, we considered gadgets with at most 14 auxiliary variables. We then created an LP with $\binom{18}{2} = 153$ variables and $2^{18-1} + 8 = 131{,}080$ constraints. The result of the LP was that there exists an 8-gadget with constant 0 reducing RMBC$_{00}$ to CUT, and that it is strict. Since we used a restricted witness function, the LP does not prove that this gadget is optimal.

However, lower bounds can be established through construction of optimal S-partial gadgets. If S is a subset of the set of satisfying assignments of RMBC$_{00}$, then its defining equalities and inequalities (see Definition 3.3) are a subset of those for a gadget, and thus the performance of the partial gadget is a lower bound for that of a true gadget.

In fact, we have always been lucky with the latter technique, in that some choice of the set S has always yielded a lower bound and a matching gadget. In particular, for reductions from RMBC to CUT, we have the following result.

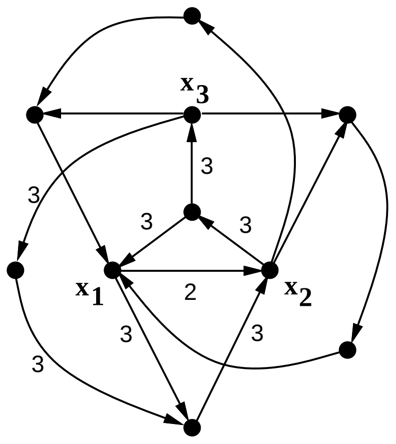


FIG. 4.2. 8*-gadget reducing* PC_0 *to* DICUT*. Edges have weight* 1 *except when marked otherwise.*

THEOREM 4.5. *There is an* 8*-gadget reducing* RMBC_{00} *to* CUT/0*, and it is optimal and strict; there is an* 8*-gadget reducing* RMBC_{01} *to* CUT/0*, and it is optimal and strict; there is a* 9*-gadget reducing* RMBC_{10} *to* CUT/0*, and it is optimal and strict; and there is a* 9*-gadget reducing* RMBC_{11} *to* CUT/0*, and it is optimal and strict.*

Proof. In each case, for some set S of satisfying assignments, an optimal S-partial gadget also happens to be a true gadget, and strict. In the same notation as in Definition 2.6, the appropriate sets S of 4-tuples (a, b, c, d) are, for RMBC_{00}, $S = \{0001, 1101, 0110, 1010\}$; for RMBC_{01}, $S = \{0000, 1100, 0111, 1011\}$; for RMBC_{10}, $S = \{0100, 1000, 0011, 1111\}$; and for RMBC_{11}, $S = \{0101, 1001, 0010, 1110\}$. □

4.2. MAX DICUT. As in the previous subsection, we observe that if there exists an α-gadget reducing an element of PC to DICUT, there exists an α-gadget with 13 auxiliary variables. This leads to LPs with $16 \cdot 15$ variables (one for each possible DICUT constraint, corresponding to a directed edge) and $2^{16} + 4 = 65{,}540$ linear constraints. The solution to the LPs gives the following.

LEMMA 4.6. *There exist* 6.5*-gadgets reducing* PC_0 *and* PC_1 *to* DICUT*, and they are optimal and strict.*

The PC_0 gadget is shown in Figure 4.2. Again x_1, x_2, and x_3 refer to the primary variable and an edge from x to y represents the constraint $\neg x \wedge b$. The PC_1 gadget is similar but has all edges reversed.

THEOREM 4.7. *For every* $\epsilon > 0$, MAX DICUT *is hard to approximate to within* $12/13 + \epsilon$.

RMBC *gadgets.* As with the reductions to CUT/0, reductions from the RMBC family members to DICUT can be done by constructing optimal S-partial gadgets, and again (with fortuitous choices of S) these turn out to be true gadgets, and strict.

THEOREM 4.8. *There is a* 6*-gadget reducing* RMBC_{00} *to* DICUT*, and it is optimal and strict; there is a* 6.5*-gadget reducing* RMBC_{01} *to* DICUT*, and it is optimal and strict; there is a* 6.5*-gadget reducing* RMBC_{10} *to* DICUT*, and it is optimal and strict; and there is a* 7*-gadget reducing* RMBC_{11} *to* DICUT*, and it is optimal and strict.*

Proof. Using, case by case, the same sets S as in the proof of Theorem 4.5, again yields in each case an optimal S-partial gadget that also happens to be a true, strict gadget. □

4.3. MAX 2-CSP. For reducing an element of PC to the 2CSP family we need consider only four auxiliary variables, for a total of seven variables. There are two nonconstant functions on a single variable, and twelve nonconstant functions on pairs of variables, so that there are $2\cdot 7+12\cdot\binom{7}{2}=266$ functions to consider overall. We can again set up an LP with a variable per function and $2^7+4=132$ linear constraints. We obtain the following lemma.

LEMMA 4.9. *There exist* 5*-gadgets reducing* PC_0 *and* PC_1 *to* 2CSP, *and they are optimal and strict.*

The gadget reducing PC_0 to 2CSP is the following:

$$\begin{array}{rrrr} X_1\wedge\neg Y_1, & X_1\wedge Y_2, & \neg X_1\wedge Y_3, & \neg X_1\wedge Y_4,\\ X_2\wedge\neg Y_1, & \neg X_2\wedge Y_2, & X_2\wedge Y_3, & \neg X_2\wedge Y_4,\\ \neg X_3\wedge Y_1, & X_3\wedge\neg Y_2, & X_3\wedge\neg Y_3, & \neg X_3\wedge\neg Y_4. \end{array}$$

The gadget reducing PC_1 to 2CSP can be obtained from this one by complementing all the occurrences of X_1.

THEOREM 4.10. *For every* $\epsilon>0$, MAX 2CSP *is hard to approximate to within* $9/10+\epsilon$.

MAX 2CSP can be approximated to within .859 [5]. The above theorem has implications for probabilistically checkable proofs. Reversing the well-known reduction from constraint-satisfaction problems to probabilistically checkable proofs (see [1]),[3] Theorem 4.10 yields the following theorem.

THEOREM 4.11. *For any* $\epsilon>0$, *constants* c *and* s *exist such that* $\mathrm{NP}\subseteq\mathrm{PCP}_{c,s}[\log,2]$ *and* $c/s>10/9-\epsilon$.

The previously known gap between the completeness and soundness achievable reading two bits was 74/73 [2]. It would be $22/21-\epsilon$ using Håstad's result [9] in combination with the argument of [2]. Actually the reduction from constraint-satisfaction problems to probabilistically checkable proofs is reversible, and this will be important in section 7.

RMBC *gadgets.*

THEOREM 4.12. *For each element of* RMBC, *there is a* 5*-gadget reducing it to* 2CSP, *and it is optimal and strict.*

Proof. Using the same selected assignments as in Theorems 4.5 and 4.8 again yields lower bounds and matching strict gadgets. □

5. Interlude: Methodology. Despite their seeming variety, all the gadgets in this paper were computed using a single program (in the language APL2) to generate an LP and call upon OSL (the IBM Optimization Subroutine Library) to solve it. This "gadget-generating" program takes several parameters.

The *source function* f is specified explicitly by a small program that computes f.

The *target family* $\mathcal{F}$ is described by a single function, implemented as a small program, and applied to all possible clauses of specified lengths and symmetries. The symmetries are chosen from among the following: whether clauses are unordered or ordered; whether their variables may be complemented; and whether they may include the constants 0 or 1. For example, a reduction to MAX CUT/0 would take as $\mathcal{F}$ the function $x_1\oplus x_2$, applied over unordered binomial clauses, in which complementation is not allowed but the constant 0 is allowed. This means of describing $\mathcal{F}$ is relatively intuitive and has never restricted us, even though it is not completely general. Finally,

[3]The reverse connection is by now a folklore result and may be proved along the lines of [2, Proposition 10.3, Part (3)].

we specify an arbitrary set S of selected assignments, which allows us to search for S-partial gadgets (recall Definition 3.3). From (3.2) and (3.4), each selected assignment $\vec{a}$ generates a constraint that $(\exists\vec{b}) : \sum_j w_j C_j(\vec{a},\vec{b}) = \alpha - (1 - f(\vec{a}))$. Selecting all *satisfying* assignments of f reproduces the set of constraints (2.2) for an α-gadget, while selecting *all* assignments reproduces the set of constraints (2.2) and (2.4) for a strict α-gadget.

Selected assignments are specified explicitly; by default, to produce an ordinary gadget, they are the satisfying assignments of f. The canonical witness for the selected set of assignments is generated by our program as governed by Definition 3.6. Notice that the definition of the witness depends on whether $\mathcal{F}$ is complementation closed or not, and this is determined by the explicitly specified symmetries.

To facilitate the generation of restricted witness matrices, we have also made use of a "don't-care" state (in lieu of 0 or 1) to reduce the number of selected assignments. For example, in reductions from RMBC_{00}, we have used selected assignments of $(00*0)$, $(011*)$, $(10*0)$, and $(11*1)$. The various LP constraints must be satisfied for both values of any don't-care, while the witness function must not depend on the don't-care values. So in this example, use of a don't-care reduces the number of selected assignments from 8 to 4, reduces the number of auxiliary variables from about 2^8 to 2^4 (ignoring duplications of the 4 primary variables, or any symmetries), and reduces the number of constraints in the LP from 2^{2^8} (about 10^{77}) to 2^{2^4} (a more reasonable 65,536). Use of don't-cares provides a technique complementary to selecting a subset of all satisfying assignments in that if the LP is feasible it provides an upper bound and a gadget, but the gadget may not be optimal.

In practice, selecting a subset of satisfying assignments has been by far the more useful of the two techniques; so far we have always been able to choose a subset which produces a lower bound and a gadget to match.

After constructing and solving an LP, the gadget-generating program uses brute force to make an independent verification of the gadget's validity, performance, and strictness.

The hardest computations were those for gadgets reducing from RMBC; on an IBM Risc System/6000 model 43P-240 workstation, running at 233MHz, these took up to half an hour and used 500 MB or so of memory. However, the strength of [9] makes PC virtually the sole source function of contemporary interest, and all the reductions from PC are easy; they use very little memory, and run in seconds on an ordinary 233MHz Pentium processor.

6. Improved positive results.

In this section we show that we can use gadgets to improve approximation algorithms. In particular, we look at MAX 3SAT, and a variation, MAX 3ConjSAT, in which each clause is a conjunction (rather than a disjunction) of three literals. An improved approximation algorithm for the latter problem leads to improved results for probabilistically checkable proofs in which the verifier examines only three bits. Both of the improved approximation algorithms rely on strict gadgets reducing the problem to MAX 2SAT. We begin with some notation.

DEFINITION 6.1. *A* (β_1, β_2) approximation algorithm *for* MAX 2SAT *is an algorithm which receives as input an instance with unary clauses of total weight* m_1 *and binary clauses of total weight* m_2, *and two reals* $u_1 \leq m_1$ *and* $u_2 \leq m_2$, *and produces reals* $s_1 \leq u_1$ *and* $s_2 \leq u_2$ *and an assignment satisfying clauses of total weight at least* $\beta_1 s_1 + \beta_2 s_2$. *If there exists an optimum solution that satisfies unary clauses of weight no more than* u_1 *and binary clauses of weight no more than* u_2, *then there is a guarantee that no assignment satisfies clauses of total weight more than* $s_1 + s_2$.

That is, supplied with a pair of "upper bounds" u_1, u_2, a (β_1, β_2) approximation algorithm produces a single upper bound of $s_1 + s_2$, along with an assignment respecting a lower bound of $\beta_1 s_1 + \beta_2 s_2$.

LEMMA 6.2 (Feige and Goemans [5]). *There exists a polynomial-time* $(.976, .931)$-*approximation algorithm for* MAX 2SAT.

6.1. MAX 3SAT. In this section we show how to derive an improved approximation algorithm for MAX 3SAT. By restricting techniques in [8] from MAX SAT to MAX 3SAT and using a .931-approximation algorithm for MAX 2SAT due to Feige and Goemans [5], one can obtain a .7704-approximation algorithm for MAX 3SAT. The basic idea of [8] is to reduce each clause of length 3 to the three possible subclauses of length 2, give each new length-2 clause one-third the original weight, and then apply an approximation algorithm for MAX 2SAT. This approximation algorithm is then "balanced" with another approximation algorithm for MAX 3SAT to obtain the result. Here we show that by using a strict gadget to reduce 3SAT to MAX 2SAT, a good (β_1, β_2)-approximation algorithm for MAX 2SAT leads to a .801-approximation algorithm for MAX 3SAT.

LEMMA 6.3. *If for every $f \in$* E3SAT *there exists a strict α-gadget reducing f to* 2SAT, *there exists a (β_1, β_2)-approximation algorithm for* MAX 2SAT, *and $\alpha \geq 1 + \frac{(\beta_1 - \beta_2)}{2(1-\beta_2)}$, then there exists a ρ-approximation algorithm for* MAX 3SAT *with*

$$\rho = \frac{1}{2} + \frac{(\beta_1 - 1/2)(3/8)}{(\alpha - 1)(1 - \beta_2) + (\beta_1 - \beta_2) + (3/8)}.$$

Proof. Let ϕ be an instance of MAX 3SAT with length-1 clauses of total weight m_1, length-2 clauses of total weight m_2, and length-3 clauses of total weight m_3. We use the two algorithms listed below, getting the corresponding upper and lower bounds on the number of satisfiable clauses:

- Random: We set each variable to 1 with probability 1/2. This gives a solution of weight at least $m_1/2 + 3m_2/4 + 7m_3/8$.
- Semidefinite programming: We use the strict α-gadget to reduce every length-3 clause to length-2 clauses. This gives an instance of MAX 2SAT. We apply the (β_1, β_2)-approximation algorithm with parameters $u_1 = m_1$ and $u_2 = m_2 + \alpha m_3$ to find an approximate solution to this problem. The approximation algorithm gives an upper bound $s_1 + s_2$ on the weight of any solution to the MAX 2SAT instance and an assignment of weight $\beta_1 s_1 + \beta_2 s_2$. When translated back to the MAX 3SAT instance, the assignment has weight at least $\beta_1 s_1 + \beta_2 s_2 - (\alpha - 1)m_3$. Furthermore, $s_1 \leq m_1$, $s_2 \leq m_2 + \alpha m_3$, and the maximum weight satisfiable in the MAX 3SAT instance is at most $s_1 + s_2 - (\alpha - 1)m_3$.

The performance guarantee of the algorithm which takes the better of the two solutions is at least

$$\rho_1 \overset{\text{def}}{=} \min_{\substack{s_1 \leq m_1 \\ s_2 \leq m_2 + \alpha m_3}} \frac{\max\{m_1/2 + 3m_2/4 + 7m_3/8, \quad \beta_1 s_1 + \beta_2 s_2 - (\alpha - 1)m_3\}}{s_1 + s_2 - (\alpha - 1)m_3}.$$

We now define a sequence of simplifications which will help prove the bound:

$$\rho_2 \overset{\text{def}}{=} \min_{\substack{t_1 \leq m_1 \\ t_2 \leq m_2 + m_3}} \frac{1}{t_1 + t_2} \max\{ \begin{array}{l} m_1/2 + 3m_2/4 + 7m_3/8, \\ \beta_1 t_1 + \beta_2 t_2 - (1 - \beta_2)(\alpha - 1)m_3\}, \end{array}$$

$$\rho_3 \overset{\text{def}}{=} \min_{\substack{t_1 \le m_1 \\ t_2 \le m_2 + m_3}} \frac{1}{t_1+t_2} \max\{ \begin{array}{l} t_1/2 + 3t_2/4 + m_3/8, \\ t_1/2 + 7m_3/8, \\ \beta_1 t_1 + \beta_2 t_2 - (1-\beta_2)(\alpha-1)m_3\}, \end{array}$$

$$\rho_4 \overset{\text{def}}{=} \min_{t_2 \le t} \frac{1}{t} \max\{ \begin{array}{l} t/2 + t_2/4 + m_3/8, \\ t/2 - t_2/2 + 7m_3/8, \\ \beta_1 t - (\beta_1 - \beta_2)t_2 - (1-\beta_2)(\alpha-1)m_3\}, \end{array}$$

$$\rho_5 \overset{\text{def}}{=} \frac{1}{2} + \left(\frac{\frac{3}{8}(\beta_1 - \frac{1}{2})}{(1-\beta_2)(\alpha-1) + (\beta_1-\beta_2) + \frac{3}{8}} \right).$$

To finish the proof of the lemma, we claim that

$$\rho_1 \ge \rho_2 \ge \cdots \ge \rho_5.$$

To see this, notice that the first inequality follows from the substitution of variables $t_1 = s_1$, $t_2 = s_2 - (\alpha-1)m_3$. The second follows from the fact that setting m_1 to t_1 and m_2 to $\max\{0, t_2 - m_3\}$ only reduces the numerator. The third inequality follows from setting $t = t_1 + t_2$. The fourth is obtained by substituting a convex combination of the arguments instead of max and then simplifying. The convex combination takes a θ_1 fraction of the first argument, θ_2 of the second, and θ_3 of the third, where

$$\theta_1 = \frac{\frac{2}{3}(1-\beta_2)(\alpha-1) + \frac{7}{6}(\beta_1-\beta_2)}{(1-\beta_2)(\alpha-1) + (\beta_1-\beta_2) + \frac{3}{8}},$$

$$\theta_2 = \frac{\frac{1}{3}(1-\beta_2)(\alpha-1) - \frac{1}{6}(\beta_1-\beta_2)}{(1-\beta_2)(\alpha-1) + (\beta_1-\beta_2) + \frac{3}{8}},$$

and

$$\theta_3 = \frac{\frac{3}{8}}{(1-\beta_2)(\alpha-1) + (\beta_1-\beta_2) + \frac{3}{8}}.$$

Observe that $\theta_1 + \theta_2 + \theta_3 = 1$ and that the condition on α guarantees that $\theta_2 \ge 0$. ☐

Remark 6.4. The analysis given in the proof of the above lemma is tight. In particular for an instance with m clauses such that

$$m_3 \overset{\text{def}}{=} m \frac{\beta_1 - 1/2}{(1-\beta_2)(\alpha-1) + (\beta_1-\beta_2) + 3/8},$$

$m_1 \overset{\text{def}}{=} m - m_3$, $m_2 \overset{\text{def}}{=} 0$, $s_1 = m_1$, and $s_2 = \alpha m_3$, it is easy to see that $\rho_1 = \rho_5$.

The following lemma gives the strict gadget reducing functions in E3SAT to 2SAT. Notice that finding strict gadgets is almost as forbidding as finding gadgets for RMBC, since there are eight existential constraints in the specification of a gadget. This time we relied instead on luck. We looked for an S-partial gadget for the set $S = \{111, 100, 010, 001\}$ and found an S-partial 3.5-gadget that turned out to be a gadget! Our choice of S was made judiciously, but we could have afforded to run through all $\binom{8}{4}$ sets S of size 4 in the hope that one would work.

LEMMA 6.5. *For every function $f \in$ E3SAT, there exists a strict (and optimal) 3.5-gadget reducing f to* 2SAT.

Proof. Since 2SAT is complementation closed, it is sufficient to present a 3.5-gadget reducing $(X_1 \vee X_2 \vee X_3)$ to 2SAT. The gadget is $X_1 \vee X_3, \neg X_1 \vee \neg X_3, X_1 \vee \neg Y, \neg X_1 \vee Y, X_3 \vee \neg Y, \neg X_3 \vee Y, X_2 \vee Y$, where every clause except the last has weight 1/2, and the last clause has weight 1. □

Combining Lemmas 6.2, 6.3, and 6.5 we get a .801-approximation algorithm.

THEOREM 6.6. MAX 3SAT *has a polynomial-time .801-approximation algorithm.*

6.2. MAX 3-ConjSAT. We now turn to the MAX 3ConjSAT problem. The analysis is similar to that of Lemma 6.3.

LEMMA 6.7. *If for every $f \in$ 3ConjSAT there exists a strict $(\alpha_1 + \alpha_2)$-gadget reducing f to 2SAT composed of α_1 length-1 clauses and α_2 length-2 clauses and there exists a (β_1, β_2) approximation algorithm for MAX 2SAT, then there exists a ρ approximation algorithm for MAX 3ConjSAT with*

$$\rho = \frac{\frac{1}{8}\beta_1}{\frac{1}{8} + (1-\alpha_1)(\beta_1 - \beta_2) + (1-\beta_2)(\alpha_1 + \alpha_2 - 1)}$$

provided $\alpha_1 + \alpha_2 > 1 + 1/8(1-\beta_2)$.

Proof. Let ϕ be an instance of MAX 3ConjSAT with constraints of total weight m. As in the MAX 3SAT case, we use two algorithms and take the better of the two solutions:

- Random: We set every variable to 1 with probability half. The total weight of satisfied constraints is at least $m/8$.
- Semidefinite programming: We use the strict α-gadget to reduce any constraint to 2SAT clauses. This gives an instance of MAX 2SAT and we use the (β_1, β_2)-approximation algorithm with parameters $u_1 = \alpha_1 m$ and $u_2 = \alpha_2 m$. The algorithm returns an upper bound $s_1 + s_2$ on the total weight of satisfiable constraints in the MAX 2SAT instance, and an assignment of measure at least $\beta_1 s_1 + \beta_2 s_2$. When translated back to the MAX 3ConjSAT instance, the measure of the assignment is at least $\beta_1 s_1 + \beta_2 s_2 - (\alpha_1 + \alpha_2 - 1)m$. Furthermore, $s_1 \le \alpha_1 m$, $s_2 \le \alpha_2 m$, and the total weight of satisfiable constraints in the MAX 3ConjSAT instance is at most $s_1 + s_2 - (\alpha_1 + \alpha_2 - 1)m$.

Thus we get that the performance ratio of the algorithm which takes the better of the two solutions above is at least

$$\rho_1 \overset{\text{def}}{=} \min_{\substack{s_1 \le \alpha_1 m \\ s_2 \le \alpha_2 m}} \frac{\max\{m/8, \quad \beta_1 s_1 + \beta_2 s_2 - (\alpha_1 + \alpha_2 - 1)m\}}{s_1 + s_2 - (\alpha_1 + \alpha_2 - 1)m}.$$

We now define a sequence of simplifications which will help prove the bound:

$$\rho_2 \overset{\text{def}}{=} \min_{\substack{t_1 \le \alpha_1 m \\ t_2 \le (1-\alpha_1) m}} \frac{1}{t_1 + t_2} \max\{m/8, \quad \beta_1 t_1 + \beta_2 t_2 - (1-\beta_2)(\alpha_1 + \alpha_2 - 1)m\},$$

$$\rho_3 \overset{\text{def}}{=} \min_{\substack{t \le m \\ t_2 \le (1-\alpha_1) m}} \frac{1}{t} \max\{m/8, \quad \beta_1 t - (\beta_1 - \beta_2) t_2 - (1-\beta_2)(\alpha_1 + \alpha_2 - 1)m\},$$

$$\rho_4 \overset{\text{def}}{=} \min_{t \le m} \frac{1}{t} \max\{m/8, \quad \beta_1 t - ((1-\alpha_1)(\beta_1 - \beta_2) + (1-\beta_2)(\alpha_1 + \alpha_2 - 1))m\},$$

$$\rho_5 \stackrel{\text{def}}{=} \frac{\frac{1}{8}\beta_1}{\frac{1}{8} + (1-\alpha_1)(\beta_1 - \beta_2) + (1-\beta_2)(\alpha_1 + \alpha_2 - 1)}.$$

In order to prove the lemma, we claim that

$$\rho_1 \geq \rho_2 \geq \cdots \geq \rho_5.$$

To see this, observe that the first inequality follows from the substitution of variables $t_1 = s_1$ and $t_2 = s_2 - (\alpha_1 + \alpha_2 - 1)m$. The second follows from setting $t = t_1 + t_2$. The third inequality follows from the fact that setting t_2 to $(1-\alpha_1)m$ only reduces the numerator. The fourth is obtained by substituting a convex combination of the arguments instead of max and then simplifying. $\square$

The following gadget was found by looking for an S-partial gadget for $S = \{111, 110, 101, 011\}$.

Lemma 6.8. *For any $f \in$ 3ConjSAT there exists a strict (and optimal) 4-gadget reducing f to 2SAT. The gadget is composed of one length-1 clause and three length-2 clauses.*

Proof. Recall that 2SAT is complementation closed, and thus it is sufficient to exhibit a gadget reducing $f(a_1, a_2, a_3) = a_1 \wedge a_2 \wedge a_3$ to 2SAT. Such a gadget is Y, $(\neg Y \vee X_1)$, $(\neg Y \vee X_2)$, $(\neg Y \vee X_3)$, where all clauses have weight 1. The variables X_1, X_2, X_3 are primary variables and Y is an auxiliary variable. $\square$

Theorem 6.9. MAX 3ConjSAT *has a polynomial-time* .367*-approximation algorithm.*

It is shown by Trevisan [16, Theorem 18] that the above theorem has consequences for $\text{PCP}_{c,s}[\log, 3]$. This is because the computation of the verifier in such a proof system can be described by a decision tree of depth 3 for every choice of random string. Further, there is a 1-gadget reducing every function which can be computed by a decision tree of depth k to kConjSAT.

Corollary 6.10. $\text{PCP}_{c,s}[\log, 3] \subseteq \text{P}$ *provided that $c/s > 2.7214$.*

The previous best trade-off between completeness and soundness for polynomial-time PCP classes was $c/s > 4$ [16].

7. Lower bounds for gadget constructions.

In this section we shall show that some of the gadget constructions mentioned in this paper and in [2] are optimal, and we shall prove lower bounds for some other gadget constructions.

The following result is useful to prove lower bounds for the RMBC family.

Lemma 7.1. *If there exists an α-gadget reducing an element of* RMBC *to a complementation-closed constraint family $\mathcal{F}$, then there exists an α-gadget reducing all elements of* PC *to $\mathcal{F}$.*

Proof. If a family $\mathcal{F}$ is complementation closed, then an α-gadget reducing an element of PC (respectively, RMBC) to $\mathcal{F}$ can be modified (using complementations) to yield α-gadgets reducing all elements of PC (respectively RMBC) to $\mathcal{F}$. For this reason, we will restrict our analysis to PC_0 and RMBC_{00} gadgets. Note that, for any $a_1, a_2, a_3 \in \{0,1\}^3$, $\text{PC}_0(a_1, a_2, a_3) = 1$ if and only if $\text{RMBC}_{00}(a_1, a_2, a_3, \overline{a_3}) = 1$. Let Γ be an α-gadget over primary variables $x_1, \ldots, x_4$ and auxiliary variables $y_1, \ldots, y_n$ reducing RMBC to 2SAT. Let Γ' be the gadget obtained from Γ by imposing $x_4 \equiv \overline{x_3}$: it is immediate to verify that Γ' is an α-gadget reducing PC_0 to $\mathcal{F}$. $\square$

7.1. Reducing PC and RMBC to 2SAT.

Theorem 7.2. *If Γ is an α-gadget reducing an element of* PC *to* 2SAT, *then $\alpha \geq 11$.*

Proof. It suffices to consider PC_0. We prove that the optimum of (LP1) is at least 11. To this end, consider the dual program of (LP1). We have a variable $y_{\vec{a},\vec{b}}$ for any $\vec{a} \in \{0,1\}^3$ and any $\vec{b} \in \{0,1\}^4$, plus additional variables $\hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})}$ for any $\vec{a} : \mathrm{PC}(\vec{a}) = 1$, where $\vec{b}^{opt}$ is the "optimal" witness function defined in section 3. The formulation is

$$\begin{array}{ll}
\text{maximize} \quad \sum_{\vec{a},\vec{b}:\mathrm{PC}(\vec{a})=0} y_{\vec{a},\vec{b}} & \\
\text{subject to} & \\
\sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} \leq 1 + \sum_{\vec{a}:\mathrm{PC}(\vec{a})=1} y_{\vec{a},\vec{b}^{opt}(\vec{a})}, & \\
\sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} C_j(\vec{a},\vec{b}) \geq \sum_{\vec{a}:\mathrm{PC}(\vec{a})=1} \hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})} C_j(\vec{a},\vec{b}^{opt}(\vec{a})) & \forall j \in [98], \\
y_{\vec{a},\vec{b}} \geq 0 & (\forall \vec{a} \in \{0,1\}^3)\ (\forall \vec{b} \in \{0,1\}^4), \\
\hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})} \geq 0 & (\forall \vec{a} : \mathrm{PC}(\vec{a}) = 1),
\end{array}$$

(DUAL1).

There exists a feasible solution for (DUAL1) whose cost is 11. ∎

COROLLARY 7.3. *If* Γ *is an* α*-gadget reducing an element of* RMBC *to* 2SAT*, then* $\alpha \geq 11$.

7.2. Reducing PC and RMBC to SAT.

THEOREM 7.4. *If* Γ *is an* α*-gadget reducing an element of* PC *to* SAT*, then* $\alpha \geq 4$.

Proof. As in the proof of Theorem 7.2 we give a feasible solution to the dual to obtain the lower bound. The LP that finds the best gadget reducing PC_0 to SAT is similar to (LP1), the only difference being that a larger number N of clauses are considered, namely, $N = \sum_{i=1}^{7} \binom{7}{i} 2^i$. The dual program is then

$$\begin{array}{ll}
\text{maximize} \quad \sum_{\vec{a},\vec{b}:\mathrm{PC}(\vec{a})=0} y_{\vec{a},\vec{b}} & \\
\text{subject to} & \\
\sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} \leq 1 + \sum_{\vec{a}:\mathrm{PC}(\vec{a})=1} y_{\vec{a},\vec{b}^{opt}(\vec{a})}, & \\
\sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} C_j(\vec{a},\vec{b}) \geq \sum_{\vec{a}:\mathrm{PC}(\vec{a})=1} \hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})} C_j(\vec{a},\vec{b}^{opt}(\vec{a})) & \forall j \in [N], \\
y_{\vec{a},\vec{b}} \geq 0 & (\forall \vec{a} \in \{0,1\}^3)\ (\forall \vec{b} \in \{0,1\}^4), \\
\hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})} \geq 0 & (\forall \vec{a} : \mathrm{PC}(\vec{a}) = 1),
\end{array}$$

(DUAL2).

Consider now the following assignment of values to the variables of (DUAL2) (the unspecified values have to be set to zero):

$$\begin{array}{l}
(\forall \vec{a} : \mathrm{PC}(\vec{a}) = 1)\ \hat{y}_{\vec{a},\vec{b}^{opt}(\vec{a})} = \frac{3}{4}, \\
(\forall \vec{a} : \mathrm{PC}(\vec{a}) = 1)(\forall \vec{a}' : d(\vec{a},\vec{a}') = 1)\ y_{\vec{a}',\vec{b}^{opt}(\vec{a})} = \frac{1}{3},
\end{array}$$

where d is the Hamming distance between binary sequences. It is possible to show that this is a feasible solution for (DUAL2) and it is immediate to verify that its cost is 4. ∎

COROLLARY 7.5. *If* Γ *is an* α*-gadget reducing an element* RMBC *to* SAT*, then* $\alpha \geq 4$.

7.3. Reducing *k*SAT to *l*SAT.

Let k and l be any integers $k > l \geq 3$. The standard reduction from EkSAT to lSAT can be seen as a $\lceil (k-2)/(l-2) \rceil$-gadget. In this section we shall show that this is asymptotically the best possible. Note that since lSAT is complementation closed we can restrict ourselves to considering just one constraint function of EkSAT, say $f(a_1, \ldots, a_k) \equiv \bigvee_i a_i$.

THEOREM 7.6. *For any $k > l > 2$, if Γ is an α-gadget reducing f to $lSAT$, then $\alpha \geq k/l$.*

Proof. We can write an LP whose optimum gives the smallest α such that an α-gadget exists reducing f to lSAT. Let b be the witness function used to formulate this LP. We can assume that b is 2^{2^k}-ary and we let $K = 2^{2^k}$. Also let N be the total number of constraints from lSAT that can be defined over $k + K$ variables. Assume some enumeration $C_1, \ldots, C_N$ of such constraints. The dual LP is

$$\begin{array}{ll}
\text{maximize} & \sum_{\vec{a},\vec{b}:\mathrm{PC}(\vec{a})=0} y_{\vec{a},\vec{b}} \\
\text{subject to} & \\
 & \sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} \leq 1 + \sum_{\vec{a}:f(\vec{a})=1} y_{\vec{a},\vec{b}^{kSAT-lSAT}(\vec{a})}, \\
\forall j \in [N] & \sum_{\vec{a},\vec{b}} y_{\vec{a},\vec{b}} C_j(\vec{a},\vec{b}) \geq \sum_{\vec{a}:f(\vec{a})=1} \hat{y}_{\vec{a},\vec{b}^{kSAT-lSAT}(\vec{a})} C_j(\vec{a}, \\
 & \vec{b}^{kSAT-lSAT}(\vec{a})), \\
\forall \vec{a} \in \{0,1\}^k,\ \forall \vec{b} \in \{0,1\}^K & y_{\vec{a},\vec{b}} \geq 0, \\
\forall \vec{a} : f(\vec{a}) = 1 & \hat{y}_{\vec{a},\vec{b}^{kSAT-lSAT}(\vec{a})} \geq 0
\end{array}$$

(DUAL3).

The witness function $\vec{b}^{kSAT-lSAT}$ is an "optimal" witness function for gadgets reducing kSAT to lSAT.

Let $A_k \subset \{0,1\}^k$ be the set of binary k-ary strings with exactly one nonzero component (note that $|A_k| = k$). Also let $\vec{0}$ (respectively, $\vec{1}$) be the k-ary string all of whose components are equal to 0 (respectively, 1). The following is a feasible solution for (DUAL3) whose cost is k/l. We only specify nonzero values:

$$\begin{array}{ll}
(\forall \vec{a} \in A_k) & \hat{y}_{\vec{a},\vec{b}^{kSAT-lSAT}(\vec{a})} = 1/l, \\
(\forall \vec{a} \in A_k) & y_{\vec{0},\vec{b}^{kSAT-lSAT}(\vec{a})} = 1/l, \\
(\forall \vec{a} \in A_k) & y_{\vec{1},\vec{b}^{kSAT-lSAT}(\vec{a})} = 1/k\ . \qquad \square
\end{array}$$

In view of the above lower bound, a gadget cannot provide an approximation-preserving reduction from MAX SAT to MAX kSAT. More generally, there cannot be an approximation-preserving gadget reduction from MAX SAT to, say, MAX $(\log n)$SAT. In partial contrast with this lower bound, Khanna et al. [13] have given an approximation-preserving reduction from MAX SAT to MAX 3SAT and Crescenzi and Trevisan [4] have provided a *tight* reduction between MAX SAT and MAX $(\log n)$SAT, showing that the two problems have the same approximation threshold.

Acknowledgments. We thank Pierluigi Crescenzi and Oded Goldreich for several helpful suggestions and remarks. We are grateful to John Forrest and David Jensen for their assistance in efficiently solving large LPs. We thank Howard Karloff and Uri Zwick for pointing out the error in the earlier version of this paper and the counterexample to our earlier claim. We thank the anonymous referees for their numerous comments and suggestions leading to the restructuring of section 3.

REFERENCES

[1] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and the hardness of approximation problems*, J. Assoc. Comput. Mach., 45 (1998), pp. 501–555.
[2] M. BELLARE, O. GOLDREICH, AND M. SUDAN, *Free bits, PCPs and nonapproximability—towards tight results*, SIAM J. Comput., 27 (1998), pp. 804–915.

[3] P. Crescenzi, R. Silvestri, and L. Trevisan, *On weighted vs. unweighted versions of combinatorial optimization problems*, Inform. and Comput., to appear.

[4] P. Crescenzi and L. Trevisan, *MAX NP-completeness made easy*, Theoret. Comput. Sci., 225 (1999), pp. 65–79.

[5] U. Feige and M. X. Goemans, *Approximating the value of two prover proof systems, with applications to MAX 2SAT and MAX DICUT*, in Proceedings of the Third Israel Symposium on Theory of Computing and Systems, Tel Aviv, Israel, 1995, pp. 182–189.

[6] M. Garey, D. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems*, Theoret. Comput. Sci., 1 (1976) pp. 237–267.

[7] M. X. Goemans and D. P. Williamson, *New 3/4-approximation algorithms for the maximum satisfiability problem*, SIAM J. Discrete Math., 7 (1994), pp. 656–666.

[8] M. X. Goemans and D. P. Williamson, *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*, J. Assoc. Comput. Mach., 42 (1995), pp. 1115–1145.

[9] J. Håstad, *Some optimal inapproximability results*, in Proceedings of the 29th ACM Symposium on Theory of Computing, El Paso, TX, 1997, pp. 1–10.

[10] H. Karloff and U. Zwick, *A 7/8-approximation algorithm for MAX* 3SAT?, in Proceedings of the 38th IEEE Symposium on Foundations of Computer Science, Miami, FL, 1997, pp. 406–415.

[11] H. Karloff and U. Zwick, *personal communication*, 1996.

[12] R. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, Advances in Computing Research, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[13] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani, *On syntactic versus computational views of approximability*, SIAM J. Comput., 28 (1999), pp. 164–191.

[14] T. Ono, T. Hirata, and T. Asano, *Approximation algorithms for the maximum satisfiability problem*, in Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory, Reykjavík, Iceland, Lecture Notes in Comput. Sci. 1097, Springer-Verlag, 1996, pp. 100–111.

[15] C. Papadimitriou and M .Yannakakis, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[16] L. Trevisan, *Parallel approximation algorithms using positive linear programming*, Algorithmica, 21 (1998), pp. 72–88.

[17] L. Trevisan, G. B. Sorkin, M. Sudan, and D. P. Williamson, *Gadgets, approximation, and linear programming*, in Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 617–626.

[18] M. Yannakakis, *On the approximation of maximum satisfiability*, J. Algorithms, 17 (1994), pp. 475–502.

[19] U. Zwick, *Approximation algorithms for constraint satisfaction problems involving at most three variables per constraint*, in Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1998, pp. 201–210.